INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Improving Code Merging Accuracy with Transformations and Member Identity

André Duarte Rocha Teles

Master's in Computer Science and Engineering

Supervisor:
Prof. Dr. André L. Santos, Assistant Professor
Iscte – Instituto Universitário de Lisboa

September, 2023

TECNOLOGIAS
E ARQUITETURA

Department of Information Science and Technology

Improving Code Merging Accuracy with Transformations and Member Identity

André Duarte Rocha Teles

Master's in Computer Science and Engineering

Supervisor:
Prof. Dr. André L. Santos, Assistant Professor
Iscte – Instituto Universitário de Lisboa

September, 2023

Acknowledgements

While developing this work, I had a lot of support and help.

I would like to thank my supervisor, Professor André Santos, for all his support, enthusiasm, suggestions, close follow-up and belief in my abilities. I would also like to thank him for helping me in the process of finding a job, especially in developing a better *Curriculum Vitae*, which is beyond the scope of this work.

I would like to thank my parents, Duarte and Mercês, for all the support, for providing all the conditions to develop this work in the best way possible, and for being reasonable in some weeks of a lot of work when the deadlines of the research paper were tight.

I would like to thank my grandmother, Conceição, for keeping me company and taking care of me during some working afternoons in her house during all these weeks, and for always praying for me.

I would like to thank my girlfriend, Isabel, for motivating me to do better and better, for being proud of me, for being understanding during the stressful weeks of less attention, and for always finding activities together to clear my head.

Finally, I would like to thank the rest of my family for their concern and for all the good moments we have shared.

All the words would not be enough to express my gratitude and happiness for sharing my life with these people.

Resumo

Convencionalmente, o processo de *merging* de ficheiros de código é realizado de forma não estruturada, utilizando algoritmos genéricos de *merge* baseados em linhas (e.g., diff3) que desconhecem a sintaxe e a semântica da linguagem de programação em que o código está escrito, resultando em vários conflitos que poderiam ser evitados. As técnicas de *merge* estruturada e semiestruturada reduzem o número de conflitos, pois têm em consideração a sintaxe da linguagem de programação. No entanto, ainda há problemas a serem resolvidos em relação a falsos positivos (conflitos que poderiam ser evitados, e.g., em mudanças na assinatura de membros) e falsos negativos (conflitos que não são detectados resultando em código não-compilável).

Esta dissertação apresenta uma técnica de *merge* que combina estratégias semiestruturadas e baseadas em transformações. Desenvolvemos o Jaid, um protótipo de ferramenta de *merge* para Java, com base no pressuposto de que os elementos estruturais do código evoluem com UUIDs ”anexados” (identidade). Embora isto tenha inconvenientes e possa exigir editores de código dedicados para uma boa usabilidade, tem a vantagem de permitir que os processos de *merge* detectem com precisão a renomeação, a deslocação e a referência de elementos do código. O Jaid tem em conta aspectos sintácticos e semânticos para fazer *merge* baseado em transformações, tendo como principal diferença, em relação a abordagens anteriores, basear-se na identidade para localizar e referenciar elementos do código. Realizámos uma experiência com 100 cenários de *merge* de três projectos *open-source* para testar a técnica e avaliar a sua viabilidade na prática.

Palavras-chave: Software merging, Sistemas de controlo de versões, Transformações, Conflitos, Identidade

Abstract

Conventionally, merging code files is performed in an unstructured manner, using generic line-based merging algorithms (e.g., diff3) that are unaware of the syntax and semantics of the programming language in which the merged code is written, resulting in several conflicts that could be avoided. Structured and semistructured merging techniques take into consideration the programming language syntax and are capable of merging processes that lead to fewer conflicts. However, there are still issues to be solved regarding false positives (conflicts that could be avoided, e.g., member signature changes) and false negatives (conflicts that go undetected resulting in non-compilable code).

This dissertation presents a merging technique that combines semistructured and transformation-based strategies, where conflict detection is aware of syntactic and semantic aspects of the programming language. We developed Jaid, a prototype merging tool for Java based on the assumption that code structural elements evolve with “attached” UUIDs (identity). While this has drawbacks and may require dedicated code editors for good usability, it has the advantage of allowing merging processes to detect with precision renaming, moving, and referencing of code elements, and in turn, avoid both false positives and false negatives. Jaid takes into account syntactic and semantic aspects to apply a merge process based on transformations, having the key difference from previous approaches of relying on identity to locate and reference code elements. We performed an experiment with 100 merge scenarios from three open-source projects to test the technique and assess its feasibility in practice.

Keywords: Software merging, Version Control Systems, Transformations, Conflicts, Identity

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1. Context

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 dissertation, 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.

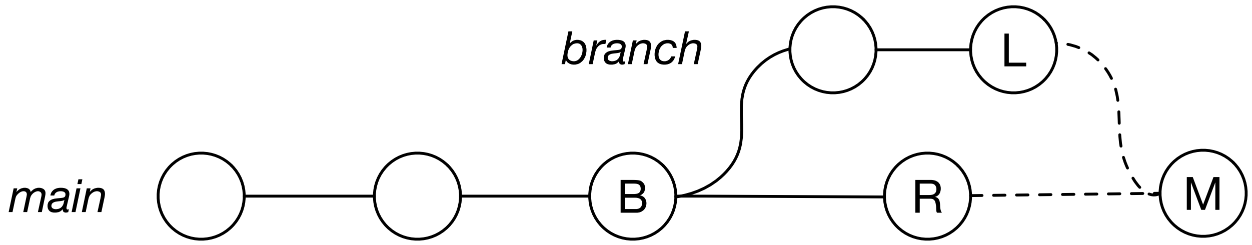


FIGURE 1. Three-way merge in Git-flow processes. left (L) and right (R) versions with a base ancestor (B) originate a merged version (M).

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 [1].

Most VCSs have built-in merge tools that are based on algorithms that compute the differences through a line-based comparison [2]. 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 be detected where there is, in fact, no conflict — these are *false positive* cases. Taking into account the drawbacks of unstructured merging, new studies have been performed to develop merge tools capable of taking the structure or grammar of the code into account [3, 4, 5, 6]. In this case, this merge strategy, because it takes into consideration the structure of the software artifact, is referred to as *structured*. Structured conflict resolution strategies also resolve conflicts where the syntax of the source code would be broken after merging.

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 [7]. The difficulty resides in the fact that source code elements do not have what will henceforth be called *identity*, and hence, a form of realizing *origin tracking* [8] 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 [9].

Both text-based and structured merging processes rely on matching the elements of the source code from different versions to discover which parts are similar and unmodified, and which new parts need to be considered during the merge process. The pieces of code that are considered the same by merge tools, and are therefore common to both revisions, are assumed to be unaltered from one version to the other. In contrast, any element of the source code that does not match another element is considered a new contribution, whether it is an addition or deletion, and must be checked for conflicts. Furthermore, it is also common to match elements that are not the same, but similar, to see if there have been small modifications to that particular element that are worth considering during the merge.

The multiple matching algorithms have their proper set of "rules" that are used to determine whether or not the similar elements actually refer to the same item. As a result, the matching algorithms adopted define the identity of code elements, which, when matched with the elements in other derived revisions, certifies that a particular element refers to the same element in an older revision, even though they are subtly distinct. The text-based approaches consider the equality of two lines to determine what is or is not changed, while structured approaches match source code elements based on their position in the whole structure, their relationships to other elements, and their own properties, typically their name. Matching is therefore an imprecise process, which in most cases, leads to different sets of matched elements depending on the algorithm used, and is also an error-prone process, due to the many concurrent changes to the same element in different revisions and the difficulty of perfectly matching them between derived revisions and their ancestor [10]. Thus, there is a need for an approach that is both unambiguous and objective in order to improve the matching process and, consequently, the merging process.

1.2. Goals

The main goal of this dissertation is to investigate if the automatic conflict resolution can be improved by using a more accurate merge process that relies on a form of representing the identity of the elements of the source code. In other words, to investigate whether the proposed technique reduces false positive and false negative conflicts, to obtain merged code that requires fewer manual corrections and also verify if the execution time of the merge process is reasonable and realistic according to the current software development practices. Furthermore, we aim to investigate if the proposed approach is able to maintain the advantages not only of the other available merge tools used by version control systems, which are mainly unstructured techniques, but also of those presented in related work (Chapter 2), which introduced the concept of structured and semistructured merging techniques. This dissertation addresses the following research questions:

RQ1. How to implement language-specific merge processes with accurate fine-grained detection of conflicts?

RQ2. What are the advantages and disadvantages of the process to be investigated in RQ1 compared to text-based (unstructured) merging processes and to state-of-the-art (semi) structured merging processes?

1.3. Methodology and results

The content of this dissertation is based on the following paper [11]:

Code Merging using Transformations and Member Identity

André R. Teles, André L. Santos

Onward! SPLASH'23, October 25–27, 2023, Cascais, Portugal

When compared to the paper, the dissertation describes more details regarding related work and prototype implementation, and the evaluation includes an additional project with merge scenarios.

This dissertation presents an approach that combines aspects of previous research on semistructured [6] and operation-based merging [12]. We perform version differencing and union using models [13] 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, this dissertation proposes 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 [14], 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[2], 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 (semi)structured merging techniques are not [15, 16].

We performed an early experiment with Jaid involving 100 merge scenarios extracted from three open-source projects. An analysis of false negatives on merges without conflicts revealed that 83% 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 on the order of a few seconds for the entire project merges. Although these are clearly slower than previous tools [15, 16], 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.

[1]https://www.jetbrains.com/mps/
[2]Publicly available at https://github.com/adrts-iscte/Jaid

1.4. Organization of the document

In this dissertation, we first present related work in Chapter 2 ending with Section 2.6, which motivates the need for more precise merging processes in light of the state-of-the-art merge tools. Chapter 3 describes the proposed merging process and Section 3.4 discusses details regarding the implementation of Jaid. Chapter 4 presents a first evaluation of our merging approach in real scenarios extracted from Github projects. Both Chapter 3 and Chapter 4 end with a section to discuss the contribution in terms of the related research question. Chapter 5 presents our conclusions and discusses the trade-offs of our approach.

CHAPTER 2

Related Work

The evolution of software development practices, in particular the parallelization of tasks, has created a demand for more accurate merging tools, while automatically resolving as many conflicts as possible and requiring less manual effort to merge all contributions into one solution, simply because of the need to increase productivity between development teams [17]. However, it is utopian to think that all merge conflicts can be resolved automatically. In such cases, manual intervention is required, or a set of conflict resolution policies must be defined to decide which revision to select and which to ignore [18]. All of these techniques have their pros and cons, so the question is: what is the merging technique that will guarantee better quality for less effort and time? [19].

2.1. Unstructured merging

Unstructured merging is a purely textual merging technique [20]. Line-based algorithms are the most common strategy for this type of merging [21, 2], 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 [2]. Examples of unstructured merge tools are the Concurrent Version System (CVS) [22] and the rcsmerge tool in the Revision Control System (RCS) [23]. 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. On the other hand, conflict detection is limited, mainly due to the granularity of line-based merging, as the algorithms do not distinguish between minor differences and major differences within the same line between different revisions, as both lines are considered different, even though one letter or the whole line has changed. 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) [15].

2.2. Structured merging

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 [3, 4, 5], with the aim of finding fewer unreal conflicts (false positives) and recognising more conflicts that are not detected by existing merge tools (false negatives), thus improving the accuracy of merging processes.

Structured merge tools, because they take the grammar of the source code into consideration, are more precise [6] 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. [6], 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.

Structured merge tools are classified according to the data structure used to parse the source code: those that use trees and those that use graphs as data structures.

OperV [24] and JDime [6] are examples of tree-based merging techniques. In addition, Asenov et al. [25] 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 [26] or ChangeDistiller [27] 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. In 2021, Castanho [28] proposed UNSETTLE, which also uses GumTree [26] as an AST matcher between different versions of a software artifact. This tool computes the changes between versions in a merge process and uses this information to automatically generate tests that are responsible for checking for semantic conflicts in the code that results from merging versions.

Mens [29] suggested graph rewriting as a way of representing software evolution. Pan et al. [30] proposed a program slice encoding algorithm based on the program dependence graph defined by Horwitz in [31], which computes textual and behavioral change information of functions.

2.3. Semistructured merging

Semistructured merge tools, as the name suggests, are not fully structured, but are a combination of unstructured and structured techniques. Semistructured 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 [32] and JDime [6] are examples of semistructured 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 [15, 16]. The FSTMerge approach was improved in jFSTMerge [15], reducing false positives and false negatives. IntelliMerge [16] 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 2.2, 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.

In 2020, a research was conducted that compared structured merging with semistructured merging. The results of this research [33] show that there is no significant difference between using structured or semistructured merging, but there are situations where structured merging is better and others where semistructured merging is better.

2.4. Model differencing

Our approach is performing model differencing [13] when extracting the transformations. Previous approaches and tools have addressed this problem using methods to differentiate models that conform to a given metamodel. DSMDiff [34] describes differencing algorithms for metamodels of the Generic Modeling Environment (GME [35]). The Eclipse Modeling Framework (EMF [36]), an implementation of the Meta-Object Facility OMG's standard [37], 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[3] provides facilities to compare models that conform to an Ecore metamodel. MPS also offers differencing capabilities for models described in the tool-specific meta-language. Despite the several existing methods for generic model differencing, in our approach, we opted to work with Java-specific models (JavaParser's AST) in order to perform fine-grained conflict detection (see Section 3.3.3). 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.

2.5. 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 [12] 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 [12], as well as the proposal and implementation of algorithms capable of segregating the conflicting transformations into sets.

Operation-based approaches are very useful for implementing an undo/redo mechanism, as is done in GINA [38]. In addition, many matrices of operations have been developed to find interferences between software evolution operations [39], with the aim of knowing the problems that may arise when both pairs of transformations are applied. In 2010, a study [40] was carried out on conflict detection strategies and the definition of conflict severity levels on operations. OperV [24] 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. In 2022, Ellis et al. proposed RefMerge [41], an operation-based refactoring-aware merging tool.

As pointed out by Lippe et al. [12], 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. Figure 2 illustrates this issue with the scenario of Figure 7. Note that a Rename Callable transformation, renames not only the callable in question, but also all of its

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

calls and a Change Callable's Body transformation, when applied, replaces the entire old body of the callable with the new extracted body. In the presented scenario in Figure 2, applying Change Callable's Body first and Rename Callable later results in the desirable merge solution, whereas the reverse order results in a compile error because the identifiers of both calls of *methodTo-BeRenamed* have not been renamed to *methodRenamed* and *methodToBeRenamed* is no longer defined.

2.6. Motivating Example

This section presents an example to motivate our approach in contrast to industrial practice and state-of-the-art approaches. Figure 3 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).

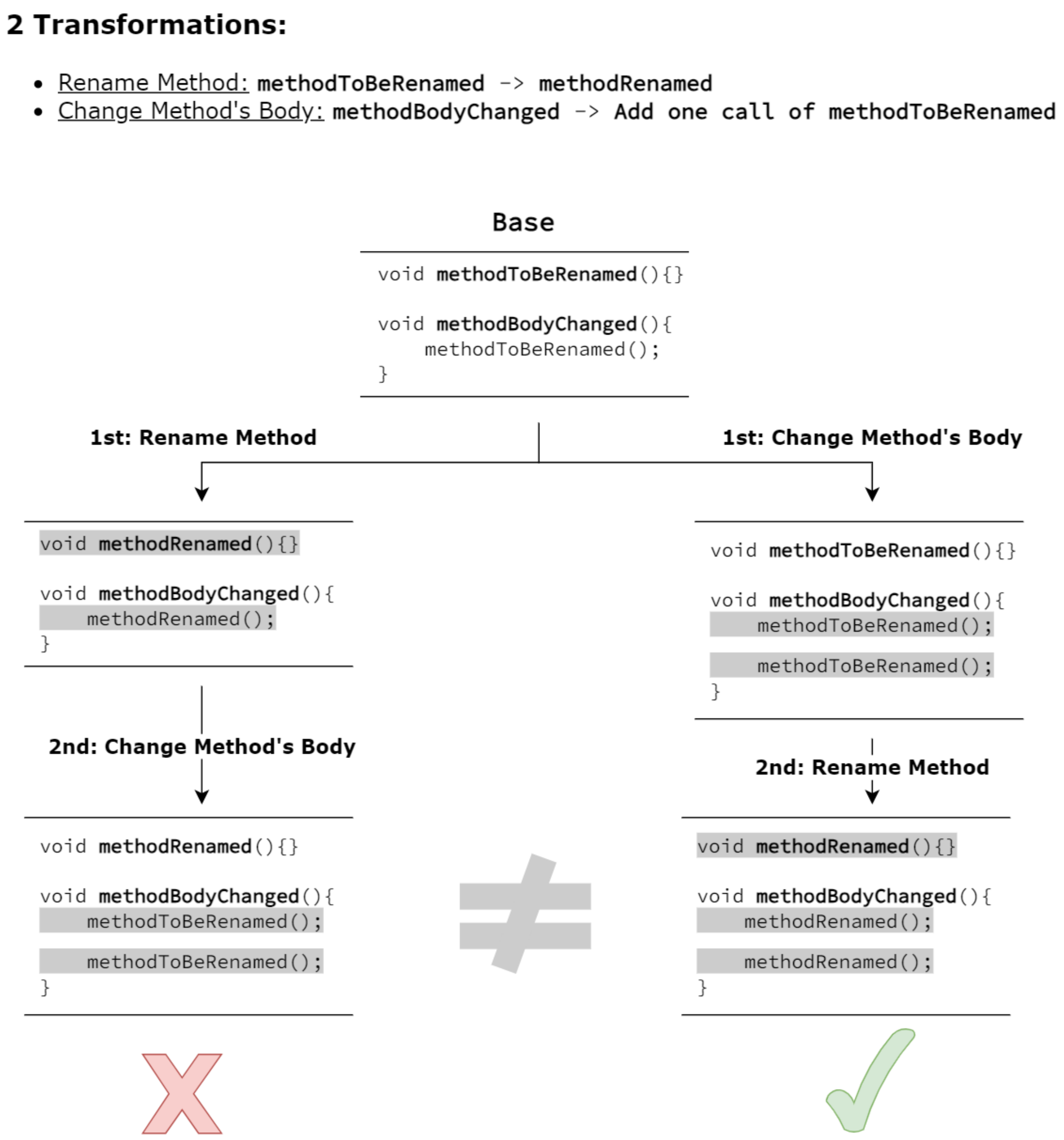
Merging the left and right versions into a valid new version of the class is possible, as the set of changes are not incompatible. Figure 3 (d) presents the ideal merge result, where both field modifications are integrated, method renames are performed, and references to it are updated accordingly. However, 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 3 (a,b,c) with git-merge[4], a widely-used industrial form of merging, jFSTMerge [15][5], IntelliMerge [16][6], which are state-of-the-art approaches to this problem, and MPS, an industrial language workbench 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 4. 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

[4]https://git-scm.com/docs/git-merge
[5]https://github.com/guilhermejccavalcanti/jFSTMerge
[6]https://github.com/Symbolk/IntelliMerge

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


2 Transformations:

- Rename Method: `methodToBeRenamed -> methodRenamed`
- Change Method's Body: `methodBodyChanged -> Add one call of methodToBeRenamed`

Base

```
void methodToBeRenamed(){}

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

1st: Rename Method

```
void methodRenamed(){}

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

2nd: Change Method's Body

```
void methodRenamed(){}

void methodBodyChanged(){
    methodToBeRenamed();

    methodToBeRenamed();
}
```

1st: Change Method's Body

```
void methodToBeRenamed(){}

void methodBodyChanged(){
    methodToBeRenamed();

    methodToBeRenamed();
}
```

2nd: Rename Method

```
void methodRenamed(){}

void methodBodyChanged(){
    methodRenamed();

    methodRenamed();
}
```

FIGURE 2. Transformation ordering in operation-based merge.

```
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 3. Illustrative three-way merge scenario involving a single class.

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 [9].

Finally, we tested the merging scenario in MPS, which carries out the merge using the model representation of the code (in Figure 4 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).

```
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 4. Merge results with industry and state-of-the-art approaches.

CHAPTER 3

Approach

3.1. Representing member identity

This dissertation proposes 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 [42]) 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.

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 5 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 Chapter 5.

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

```
//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 5. Java source code with member identity encoded as UUIDs in comments.

3.2. Transformations

The fundamental part of an operation-based approach are operations [12], 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 [43]) 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.

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

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 6 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 3.4.2). 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 3.3.3). 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 3.3.4).

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.

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 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

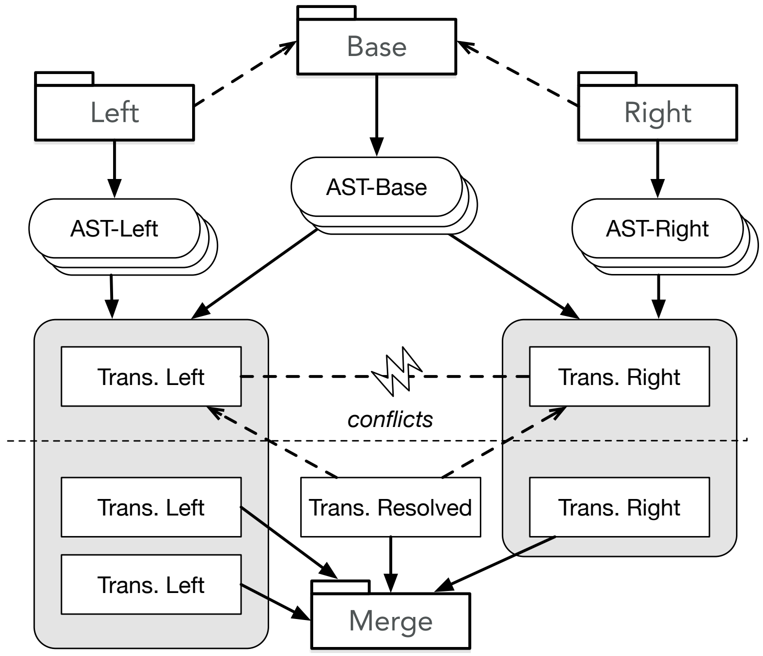


FIGURE 6. Merge process overview.

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
```

result in a transformation, whose type is specific to the kind of modification (e.g., Rename Class, Change Method's Body). 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 3 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 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

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
    AddLocalMoveTransformations($transformations$, $baseChildren$, $versionChildren$)
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 $(a, r) \in \{(a, r) \in (additions \times removals) : a.uuid = r.uuid \wedge a.node.isStatic\}$ do
        $transformations \leftarrow transformations \cup \{MoveNode(a, r)\}$
        $transformations \leftarrow transformations - \{a, r\}$
        AddNodeTransformations($transformations$, $r.node$, $a.node$)
    end for
end procedure
```

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. Shared transformations

When two transformations in the two versions (left and right) 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.

Consider two new classes *X* and *Y*, added in the left and right branches, respectively. These modifications to the base version would generate two Add Class transformations. Both *X* and *Y* are exactly the same, except for their UUIDs and their child nodes (fields, methods, etc.), which, by being committed in two different versions (left and right), have been injected with different UUIDs and thus different identities. Since both are equivalent, and the comparison of transformations ignores the UUIDs of the nodes to prevent the cases of new node insertions in the project, both transformations are redundant and passible to be included in the shared set. Although they are equivalent, we cannot simply apply one and ignore the other because their identities are different. This is particularly important because with different changes between the left and right versions, new references can be added that refer to their constructs. Suppose only the transformation that inserts class *X* is applied, and the other one about *Y* is ignored (as they are redundant), and a new reference to *Y* is added in the right branch. When translating the identifier of the reference (see Section 3.3.5), since the reference was added in the right branch, it will refer to the UUID of class *Y*. However, in the merged version, only the UUID of *X* exists, since its Add Class transformation was the only one applied. To prevent these cases, an intermediate step is performed to homogenize the identity of all members of two redundant transformations, the main one and all its child nodes. In this intermediate step, all UUIDs of *Y* and its children are replaced by those of *X* in case the transformation of class *X* is applied. Otherwise, all UUIDs of *X* and its children are replaced by those of *Y* if the transformation of class *Y* is applied.

3.3.3. 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. Tables 2–4 present 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 cause of the conflict.

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 (Tables 2–4).

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 4.

TABLE 2. Transformation pairs with potential conflicts on file transformations.

Type	**Package**	**Imports**	**Add**	**Remove**	**Rename**	**Modifiers**	**Implements**	**Extends**	**Move**
Move			●	●	●				●
Extends				●				●	
Implements				●			●		
Modifiers				●		●			
Rename			●	●	●				
Remove									
Add			●						
Imports		●							
Package	●								

3.3.4. Applying transformations

If there are no conflicting transformations, all the transformations of the three sets (left, right and shared) can be safely merged into a copy of the base version in order to obtain a newly merged

TABLE 3. Transformation pairs with potential conflicts on method transformations.

Method	**Add**	**Remove**	**Signature**	**Body**	**Modifiers**	**Return Type**	**Move**
Add	●		●				●
Remove			●	●	●	●	●
Signature			●				●
Body				●			
Modifiers					●		
ReturnType						●	
Move							●

TABLE 4. Transformation pairs with potential conflicts on field transformations.

Field	**Add**	**Remove**	**Rename**	**Type**	**Modifiers**	**Initializer**	**Move**
Move	●	●	●				●
Initializer		●		●		●	
Modifiers		●			●		
Type		●		●			
Rename	●	●	●				
Remove							
Add	●						

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
```

version. In the presence of conflicts, This dissertation suggests 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 (see Section 3.4.3). 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 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 3.4.2) 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.

3.3.5. Translation of identifiers by reference

Most transformations are straightforward to apply, with the exception of transformations that may introduce new references, which 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 4). We address this problem so that those “outdated” references are translated into the correct ones.

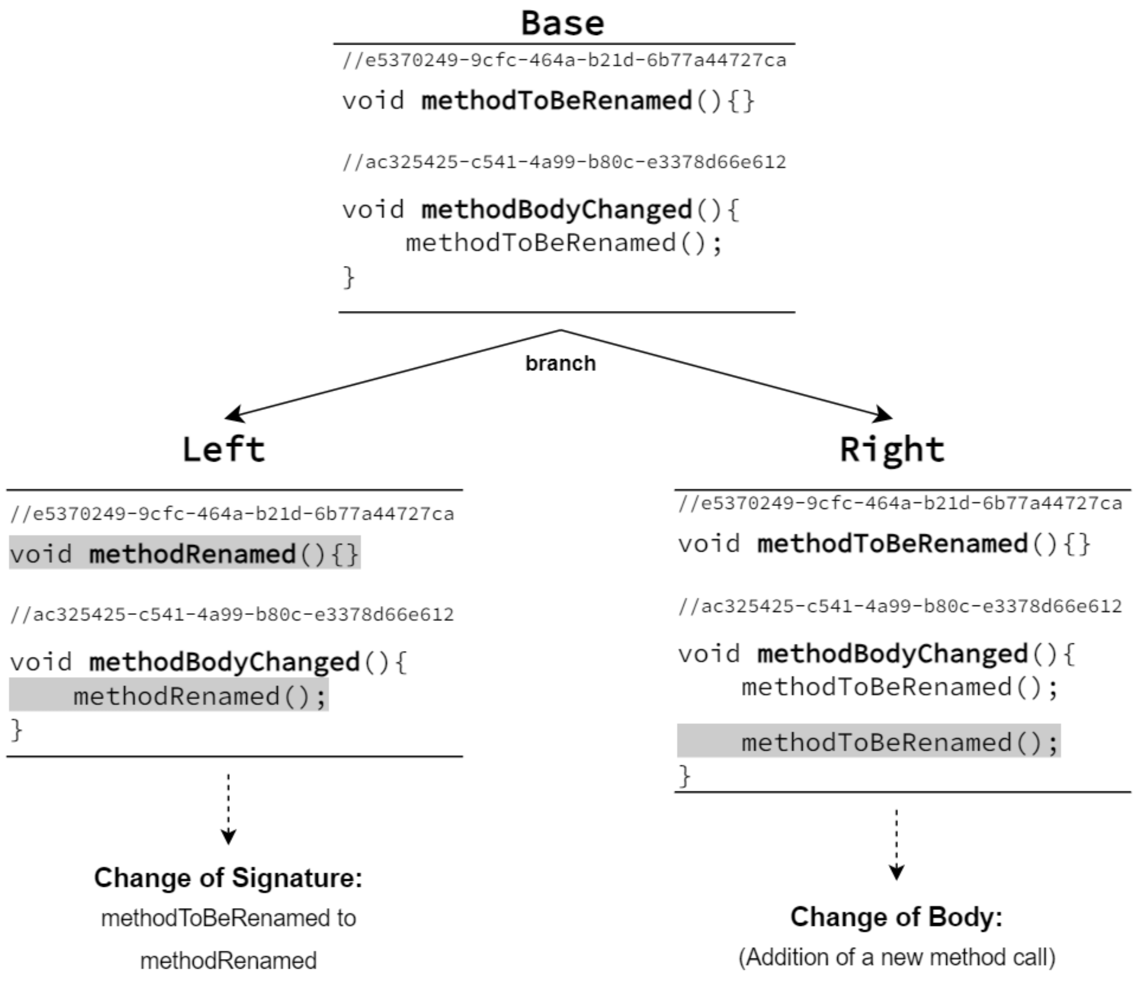
Figure 7 shows a three-way merge scenario to illustrate this case in which two branches are derived from a base branch: the left branch, in which the method named *methodToBeRenamed* is renamed to *methodRenamed* along with all its calls, resulting in a Change of Signature transformation; and the right branch, in which a new method call to *methodToBeRenamed* is added to the method named *methodBodyChanged*, resulting in a Change of Body 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. Figure 2 illustrates that if we perform right followed by left, we reach the desired output. However, 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

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 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 8.

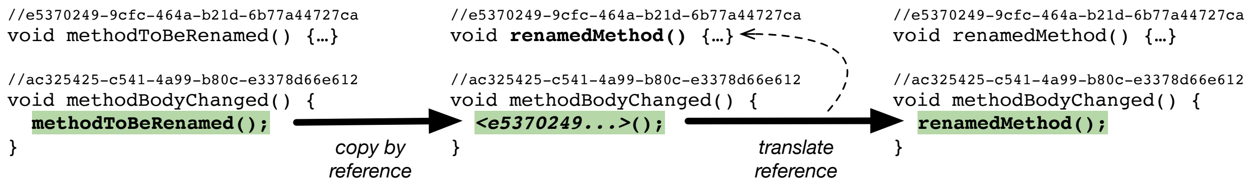

Base
branch
Left
Right
Change of Signature:
methodToBeRenamed to methodRenamed
Change of Body:
(Addition of a new method call)

```
//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 7. Three-way merge scenario.



```
//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 8. Translating identifiers by reference in body modify transformations.

The order in which transformations are applied is a limitation of operation-based merging as showed in Figure 2. Figure 9 shows how the proposed approach overcomes this limitation of operation-based merging, allowing the transformations to be applied in any order and the output to be always correct. The transformations are able to be applied regardless of order, due to the translation of identifiers by reference step should always be applied after transformations that may introduce new references to an element. Examples of these transformations include inserting new elements such as callables or fields, changing the parameters of a callable that may introduce a parameter of class type, or making a single change to the body of a callable, as shown in the scenario. Along the left path, where the first transformation to be applied is the Rename Callable, the method called *methodToBeRenamed* and its call are both renamed. Then, by applying the Change Callable's Body transformation, the entire body of the method *methodBodyChanged* is replaced with the body of the method with the corresponding UUID in the branch shown in Figure 7. This modification introduces two broken references to the method *methodToBeRenamed*, since it has been renamed and its signature no longer matches its calls. These references are fixed by obtaining the UUID of the element they were referencing in the branch and copying its current identifier into the merged version. For this reason, our approach requires an unambiguous way of assigning identity to elements in order for the translation of identifiers by reference step to work properly, and for the entire merge process to be error-free. On the right path, as illustrated in Figure 2, the order in which the transformations are applied does not create any inconsistency, however the translation of identifiers step is always performed after a Change Callable's Body transformation, even though nothing has changed in the final merged version.

3.4. 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. 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.

3.4.1. Abstract Syntax Trees for Java

Abstract syntax trees are data structures commonly used in compilers to represent the structure of the program. It is called a syntax tree because the underlying data structure is a tree of the syntactic structure of the source code. It is also called abstract because not all the syntactic details shown in the textual representation of the source code are transferred to the tree, only the context-sensitive details. For example, parentheses and semicolons are both implicit in ASTs. Considering an AST-to-Code process of Java, all statements should end in a semicolon, even though they are not shown in the parsed AST, as well as the parentheses of an if-condition.

A critical library that was fundamental for developing Jaid was JavaParser[7], 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., [16]). 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. 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 3.3.3), and transformations are applied through AST manipulations to obtain the final merged version (Sections 3.3.4 and 3.3.5).

[7]https://javaparser.org

3.4.2. 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 Chapter 4), because, as opposed to other merging techniques (e.g., [15, 16]), 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 4).

3.4.3. Local move transformations

In Jaid, a local move transformation is a type of transformation that is responsible for repositioning an element within its parent. Types, methods, constructors, fields, and enum constants are all the elements that can be moved in Jaid.

A local move transformation stores information about:

(1) The UUID of the element to be moved;
(2) The index of the parent member list to which the element will be repositioned;
(3) The ordinal index of the order in which a particular transformation is applied within the same parent, as explained below;

Figure 10 presents the application of the step-by-step algorithm implemented in Jaid to extract local move transformations for a given scenario.The Algorithm 7 describes the process of extracting local move transformations based on two lists of child nodes in different versions (base and version).

Algorithm 7 Procedure to obtain all local move transformations between two child lists (base and version).

```
procedure AddLocalMoveTransformations($transformations$, $baseChildren$, $versionChildren$)
    $commonBase \leftarrow baseChildren - (baseChildren - versionChildren)$
    $commonVersion \leftarrow versionChildren - (versionChildren - baseChildren)$
    if $commonBase \neq commonVersion$ then
        $mapBasePositions \leftarrow$AssociateBaseToVersionIndexes($commonBase$, $commonVersion$)
        $lisOfIndexes \leftarrow$LongestIncreasingSubsequence($mapBasePositions.values()$)
        $mapIndexToBaseElements \leftarrow listOfIndexes.map(ind \rightarrow commonBase[ind])$
        $i \leftarrow commonBase.size()$
        $orderIndex \leftarrow 0$
        while $i \neq 0$ do
            $i \leftarrow i - 1$
            $elem \leftarrow commonVersion[i]$
            if $elem \notin mapIndexToBaseElements$ then
                $transformations \leftarrow transformations \cup \{LocalMove(elem.uuid, i, orderIndex)\}$
                $orderIndex \leftarrow orderIndex + 1$
            end if
        end while
    end if
end procedure

function AssociateBaseToVersionIndexes($commonBase$, $commonVersion$)
    $mapBasePositions \leftarrow \emptyset$
    for all $baseMember \in commonBase$ do
        $commonVersionIndex \leftarrow commonVersion.indexOf(baseMember)$
        $mapBasePositions \leftarrow mapBasePositions \cup (baseMember : commonVersionIndex)$
    end for
    return $mapBasePositions$
end function
```

The extraction of local move transformations is checked on an element with children (Algorithm 3) to extract the minimum number of atomic child local moves. In Jaid, this is achieved by obtaining the *Longest Increasing Subsequence* algorithm to the two lists of child nodes for comparison, base and branch. Note that applying the *LIS* algorithm requires two lists with the same elements, which may not always be true between the two lists of child nodes. To achieve this, the lists to check for local moves are the lists of child nodes without the children to be removed and the children to be inserted, in other words, with only those members with UUIDs that are common to both lists. For this reason, at the time the local move transformations are applied, all the removal transformations must already be applied, and the insertion transformations must be applied after the local move transformations are applied, as mentioned in Section 3.3.4, so that all applications of transformations result in a reliable merge.

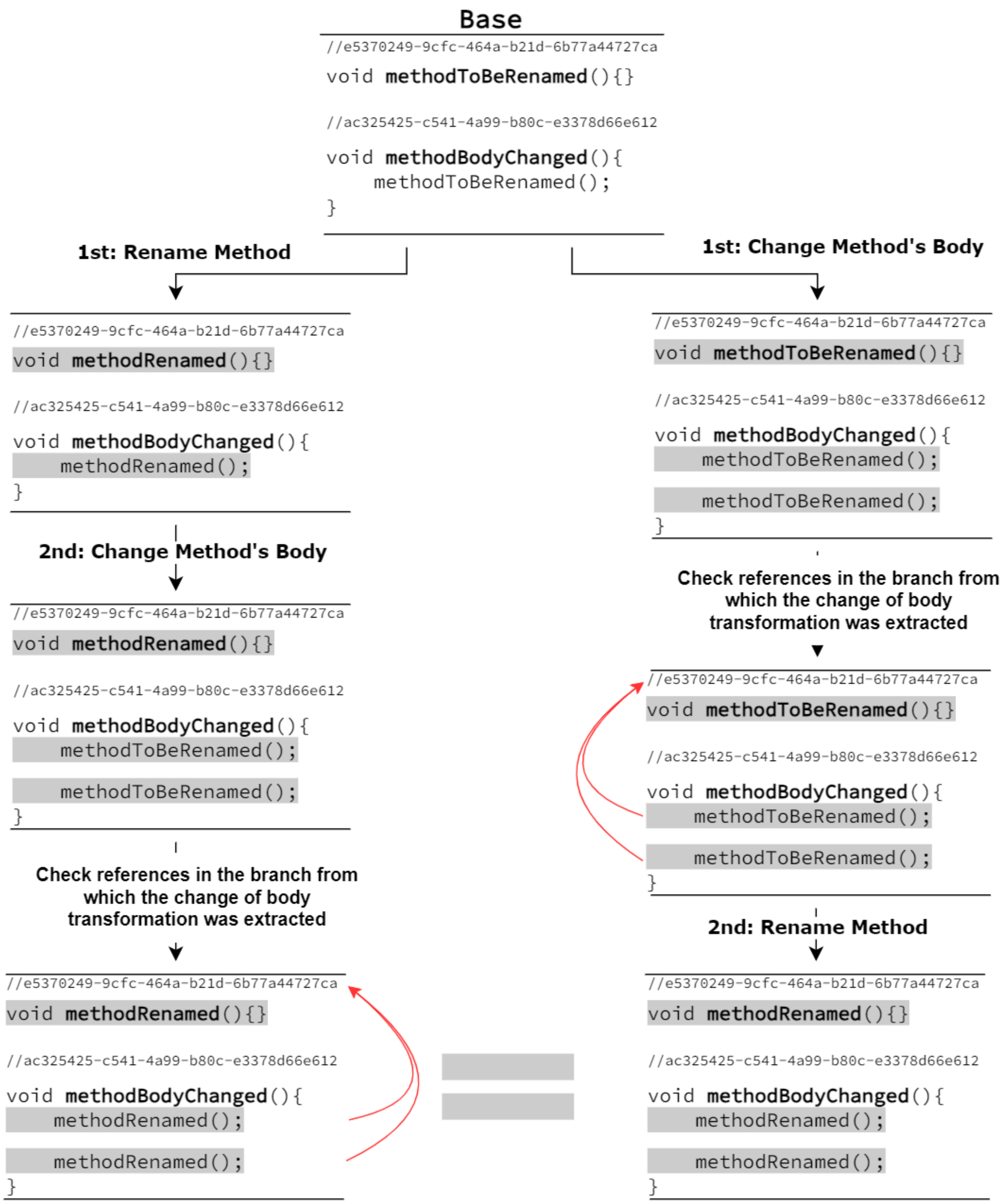
3.5. Discussion

There is a trade-off between prototyping a tool that is language-independent in contrast to a language-specific one. A merge tool that can be generalized to any programming language clearly cannot accommodate all the syntactic and semantic nuances of each language, and thus cannot detect all the conflicts of the language's code in detail, nor can it prevent new false negative conflicts from occurring. On the other hand, the development of a language-specific merge tool, if implemented meticulously with attention to detail, can detect almost any conflict because it knows the grammar and semantic aspects of the language in question. In this sense, we opted to develop a merge tool with an emphasis on detailed conflict detection, and set aside the goal of generalizable, language-independent implementation, since the main goal is to improve the accuracy of the merging process, with special attention to reducing the number of false positives and negatives. Next, we had to figure out how best to represent the source code in a well-structured way, and in particular, the code changes, whose representation must have a certain level of detail that would facilitate the conflict detection phase. Therefore, we found that an operation-based merge approach was a suitable option, since representing changes to software artifacts through transformations is a deterministic way to control the evolution of software constructs on a state-wise basis. The other benefit of using transformations to express software modifications is that, by being handled using a well-defined format, their information about a change to a software construct can be easily retrieved and easily compared to facilitate the conflict detection phase.

We aimed at improving the accuracy of merging processes by reducing the number of false negatives and positives while correctly finding the true negatives and positives. Even though a lot of research has been done on different merging techniques, as discussed in Chapter 2, there are some shortcomings related to identifying which source code elements match between different versions of the same software artifact. For this purpose, an innovative way of assigning identity to elements was considered and resulted in augmenting the comments of the elements with UUIDs, which are assumed to be correctly maintained throughout the evolution of the software artifact. Recalling the first research question (RQ1):

- How to implement language-specific merge processes with accurate fine-grained detection of conflicts?

The development of Jaid reflected the idea of implementing a language-specific merging tool with fine-grained conflict detection using transformations, and also taking advantage of member identity to make it more precise.


Base
1st: Rename Method
1st: Change Method's Body
2nd: Change Method's Body
Check references in the branch from which the change of body transformation was extracted
Check references in the branch from which the change of body transformation was extracted
2nd: Rename Method

```
//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();
}
```

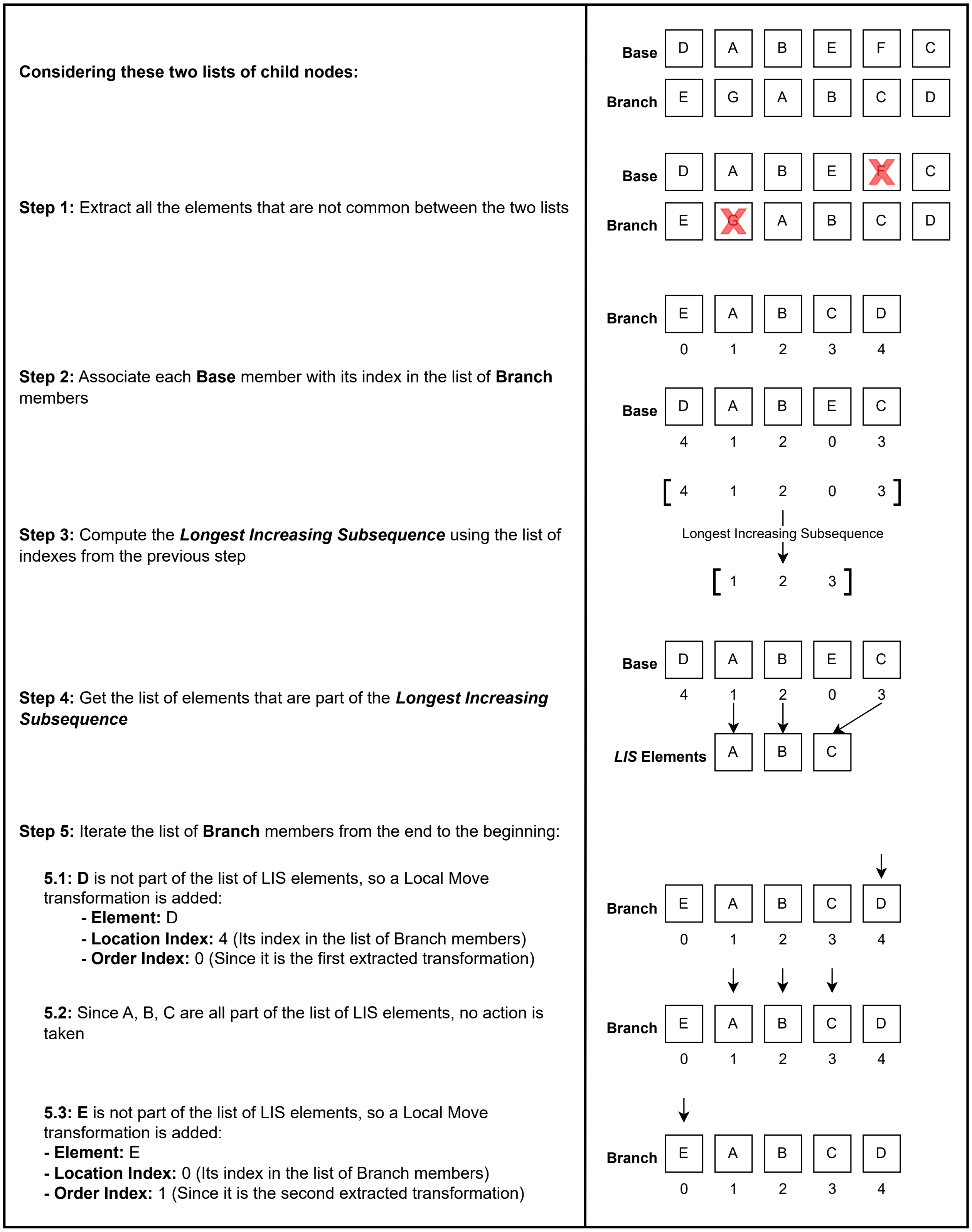
```
//e5370249-9cfc-464a-b21d-6b77a44727ca
void methodRenamed(){}

//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();
}
```

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

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

    methodRenamed();
}
```

FIGURE 9. Translate identifiers by reference.



Considering these two lists of child nodes:

Step 1: Extract all the elements that are not common between the two lists

Step 2: Associate each **Base** member with its index in the list of **Branch** members

Step 3: Compute the ***Longest Increasing Subsequence*** using the list of indexes from the previous step

Step 4: Get the list of elements that are part of the ***Longest Increasing Subsequence***

Step 5: Iterate the list of **Branch** members from the end to the beginning:

5.1: D is not part of the list of LIS elements, so a Local Move transformation is added:
- **Element:** D
- **Location Index:** 4 (Its index in the list of Branch members)
- **Order Index:** 0 (Since it is the first extracted transformation)

5.2: Since A, B, C are all part of the list of LIS elements, no action is taken

5.3: E is not part of the list of LIS elements, so a Local Move transformation is added:
- **Element:** E
- **Location Index:** 0 (Its index in the list of Branch members)
- **Order Index:** 1 (Since it is the second extracted transformation)

FIGURE 10. The step-by-step algorithm for extracting local move transformations for a given scenario.

CHAPTER 4

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.

In answer to RQ2, we examine some metrics, such as false negatives and positives, of the results obtained to investigate whether or not the proposed approach actually reduces integration effort without negatively impacting the correctness of the merge process, resulting in increased productivity and quality.

4.1. Collecting merge scenarios

We extracted three sets of real merge scenarios from the supplementary material of the paper by Cavalcanti et al. [15]. 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 merge scenarios from the projects' Git repositories. We selected three out of the four projects available in the small sample contained in the package, Bukkit[8], jsoup[9] and clojure[10].

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

[8]https://github.com/Bukkit/Bukkit/
[9]https://github.com/jhy/jsoup
[10]https://github.com/clojure/clojure

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 [16, 20] as a tool to calculate differences in ASTs, but in our case to give identity to their elements. We used GumTree [26] for this purpose, one of the state-of-the-art tools, with a tree generator based on JavaParser's ASTs. In the case of elements that have no other matching elements, they have a random UUID attached, which reflects on a removal if the element is from the base branch, or an insertion if the element is from a version branch (left or right).

After finding all the mappings resulting from the GumTree tool, correspondence was made between a GumTree node 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 100 merge scenarios, 19 from Bukkit, 44 from jsoup and 37 from clojure.

4.2. 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 5 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 two of their transformation sets (left or right or shared set) are empty due to changes over members not being covered by Jaid (see Section 3.4).

TABLE 5. 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%)
clojure	37	4 (11%)	29 (78%)	4 (11%)
Total	**100**	**10 (10%)**	**72 (72%)**	**18 (18%)**

4.2.1. 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 project (Bukkit, jsoup or clojure) of the merge scenario, we investigated if these files would build successfully, and if not, what was the reason.

We found 12 (17%) scenarios in which errors were not caused by shortcomings in the proposed approach or a malfunction in its subsequent implementation in Jaid, but instead were all related to Java's missing coverage of Jaid, and therefore no false negatives were found. The most frequent case (7 out of 12 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 five scenarios failed to build successfully due to the absence of transformations regarding converting a class to an interface, exception classes changes in the throws keyword used along with the method signature, and the lack of coverage of information related to static blocks and annotation declarations. The remaining 60 (83%) merge scenarios were built successfully.

Table 6 presents the number of merge scenarios per project that failed to build successfully due to missing Java coverage. Note that Table 6 gives a total of 13 reasons for 12 merge scenarios, as one of the merge scenarios of jsoup was not successfully built because of the lack of Java coverage of two different structural elements: static block and class body of enum constant.

TABLE 6. Summary of the number of merge scenarios per project that failed to build successfully due to missing Java coverage.

Missing Coverage	Bukkit	jsoup	clojure
Transform Class in Interface	1	0	0
Method throws exception	1	0	0
Class Bodies of Enum Constants	0	7	0
Static Block	0	1	1
Annotation Declaration	0	0	2

4.2.2. 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 51 conflicts were found, but only 8 different types of conflict, a small fragment of the conflictual pairs. Table 7 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.

TABLE 7. 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.
13	RemoveCallable–*	The removed callable has changes in it.
1	AddCallable–Signature	The new added callable has the same signature as the changed one.
1	AddCallable–AddCallable	The two newly added callables are different, but have the same signature.
7	RemoveField–Body	The new body has references to the removed field.

4.2.3. 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%, 70% and 65% of the time for Bukkit, jsoup and clojure, respectively. The remaining time is consumed by extracting transformations (45% for Bukkit, 28% for jsoup and 32% for clojure), 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, since the tasks of resolving references and extracting transformations between different structural elements are all independent. The overall execution times of the merging process took on average 2.9, 13.7 and 54.7 seconds for Bukkit, jsoup, and clojure projects, respectively, which we consider acceptable execution times to make the proposed approach a viable option in practice. The total number of references of the three versions (base, left and right) of a merge scenario were on average 4854, 36474 and 78621 for Bukkit, jsoup, and clojure projects, respectively.

Additionally, since the parsing/indexing phase is the most time-consuming task of the whole process, we explored whether the total number of references to resolve of a version (base + left + right) and the execution time of the whole process are directionally proportional to each other. We computed the Pearson correlation between these two variables and obtained the following results: 0.96 for Bukkit, 0.26 for jsoup and 0.97 for clojure. Figure 11 presents the relationship between the total number of references and the merge process execution time for each project, where it is visible that both Bukkit and clojure project results reflect a linear regression, whereas jsoup does not.

4.2.4. Threats to validity

The fact of running an experiment with only two projects, even though 100 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.

4.3. Discussion

The answer to the second research question (RQ2):

- What are the advantages and disadvantages of the process to be investigated in RQ1 compared to text-based (unstructured) merging processes and to state-of-the-art (semi) structured merging processes?

is based on the results obtained from the experiment, with emphasis on the manually obtained false negatives and positives metrics, along with the total execution time of the merge process.

Note that some structural elements are not covered by Jaid, so the entire code of a version of a merge scenario may not be evaluated because the missing covered elements are ignored. In some merge scenarios, the results of the experiment may be biased because the majority of changed and conflicting elements may not be covered software constructs, such as class bodies of enum constants or static blocks. With exception to these cases, where build issues were detected due to missing coverage of Java, not a single false negative was found regarding elements covered by Jaid, in other words, not a single compilation error was introduced by applying all the extracted transformations, which allows us to state with confidence that if the application process of each different type of transformation is implemented correctly, it will never output a non-compilable merge result.

Moreover, all conflicts found are in fact between two conflicting transformations and require human intervention in order to resolve them, since Jaid's conflict detection phase is implemented on the basis of conflicting pairs that cannot be resolved automatically (recall Section 3.3.3). Since no false positives were found, these results demonstrate that the idea behind developing a database of handlers for potential conflicting pairs, along with the proposed way of giving unambiguous identity to the source code elements, is a modularized, scalable, and precise way of conflict detection.

On the other hand, the overall execution time is slower when compared to the VCS practices or other state-of-the-art tools. The duration of the merge process increases as the number of references to be resolved in a merge scenario increases. However, the time a developer would take to manually identify the conflict and further resolve would likely take longer when compared to using a tool that clearly presents conflicts with a message and specifies which are the conflicting elements of the code.

Our false negative and positive conflict results were not compared with the state-of-the-art semistructured merge tools jFSTMerge [15] or IntelliMerge [16], because tree/graph-based matching is the way these merge tools provide identity, which is similar to our fabricated merge scenarios. Preserving the identity of structural source code elements throughout the development cycle is a property that is fundamental and cannot be simulated, since providing it manually or using some matching tools or algorithms creates imprecisions in the matching phase of elements between merge scenario versions. For this reason, we consider the lack of real merge scenarios with member identity a plausible justification for not comparing Jaid with the other state-of-the-art merge tools.

Another aspect that compromises the comparison between ours and other merge tools is the granularity of the conflicts. By granularity, we understand the level of detail a single conflict can encompass. Figure 12 shows that the same merge scenario produces three different outputs and three different numbers of conflicts when merged with three different merge tools. jFSTMerge's output considers the entire method block as conflicting, while IntelliMerge, by finding a common brace between the two methods, makes a distinction between the modifier/return type and the body, producing two different conflicts. In our approach, since a transformation is generated for each element composing a method declaration that is changed, transformations are generated for the changes in the modifiers, return type and body, for each branch (left and right). Since the changes between the two branches cannot all be combined, there is a conflict between each pair of transformations, resulting in three conflicts.

However, it is not clear that our approach will always have more conflicts. Other approaches use a finer granularity when searching for differences within bodies, based on a per-statement comparison, whereas in our approach, we handle the body as a whole block of statements, as explained in Section 3.3.1.

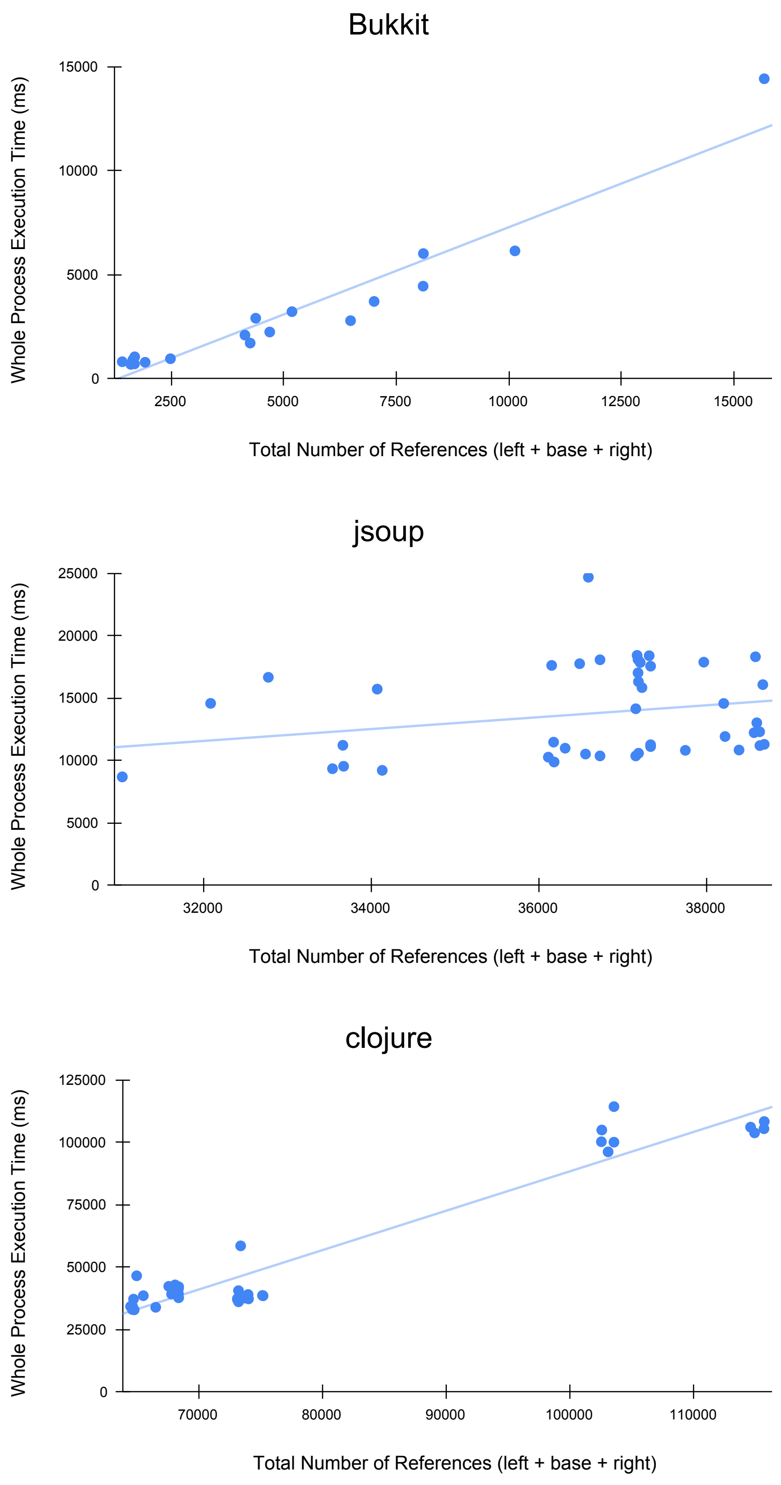
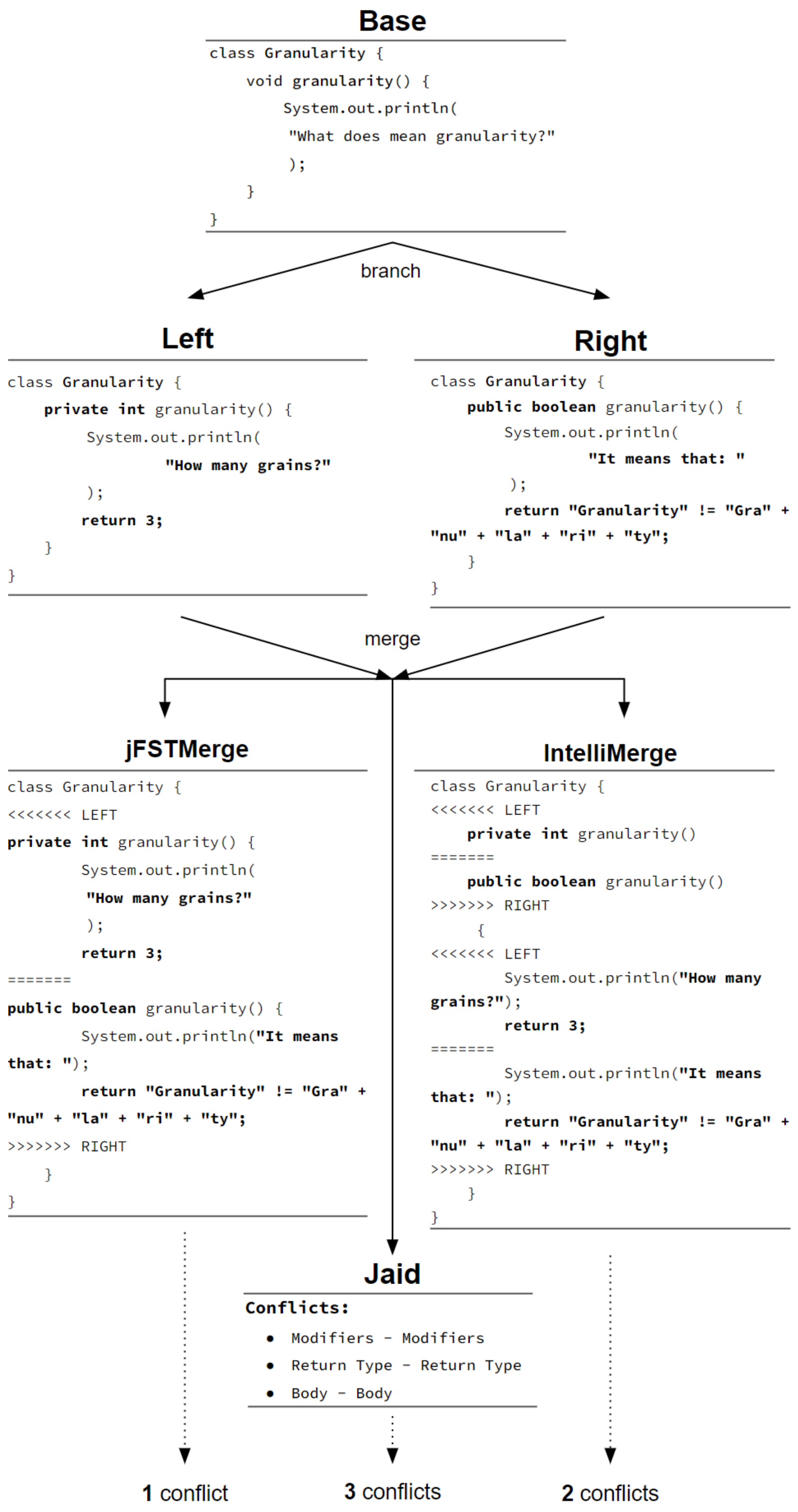


FIGURE 11. Merge times in relation to the total number of references.



Base

```
class Granularity {
    void granularity() {
        System.out.println(
         "What does mean granularity?"
         );
    }
}
```

Left

```
class Granularity {
    private int granularity() {
         System.out.println(
                  "How many grains?"
         );
        return 3;
    }
}
```

Right

```
class Granularity {
    public boolean granularity() {
        System.out.println(
                "It means that: "
         );
        return "Granularity" != "Gra" +
"nu" + "la" + "ri" + "ty";
    }
}
```

jFSTMerge

```
class Granularity {
<<<<<<< LEFT
private int granularity() {
        System.out.println(
         "How many grains?"
         );
        return 3;
=======
public boolean granularity() {
        System.out.println("It means
that: ");
        return "Granularity" != "Gra" +
"nu" + "la" + "ri" + "ty";
>>>>>>> RIGHT
    }
}
```

IntelliMerge

```
class Granularity {
<<<<<<< LEFT
    private int granularity()
=======
    public boolean granularity()
>>>>>>> RIGHT
     {
<<<<<<< LEFT
        System.out.println("How many
grains?");
        return 3;
=======
        System.out.println("It means
that: ");
        return "Granularity" != "Gra" +
"nu" + "la" + "ri" + "ty";
>>>>>>> RIGHT
    }
}
```

Jaid

```
Conflicts:
  •  Modifiers - Modifiers
  •  Return Type - Return Type
  •  Body - Body
```

FIGURE 12. Different levels of conflict granularity per merging tools.

CHAPTER 5

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.

To answer the first research question, the implementation of Jaid using the proposed approach proved to have the potential to be a state-of-the-art merge tool, although it only supports Java and would require a long effort to extend to other languages. 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. The execution of the experiment validated the realization of the approach and provided some evidence that it is feasible in practice.

Regarding the second research question, we have shown how more cases of false negatives and false positives can be successfully addressed, most notably, with respect to changes involving renamings. On the other hand, the results showed that the overall merge execution time is significantly slower compared to other merge tools, as it scales with the growing number of source code references, which can be a problem in larger projects.

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.

5.1. 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 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 5). 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., [44, 45, 46, 47]). 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 [14], MPS[11]). 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

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

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).

5.2. 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.

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 Chapter 1).

5.3. Future Work

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.

Additionally, 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.

It would also be interesting to explore the same approach in a graph-based implementation to study the influence of identity on semantic conflict detection, since graph-based tools are by their nature better suited to represent the behavioral concepts of software artifacts than tree-based ones.

References

[1] C. Brindescu, I. Ahmed, C. Jensen, and A. Sarma, “An empirical investigation into merge conflicts and their effect on software quality,” *Empirical Softw. Engg.*, vol. 25, no. 1, pp. 562–590, jan 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09735-4

[2] J. W. Hunt and M. D. Mcilroy, “An algorithm for differential file comparison,” *Computer Science*, 1975. [Online]. Available: http://www.cs.dartmouth.edu/%7Edoug/diff.pdf

[3] J. Buffenbarger, “Syntactic software merging,” in *Software Configuration Management*, J. Estublier, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 153–172.

[4] B. Westfechtel, “Structure-oriented merging of revisions of software documents,” in *Proceedings of the 3rd International Workshop on Software Configuration Management*, ser. SCM ’91. New York, NY, USA: Association for Computing Machinery, 1991, pp. 68–79. [Online]. Available: https://doi.org/10.1145/111062.111071

[5] T. Apiwattanapong, A. Orso, and M. J. Harrold, “Jdiff: A differencing technique and tool for object-oriented programs,” *Automated Software Eng.*, vol. 14, no. 1, pp. 3–36, mar 2007. [Online]. Available: https://doi.org/10.1007/s10515-006-0002-0

[6] S. Apel, O. Leßenich, and C. Lengauer, “Structured merge with auto-tuning: balancing precision and performance,” in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 120–129.

[7] R. van Rozen and T. van der Storm, “Origin tracking + text differencing = textual model differencing,” in *Theory and Practice of Model Transformations*, D. Kolovos and M. Wimmer, Eds. Cham: Springer International Publishing, 2015, pp. 18–33.

[8] A. van Deursen, P. Klint, and F. Tip, “Origin tracking,” *J. Symb. Comput.*, vol. 15, no. 5–6, pp. 523–545, may 1993. [Online]. Available: https://doi.org/10.1016/S0747-7171(06)80004-0

[9] L. Silva, P. Borba, and A. Pires, “Build conflicts in the wild,” *Journal of Software-Evolution and Process*, p. (also appeared in ICSME’2022 Journal First track), 2022.

[10] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, “Renaming and shifted code in structured merging: Looking ahead for precision and performance,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’17. IEEE Press, 2017, p. 543553.

[11] A. R. Teles and A. L. Santos, “Code merging using transformations and member identity,” in *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 23) - October 25 - 27, 2023*, 2023. [Online]. Available: https://doi.org/10.1145/3622758.3622891

[12] E. Lippe and N. van Oosterom, “Operation-based merging,” *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 5, pp. 78–87, nov 1992. [Online]. Available: https://doi.org/10.1145/142882.143753

[13] M. Alanen and I. Porres, “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*, ser. Lecture Notes in Computer Science, P. Stevens, J. Whittle, and G. Booch, Eds., vol. 2863. Springer, 2003, pp. 2–17. [Online]. Available: https://doi.org/10.1007/978-3-540-45221-8_2

[14] C. Simonyi, M. Christerson, and S. Clifford, “Intentional software,” *SIGPLAN Not.*, vol. 41, no. 10, pp. 451–464, oct 2006. [Online]. Available: https://doi.org/10.1145/1167515.1167511

[15] G. Cavalcanti, P. Borba, and P. Accioly, “Evaluating and improving semistructured merge,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133883

[16] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, “Intellimerge: A refactoring-aware software merging technique,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360596

[17] N. A. A. Khleel and K. Nehéz, “Merging problems in modern version control systems,” *Multidiszciplináris tudományok*, vol. 10, no. 3, p. 365376, 2020. [Online]. Available: http://dx.doi.org/10.35925/j.multi.2020.3.44

[18] W. K. Edwards, “Flexible conflict detection and management in collaborative applications,” in *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST ’97. New York, NY, USA: Association for Computing Machinery, 1997, p. 139148. [Online]. Available: https://doi.org/10.1145/263407.263533

[19] G. Cavalcanti, “What merge tool should i use?” in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 19–20. [Online]. Available: https://doi.org/10.1145/3135932.3135943

[20] C. Brindescu, “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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 952–955. [Online]. Available: https://doi.org/10.1145/3236024.3275430

[21] T. Mens, “A state-of-the-art survey on software merging,” *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, 2002.

[22] B. Berliner, “Cvs ii: Parallelizing software development,” in *Proc. The Advanced Computing Systems Professional and Technical Association (USENIX) Conf.*, 1990, pp. 22–26.

[23] W. F. Tichy, “Rcs — a system for version control,” *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380150703

[24] T. T. Nguyen, H. A. Nguyen, N. H. Pham, and T. N. Nguyen, “Operation-based, fine-grained version control model for tree-based representation,” in *Fundamental Approaches to Software Engineering*, D. S. Rosenblum and G. Taentzer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 74–90.

[25] D. Asenov, B. Guenat, P. Müller, and M. Otth, “Precise version control of trees with line-based version control systems,” in *Fundamental Approaches to Software Engineering*, M. Huisman and J. Rubin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 152–169.

[26] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 313–324. [Online]. Available: https://doi.org/10.1145/2642937.2642982

[27] B. Fluri, M. Wursch, M. PInzger, and H. Gall, “Change distilling:tree differencing for fine-grained source code change extraction,” *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[28] N. Castanho, “Semantic conflicts in version control systems,” Lisbon, Portugal, 2021, available at http://hdl.handle.net/10451/50658.

[29] T. Mens, “Conditional graph rewriting as a domain-independent formalism for software evolution,” in *Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, ser. AGTIVE ’99. Berlin, Heidelberg: Springer-Verlag, 1999, p. 127143.

[30] K. Pan, E. J. Whitehead, and G. Ge, “Textual and behavioral views of function changes,” in *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, ser. TEFSE ’05. New York, NY, USA: Association for Computing Machinery, 2005, p. 813. [Online]. Available: https://doi.org/10.1145/1107656.1107659

[31] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *SIGPLAN Not.*, vol. 23, no. 7, p. 3546, jun 1988. [Online]. Available: https://doi.org/10.1145/960116.53994

[32] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, “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*, ser. ESEC/FSE ’11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 190–200. [Online]. Available: https://doi.org/10.1145/2025113.2025141

[33] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, “The impact of structure on software merging: Semistructured versus structured merge,” in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’19. IEEE Press, 2020, pp. 1002–1013. [Online]. Available: https://doi.org/10.1109/ASE.2019.00097

[34] Y. Lin, J. Gray, and F. Jouault, “Dsmdiff: a differentiation tool for domain-specific models,” *European Journal of Information Systems*, vol. 16, no. 4, pp. 349–361, 2007. [Online]. Available: https://doi.org/10.1057/palgrave.ejis.3000685

[35] A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, “Composing domain-specific design environments,” *Computer*, vol. 34, no. 11, pp. 44–51, nov 2001.

[36] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Boston, MA: Addison-Wesley, 2009.

[37] OMG, *OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1*, Object Management Group Publication, Rev. 2.4.1, Jun. 2013. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1

[38] T. Berlage and A. Genau, “A framework for shared applications with a replicated architecture,” in *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST ’93. New York, NY, USA: Association for Computing Machinery, 1993, pp. 249–257. [Online]. Available: https://doi.org/10.1145/168642.168668

[39] M. S. Feather, “Detecting interference when merging specification evolutions,” in *Proceedings of the 5th International Workshop on Software Specification and Design*, ser. IWSSD ’89. New York, NY, USA: Association for Computing Machinery, 1989, pp. 169–176. [Online]. Available: https://doi.org/10.1145/75199.75226

[40] M. Koegel, M. Herrmannsdoerfer, O. von Wesendonk, and J. Helming, “Operation-based conflict detection,” in *Proceedings of the 1st International Workshop on Model Comparison in Practice*, ser. IWMCP ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 21–30. [Online]. Available: https://doi.org/10.1145/1826147.1826154

[41] M. Ellis, S. Nadi, and D. Dig, “Operation-based refactoring-aware merging: An empirical evaluation,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, p. 26982721, dec 2022. [Online]. Available: https://doi.org/10.1109/TSE.2022.3228851

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

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[44] D. B. Garlan and P. L. Miller, “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*, ser. SDE 1. New York, NY, USA: Association for Computing Machinery, 1984, pp. 65–72. [Online]. Available: https://doi.org/10.1145/800020.808250

[45] A. J. Ko and B. A. Myers, “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*, ser. CHI ’06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 387–396. [Online]. Available: https://doi.org/10.1145/1124772.1124831

[46] A. L. Santos, “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*, ser. Programming ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 120–125. [Online]. Available: https://doi.org/10.1145/3397537.3397561

[47] T. Beckmann, P. Rein, S. Ramson, J. Bergsiek, and R. Hirschfeld, “Structured editing for all: Deriving usable structured editors from grammars,” in *Proceedings of the 2023 CHI Conference on Human*

Factors in Computing Systems, ser. CHI '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3544548.3580785