DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2020

Spork: Move-enabled structured merge for Java with GumTree and 3DM

SIMON LARSÉN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Spork: Move-enabled structured merge for Java with GumTree and 3DM

SIMON LARSÉN

Master in Computer Science
Date: September 1, 2020
Supervisor: Martin Monperrus
Examiner: Benoit Baudry
School of Electrical Engineering and Computer Science
Swedish title: Spork: Strukturerad sammanfogning för Java med GumTree och 3DM

Abstract

With software development becoming an increasingly parallel process in which branching lines of development is common, the shortcomings of unstructured merge techniques are becoming more tangible. In particular, merge conflicts occur frequently and often unnecessarily. Structured merge techniques typically operate on abstract syntax trees instead of raw text, and present major improvements over unstructured merge in terms of avoiding unnecessary conflicts. Structured merge can however be prone to slow execution, has problems dealing with refactorings such as renamings and moves of code blocks, and is often poor at preserving source code formatting.

In this thesis, we implement and evaluate SPORK, a structured merge tool for the JAVA programming language. SPORK is built using move-enabled diff and merge algorithms that can resolve refactoring-related conflicts automatically. The algorithms used are highly efficient, making SPORK scale well with larger merges. SPORK also has features for reusing the original source code of the input files in a merge, leading to improved preservation of source code formatting.

We compare SPORK to the state of the art structured merge tool JDIME in an empirical evaluation on 890 merge scenarios from 119 different open source JAVA projects, comprising a total of 1740 file merges. SPORK and JDIME tend to be equally fast in the median case, with median running times of 1.45 and 1.48 seconds per file merge, respectively. JDIME is the faster tool below the median running time, but SPORK is faster above the median, indicating that SPORK scales better with larger merges. Additionally, SPORK produces fewer but slightly larger conflicts, mostly due to known issues. We also find no statistically significant difference between SPORK's and JDIME's abilities to produce correct merges. In some cases, SPORK's move-enabled algorithms allows it to automatically resolve merges with renamings that cause conflicts for JDIME. In conjunction, these results constitute strong evidence that move-enabled merge is feasible for use in a structured merge tool for the JAVA programming language. SPORK is also more than 4 times better than JDIME at preserving formatting in the median case. Although the results are promising, there are unresolved issues with conflicts related to moves and deletes that reduce SPORK's performance and reliability, necessitating further work.

Sammanfattning

I samband med att mjukvaruutveckling blir en alltmer parallelliserad process så blir tillkortakommanden hos ostrukturerad sammanfogning (eng: merge) mer påtagliga. I synnerhet är sammanfogningskonflikter frekvent förekommande, och många av dessa sker i onödan. Strukturerad sammanfogning arbetar vanligtvis på abstrakta syntaxträd istället för på text, och kan till stor del avhjälpa problem med onödiga konflikter. Det finns dock aspekter av strukturerad sammanfogning som är i behov av förbättring, bland annat lång körtid, problem att hantera omstruktureringar såsom namnändringar och förflyttningar av kodblock, samt undermålig förmåga att bevara formatering.

I detta examensarbete implementerar och utvärderar vi SPORK, ett strukturerat sammanfogningsverktyg för program skrivna i JAVA. SPORK bygger på förflyttningsmedvetna (eng: move-enabled) skillnads- och sammanfogningsalgoritmer som ofta automatiskt kan sammanfoga filer trots omstruktureringar. Dessa algoritmer är även effektiva, vilket gör att SPORK skalar väl med större sammanfogningar. Utöver detta har SPORK funktionalitet för att direkt återanvända källkod från filversionerna som sammanfogas, vilket bättre återskapar den ursprungliga formateringen.

Genom en experimentell utvärdering jämförs SPORK med det toppmoderna strukturerade sammanfogningsverktyget JDIME på 890 sammanfogningsscenarion från 119 olika JAVA-projekt med öppen källkod, som totalt uppgår till 1740 olika filsammanfogningar. Viktiga resultat inkluderar att SPORK och JDIME är lika snabba i medianfallet, med körtider på 1.45 respektive 1.48 sekunder per filsammanfogning. Under medianen är SPORK marginellt långsammare än JDIME, men över medianen är SPORK betydligt snabbare, vilket tyder på att SPORK skalar bättre med större sammanfogningar. SPORK producerar även färre men något större konflikter än JDIME, vilket till stor del beror på kända problem. Ingen statistiskt signifikant skillnad mellan verktygens förmågor att sammanfoga korrekt kan påvisas, men det finns bevis för att SPORKS förflyttningsmedvetna algoritmer låter det automatiskt lösa sammanfogningar med namnändringar som orsakar konflikter för JDIME. Sammantaget utgör dessa resultat starkt bevis för att förflyttningsmedvetna algoritmer är användbara i ett praktiskt sammanfogningsverktyg för JAVA-kod. Vidare finner vi även att SPORKS återanvändning av källkod leder till en mer än 4 gånger bättre bibehållning av formatering i medianfallet gentemot JDIME. Trots dessa lovande resultat finns det fortfarande olösta problem med konflikter relaterade till förflyttningar och borttagningar av kod som sänker SPORKS prestanda och pålitlighet, och ytterligare arbete krävs för att avhjälpa detta.

Acknowledgments

This thesis marks my greatest achievement to date, and I want to express my sincerest gratitude to everyone that has helped me along the way.

I first want to thank my supervisor, Prof. Martin Monperrus. Not only did Martin provide invaluable and frequent feedback on all parts of the thesis, but his advice also allowed me to overcome several technically challenging obstacles that had completely halted my progress.

I also want to thank my examiner, Prof. Benoit Baudry, for timely and concise feedback on the thesis, and always being quick to respond to questions.

I also want to thank Prof. Jean-Rémy Falleri for his assistance throughout the thesis project. Most notably, Jean-Rémy provided support on the use of GUMTREE, starter code for implementing 3DM-MERGE, as well as extensive feedback on the experimental methodology.

Finally, I want to thank those closest to me for the unwavering support and encouragement throughout the years of my studies. I especially want to thank my father, Per Larsén, who sadly passed away shortly before I began working on this thesis. The completion of this thesis is in no small part a commemoration of my father, who both enabled and encouraged my long years of studies in natural languages, materials science and computer science, ultimately leading me to where I am today.

Contents

1 Introduction **1**
- 1.1 Problem statement . . . 2
- 1.2 Research questions . . . 4
- 1.3 Contributions . . . 5
- 1.4 Thesis overview . . . 5

2 Background **7**
- 2.1 Version control systems . . . 7
  - 2.1.1 Repositories and commits . . . 7
  - 2.1.2 Centralized and distributed version control systems . . . 8
  - 2.1.3 Branching in Git . . . 9
  - 2.1.4 Merging in Git . . . 10
- 2.2 File differencing . . . 11
  - 2.2.1 Textual diff . . . 12
  - 2.2.2 Structured diff . . . 12
- 2.3 File merging . . . 15
  - 2.3.1 Two-way and three-way unstructured merge . . . 15
  - 2.3.2 Three-way structured merge . . . 17

3 State of the art on merging **21**
- 3.1 Structured and semistructured merge . . . 21
  - 3.1.1 Structured diff algorithms . . . 22
  - 3.1.2 Structured merge tools . . . 23
  - 3.1.3 Semistructured merge tools . . . 25
- 3.2 Merge conflict resolution helpers . . . 27
- 3.3 Merge conflict prediction and workspace awareness . . . 29
- 3.4 Properties of branching and merging . . . 30
- 3.5 Benchmarking merge tools . . . 34

4 Primer on 3DM-merge **37**
4.1 PCS triples, content tuples and the change set 37
4.1.1 PCS triples 38
4.1.2 Content tuples 39
4.1.3 Consistency 40
4.1.4 Well-formedness 40
4.2 Matchings and class representatives 41
4.3 Inconsistencies and conflicts 42
4.4 The merge algorithm 45

5 Spork **50**
5.1 Parsing 50
5.2 Tree matching 52
5.3 Merging 53
5.3.1 Mapping to class representatives 53
5.3.2 Converting an AST to a change set 54
5.3.3 The SPORK-3DM algorithm 56
5.4 Converting from change set to SPOON AST 58
5.4.1 Building an intermediate tree 58
5.4.2 Building a SPOON AST 60
5.4.3 Merging duplicated type members 61
5.5 Pretty-printing 61
5.6 Merge conflicts 62
5.6.1 Insert/insert conflicts 62
5.6.2 Delete/delete conflicts 62
5.6.3 Insert/delete conflicts 64
5.6.4 Delete/edit conflicts 65
5.6.5 Move conflicts 66
5.7 Fallback mechanisms 68
5.7.1 Hard parent inconsistency elimination 68
5.7.2 Local line-based fallback 68
5.7.3 Global line-based fallback 69

6 Experimental methodology **70**
6.1 Research questions 70
6.1.1 Rationale for RQ1 71
6.1.2 Rationale for RQ2 71
6.1.3 Rationale for RQ3 71
6.1.4 Rationale for RQ4 72

6.2 Dataset 72
6.2.1 Filtering projects 73
6.2.2 Filtering merge commits 74
6.2.3 Executing the experiments 75
6.3 Experiment protocol for RQ1, RQ2 and RQ3 76
6.4 Experiment protocol for RQ4 77
6.4.1 Experiment design 77
6.4.2 Experiment protocol 79
6.5 Statistical tests 80
6.6 Test system 81

7 Experimental results **82**
7.1 Conflict hunk quantities and sizes 82
7.1.1 Conflict hunk quantities 82
7.1.2 Conflict hunk sizes 86
7.2 Running times 87
7.3 Preservation of formatting 90
7.4 Reproduction of merges 91
7.5 Merge failures 93

8 Discussion **94**
8.1 Findings 94
8.2 Threats to validity 97
8.3 Future research 98
8.4 Ethical considerations 100

9 Conclusions **101**

Bibliography **103**

A Projects used in experiments **111**

Chapter 1

Introduction

Merging of source code is an increasingly important activity in software development. With the rise of *distributed version control systems* (DVCS) such as GIT and MERCURIAL that facilitate easy branching and parallell development [1], a corresponding increase in the cost of integrating changes can also be discerned [2]. Due to the fundamental mechanisms of DVCS, there is in fact no feasible way to avoid branching in a multi-developer project [1, 3]. As a result, the use of DVCS necessitates a way to *merge* parallell versions into a single consistent state.

The state of the practice is to merge source code by its raw text, typically with a line being an indivisible unit [4]. This is often referred to as *unstructured*, *line-based*[1] or *textual* merge. Although unstructured merge is simple and efficient, source code is not well represented by its raw text. Any given set of syntactical elements can typically be formatted in an unbounded number of ways, and there are also syntactical elements such as parentheses that in many situations are optional, and thus entirely cosmetic. For example, the three statements **return**␣a+b;, **return**␣a␣+␣b; and **return**␣(a+b); are all structurally and semantically equivalent, but a line-based comparison deems them all to be different due to differing use of whitespace[2] and redundant parentheses. Such minute differences are prone to cause unnecessary *merge conflicts*. These must be resolved manually by the developer, which is an error prone activity that takes considerable effort, and often introduces bugs [1, 2, 4, 5, 6].

This is not to say that merge conflicts are inherently bad. If developers make changes that are truly inconsistent, such as changing the name of the

[1] Line-based merge is the most common form of unstructured merge, and so the terms are often used interchangeably. Note however that unstructured merge does not necessarily have to be line-based.

[2] Note that ␣ represents an explicit space character

same method to two different things, then the merge algorithm should make this known. As unstructured merge lacks understanding of the meaning of the source code, it is also prone to failing to detect conflicts, such as duplicated fields and methods in a JAVA program [7]. These problems make unstructured merge unsuitable for the merging of source code, as it is both prone to find conflicts where there are none (*false positives*), and fail to find conflicts that actually exist (*false negatives*).

Clearly, there is a need for a merge technique that is more aware of the source code structure, both in order to avoid uneccessary conflicts and to discover conflicts that unstructured merge cannot. This forms the key motivation for *structured* merge, where the merge process acts on a structured representation of the source code instead of the raw text. Commonly, this structure is the *abstract syntax tree* (AST) [7, 8, 9], which is typically less influenced by formatting differences and thereby avoids many of the inherent problems of unstructured merge. A structured merge may also take semantics into account[3], which in the JAVA programming language may take the form of detecting duplicated fields, or acknowledging that the order of methods in a class is not important [7].

1.1 Problem statement

With this thesis, we attempt to further the field of structured merge with the goal of bringing it closer to being more broadly adopted by developers, while at the same time improving upon the state of the art. We delimit the thesis to concern merge tools for the JAVA programming language. While some problems may generalize to other languages, especially from the C-family of languages, recent research shows that merge characteristics differ between languages [10].

We aim to implement a tool for merging JAVA source code that fulfills five primary requirements. First, the tool should operate on a fully resolved AST. By fully resolved, we mean that AST nodes are not allowed to be simplified, such as replacing the body of a method node with text. This is the defining feature that makes it a fully structured merge. It should also be able to draw upon JAVA semantics to automatically resolve common forms of conflicts, such

[3] Mens [4] distinguishes between *syntactic* merge, which acts only on syntax, and *semantic* merge, which incorporates knowledge of language semantics. We make no such distinction, and use the term structured merge to refer to any merge algorithm that operates on a structured representation of source code.

as ordering conflicts that occur when the order of methods is ambiguous[4] [11].

Second, the tool must use diff and merge algorithms capable of detecting and acting upon moved and updated AST nodes. We refer to such algorithms as *move-enabled* [12], and to the best of our knowledge, no current fully structured merge tool for any programming language employs such algorithms. Recent research shows that refactorings are a common cause of conflicts [13], and we believe that move-enabled merge may be able to avoid conflicts in the face of refactorings. This can help resolve relatively simple conflicts caused for example by renamings of methods or classes, which for a non-move-enabled structured merge algorithm requires non-trivial extensions [14]. Theoretically, more involved conflict situations can also be automatically resolved, such as when a code block is extracted into a method of its own in one revision being merged, and the contents of the block is edited in the other revision.

Third, the tool must be fast enough to be useful in practice, and should improve upon the state of the art. Structured merge is known for being slow [7], which is one aspect hindering further adoption. The algorithms involved are typically $O(n^2)$ or worse, so scaling is a problem. Small merges may finish adequately fast, but larger merges may take a long time to complete. As merging is done in an interactive computing environment, where the developer has to wait for it to finish, it is important that it is not so slow as to cause frustration. An additional concern in context to this thesis is that accounting for moves makes the structured merge problem even harder [12], and it is unknown whether or not a move-enabled structured merge can even be made fast enough to be useful in practice.

Fourth, the tool must to a large extent preserve the original formatting of the merged source code. As a structured merge acts upon an abstract representation of the source code, it must necessarily conclude in a pretty-printing phase. If the formatting of the original source code is not taken into account, the pretty-printing phase may fundamentally alter the formatting of the code. This is problematic, as it is well known that developers value formatting, and use tools such as CHECKSTYLE[5] to enforce project specific rules. Research also shows that formatting plays an important role in code readability [15].

Fifth, the tool should perform comparably to the current state of the art when it comes to correctly merging files. The end goal of using a move-enabled merge is to improve in this aspect as well, but in light of the other improvements we seek to make, we consider achieving performance on par

[4] Recall that the order of methods in a JAVA program is not important, and so a resolution is to pick any order.

[5] `https://checkstyle.sourceforge.io/`

with the current state of the art to be adequate. As there is no general way of determining the correctness of a merge, we must also find a way to approximate correctness.

1.2 Research questions

In order to address the problems presented in Section 1.1, we implement a structured merge tool called SPORK. The merge is based on the GUMTREE [12] and 3DM-MERGE [16] algorithms, both of which are move-enabled, and is built on top of the SPOON metaprogramming library [17]. We define our research questions with the goal of evaluating SPORK in relation to the current state of the art in the aspects presented in Section 1.1. We choose the structured merge tool JDIME as a baseline for the state of the art in structured merge, as it to the best of our knowledge is the most extensively validated structured merge tool for the JAVA programming language [7, 9, 14, 18, 19]. With this context, the research questions are as follows.

> RQ1: How does SPORK compare to JDIME in terms of amounts and sizes of conflicts?
>
> RQ2: How does SPORK compare to JDIME in terms of running time?
>
> RQ3: How does SPORK compare to JDIME in terms of preserving source code formatting?
>
> RQ4: How does SPORK compare to JDIME in terms of producing merges that are semantically equivalent to the merges committed by developers?

While RQ1, RQ2 and RQ3 should be self-explanatory based on the problem statement presented in Section 1.1, RQ4 requires some additional context. Recent work in comparing merge tools focuses primarily on examining merge conflicts [7, 10, 19, 20, 21]. While valuable, we argue that this fails to capture the bigger picture, as the vast majority of merges are conflict-free. A low frequency of conflicts is in fact one of the primary motivations for considering structured merge. RQ4 represents an attempt to address the conflict-free cases by comparing the merge results produced by the merge tools to those produced by the developers, without considering non-semantic details such as formatting, method ordering and local variable naming. How this is achieved is explained in Chapter 6.

1.3 Contributions

With this thesis, we make 6 primary contributions to the field of structured merging. They are as follows.

- We provide a comprehensive overview of the field of merging, with detailed descriptions of the fundamental principles in Chapter 2, as well as a large selection of the state of the art in Chapter 3 and Chapter 4.
- We implement SPORK[6], a move-enabled structured merge tool with a modular design, available under a permissive open source license. See Chapter 5.
- We propose a new way of evaluating the general performance of a merge tool for the JAVA programming language by way of normalized bytecode comparison. See Chapter 6.
- We provide a well-documented benchmarking suite that can be reused as-is to evaluate JAVA merge tools, or adapted to be used with other languages. See Chapter 6.
- We show that a SPORK is fast and accurate enough to be used in practice, which by extension shows that a move-enabled, fully structured merge for JAVA is feasible. See Chapter 7.
- We investigate problems of move-enabled merge that do not apply to non-move-enabled merge. See Chapter 7 and Chapter 8.

1.4 Thesis overview

The layout of the thesis is as follows. Chapter 2 provides background information on version control systems, as well as unstructured and structured differencing and merging. Chapter 3 presents a comprehensive overview of the state of the art in merging, as well as related techniques for reducing the effort in integrating changes in parallel development. Chapter 4 presents the 3DM-MERGE algorithm, which although being primarily related work warrants its own chapter due to it being a fundamental part of the thesis. Chapter 5 details SPORK itself, and it should be noted that to understand Chapter 5, one

[6] `https://github.com/KTH/spork`

must be familiar with the concepts presented in Chapter 4. Chapter 6 then details the methodology for the experimental evaluation, and Chapter 7 presents the experimental results. Chapter 8 contains a discussion of the results, and Chapter 9 presents the conclusions.

Chapter 2

Background

This chapter covers the background knowledge required to understand the rest of the thesis. Section 2.1 presents version control systems, which pose the primary reason for why this thesis is relevant. Section 2.2 describes the basic principles of file differencing, and Section 2.3 does the same for file merging.

2.1 Version control systems

A *version control system* (VCS) is a system for managing versions of files, which are commonly referred to as *revisions*. Among other things, a VCS keeps track of what changes were made to which files and at what time, and typically also who made those changes [22].

This section presents background information on VCSs and defines the related terminology used throughout the thesis. Section 2.1.1 and Section 2.1.2 define basic VCS terminology and concepts, while Section 2.1.3 and Section 2.1.4 present branching and merging in GIT, respectively.

2.1.1 Repositories and commits

The first important concept is the *repository*. This is where the revision history of any given project[1] is stored. The revision history consists of discrete points in time at which the repository state was saved, typically referred to as *commits*. The user decides both when to commit, and what changes to include in the commit [22, 23, 24]. How individual file revisions and entire commits are stored differs from system to system. For example, the early SOURCE CODE

[1] By project, we mean any non-empty set of files that a user wants to version control together

CONTROL SYSTEM (SCCS) stores the history of any given file as file deltas [23], while the currently popular GIT stores a snapshot of the entire project with each file in its entirety for each commit [25]. In both cases, the state of the files at any given commit can be restored from the repository, in the case of SCCS by applying the file deltas from some known starting point, and in GIT by simply reading the stored file revision[2].

2.1.2 Centralized and distributed version control systems

A *centralized version control system* (CVCS) is built on a client-server architecture. A central repository that contains the entire version history acts as a server, and individual developers can request a limited portion of said history via a client [22, 24]. When a developer has made changes that they wish to integrate into the version history, they must connect to the server in order to create a commit. This means that committing is tied in with publishing changes to the server [26]. Major disadvantages of CVCS include that it requires a network connection, preventing offline work, and also that it hinders developers from working incrementally and trying out experimental changes due to the intertwining of committing and publishing [26, 27]. Early examples of VCS, such as SCCS, RCS and CVS, are all centralized [23]. SUBVERSION is a more modern example of a CVCS, and is still widely used [26].

In a *distributed version control system*[3] (DVCS), there is no central server with a ground truth for the state of the repository. Instead, a repository can exist on an unbounded amount of machines, and each copy of the repository contains the entire version history [22, 23, 24]. As a consequence, commits can be created locally without communicating with any server, which is one of the strengths of DVCS [22, 26, 27]. Although there is no central repository by design, developers still tend to keep a primary repository on a hosting service such as GITHUB[4] or GITLAB[5]. Two notable DVCSs are MERCURIAL and GIT, the latter being the most popular one by far [26]. DVCS started to gain popularity around the end of the 2000s [28], and has since seemingly surpassed CVCS in popularity [26]. The increased use of DVCS has also lead to an increase in so-called *branching* [1, 28], which is discussed next.

[2]It should be noted that GIT does compress file histories into deltas from time to time to save space, but this is an optimization that is not part of the core version history model [25]

[3]Sometimes referred to as *decentralized* VCS

[4]`https://github.com`

[5]`https://gitlab.com`

2.1.3 Branching in Git

Modern VCSs allow for a single repository to have multiple independent lines of development, called *branches*. Among other things, branches allow developers to try out experimental changes, keep platform dependent code separate from each other and maintain released versions of software independently from development on upcoming releases [29]. Branching is possible in many VCSs, both centralized and distributed [24], but this thesis only treats branching in GIT. This is in part due to GIT's dominant place in the software engineering space, and in part due to branching being a central part of GIT's workflow [24].

The simplest possible repository has only a single branch, and its commit history is a linear sequence of commits, as illustrated in Figure 2.1. In GIT, the mainline branch is often called `master` by convention. In all further examples, assume that `master` is the mainline branch. Also note how the commit history is represented in reverse in the sense that later commits reference their predecessors (called *parents* in GIT), rather than the other way around. This is how GIT works, the commit history graph is traversed backwards [25].

Figure 2.2 shows a situation in a repository with multiple branches, with two new branches at different stages of development. First, there is the `release` branch, which evidently has yet to diverge from the `master` branch, it is simply a bit behind. This situation occurs when a new branch is created, but no commits have yet been created on it. Then, it does not diverge from the branch it was created from, but simply lags behind as new commits are created on the original branch. Second, the `feature` branch has diverged from `master`, as new commits have been created on it. The changes introduced by commits 4 and 5 are not present on the `master` branch, and likewise the changes introduced by commit 3 are not present on the `feature` branch. This is an example of *explicit* branching [1]. A user has decided to create the `feature` branch, and done so with a command.

Bird et al. [1] claim that branching is unavoidable in a DVCS like GIT, as implicit branches are created when different developers create commits in

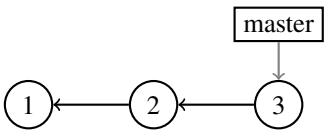


Figure 2.1: *A linear commit history. Circles are commits, and rectangles are branches.*

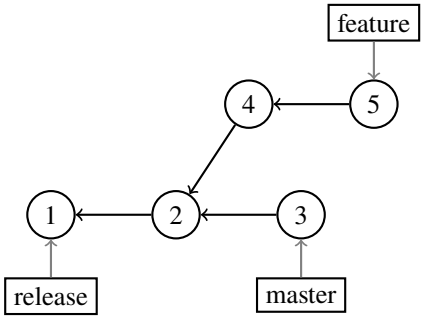
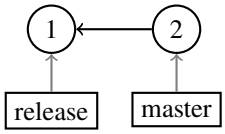
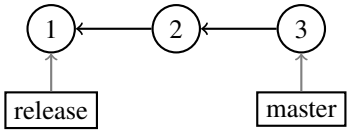


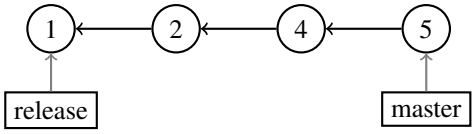
Figure 2.2: *A branching commit history. Circles are commits, and rectangles are branches.*



(a) *Original repository*



(b) *Alice's copy*



(c) *Bob's copy*

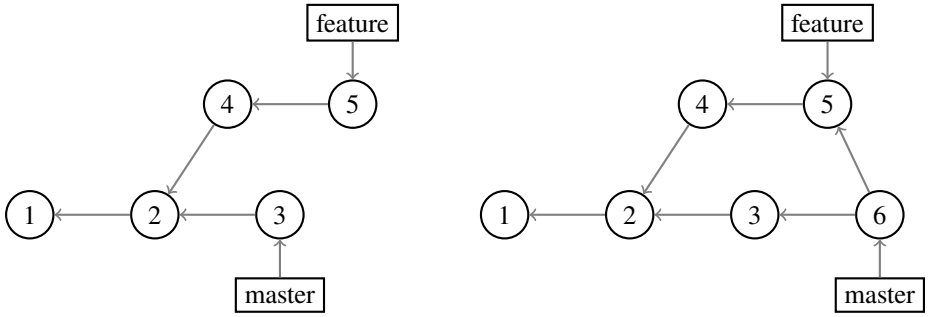
Figure 2.3: *Implicit branching through commits in different repository copies*

their local repository copies. For example, consider that Alice and Bob both copy a repository at the same time, such that they get identical copies. If both Alice and Bob create commits, these always differ[6], and so implicit branching occurs. Figure 2.3 shows the same divergent commit history created explicitly in Figure 2.2, but through implicit branching in Alice's and Bob's local copies. This goes to show that branching is inherently part of a DVCS, and so a robust mechanism for joining divergent commit histories is essential.

2.1.4 Merging in Git

When branches diverge as discussed in Section 2.1.3, there is a need to be able to join them together again in a so-called *merge*. Figure 2.4 presents a merge scenario in which the `feature` branch is merged into `master`. On a high level, the merge process consists of integrating changes from both branches

[6]Even if the content is identical, the time stamp and committer is not



(a) *A branching commit history*

(b) `feature` *merged into* `master` *with merge commit 6*

Figure 2.4: *Schematic drawing of a merge*

into a single consistent state, creating a new commit (here, commit 6) and advancing the `master` branch to it. GIT does this by performing a *three-way merge* between the tips of the branches to be merged, and their *best common ancestor* [25], sometimes referred to as the *merge base*[7]. In the rest of this thesis, the common ancestor is referred to simply as *base* (here, commit 2), the branch being merged into as *left* (here, `master`), and the other branch as *right* (here, `feature`). The left and right revisions are sometimes collectively referred to as the *current* revisions.

The difficulty of merging two branches depends on the changes introduced in them. In a general sense, all changes from both branches should be present in the merge. As long as there is no overlap between the files touched by the left and right branches, the merge resolution is trivial; simply pick the file revision that is different from the base revision. When there is overlap, there is a *conflict* in the merge. Ideally, conflicts should be resolved automatically, as manual resolution of merge conflicts is both error-prone and time-consuming [2, 4, 5, 6], and developers dread doing it [1]. The situation of interest for this thesis is when the same file has been edited in both branches, as this necessitates a merge of said file. This is called a *file merge*, and is discussed further in Section 2.3.

2.2 File differencing

In order to merge file revisions, it is typically a necessity to first determine how they differ. This is called *differencing*. The end goal of a diff algorithm

[7] `https://git-scm.com/docs/git-merge-base`

operating on files `A` and `B` is to compute an *edit script*, which is a list of operations that transforms `A` into `B`, or vice versa [12, 30]. In general, the shorter the edit script the better its quality. Differencing is often a two-step process, consisting of first matching the common parts of `A` and `B`, and based on these matchings inferring an edit script. Larger matchings typically lead to smaller edit scripts [12, 30].

Section 2.2.1 introduces the fundamental concepts of file differencing by describing textual diff. Section 2.2.2 then builds on the concepts from Section 2.2.1 to illustrate structured diffs.

2.2.1 Textual diff

In textual diff, two files are compared by their raw text, most often with a line of text as an atomic unit [4]. One of the most famous textual diff algorithms is the one proposed by Myers [30], which is still used as the default diff algorithm in GIT[8]. It is a longest common subsequence (LCS) algorithm, that greedily matches the longest subsequences that two files have in common. Note that a subsequence does not need to be contiguous in any one file. For example, the sequence `abc` has a subsequence `ac`. There may also be several longest common subsequences. For example, `abcd` and `abdc` share both `abc` and `abd`. The Myers algorithm finds one such longest subsequence.

While the details of the Myers algorithm are not important for this thesis, the general concept of matching common parts is. Applied to the lines of two files, it conceptually calculates a matching as shown in Figure 2.5c. The edit script can then be inferred by considering the lines that fall outside of matched blocks. For example, considering the left revision as the new revision, **print**(tot) is an insertion, as it is not present in the right revision. Considering the right revision as the new revision instead, **print**(tot) is a deletion by the same logic. Myers [30] shows a systematic approach for computing an edit script of deletions and insertions from an LCS, and proves that maximizing the matched subsequence in fact minimizes the edit script.

2.2.2 Structured diff

In structured diff, two files are compared not by their raw text, but by some underlying structure that the text encodes [4]. The most relevant structure for this thesis is the *abstract syntax tree* (AST), which is a precise representation of the structure that source code encodes. As with textual differencing, tree

[8] `https://git-scm.com/docs/git-diff`

```
1  def arith_sum(n):
2    tot = 0
3    print(tot)
4    for i in 1..n:
5      tot += i
6    return tot
```

(a) *Left revision*

```
1  def arith_sum(n):
2    tot = 0
3    for i in 1..n:
4      tot = tot + i
5    print(tot)
6    return tot
```

(b) *Right revision*

```
------------------------------------------ start
def arith_sum(n):          def arith_sum(n):
  tot = 0                    tot = 0
------------------------------------------ end
  print(tot)
------------------------------------------ start
  for i in 1..n:             for i in 1..n:
------------------------------------------ end
    tot += i                   tot = tot + i
                             print(tot)
------------------------------------------ start
  return tot                 return tot
------------------------------------------ end
```

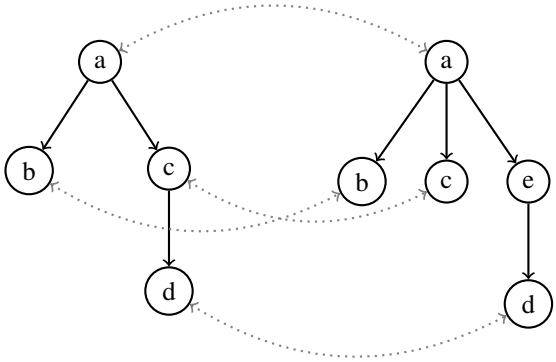
(c) *Visualization of an LCS. The start and end markers in the right margin mark the start and end of matching blocks. The LCS is the ordered union of the matching blocks.*

Figure 2.5: *Example of an LCS matching*

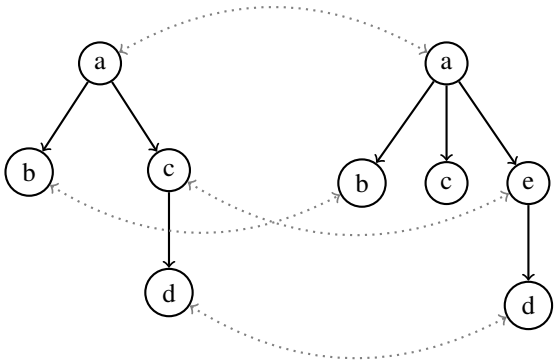
differencing typically consists of a matching step and an edit script inference step [12]. While basic tree diff algorithms can detect insertions and deletions, more advanced algorithms can also detect moved and updated nodes. Tree differencing that considers moves and updates is a difficult topic that has been studied extensively [31, 32], and is an NP-hard problem [12].

As an example, consider the tree matching scenario in Figure 2.6a, which presents one possible matching of two trees. Nodes `a`, `b` and `c` are matched together, and so can be inferred to be unaltered. `e` is not matched so it is an insertion. `d` could also be matched if the matching algorithm is move-aware, and so the edit script can be inferred as two operations: insert `e` as a child of `a`, and move `d`. This illustrates a key difference between a move-aware and non-move-aware tree differencing. A naive top-down matching would likely not match the `d` nodes, which would negatively affect the edit script. In particular, the move of `d` would then be modeled as two operations: delete `d` from `c`'s child list and insert `d` as a child of `e`.

The matching in Figure 2.6b is equally valid. Here, `c` in the left tree is matched to `e` in the right tree, implying an update operation, and a new `c` node is inserted. Thus, another valid edit script would be: update `e` to `c`, insert `c` ahead of `e`.



(a) *A matching between two trees, where* `d` *is detected as moved*



(b) *A matching between two trees, where* `c` *has been updated to* `e`

Figure 2.6: *Schematic drawing of two possible matchings between two trees. Solid lines represent actual edges, while dashed lines represent matchings between tree nodes.*

2.3 File merging

In Section 2.1.4, GIT is noted to perform a so-called three-way merge between the tips of the left and right branches and their best common ancestor, the merge base. This is not only important for determining which files to merge, but it is also important in each individual file merge. Section 2.3.1 presents the general mechanisms of a three-way file merge, and also explains why it is superior to its simpler alternative, the *two-way merge*. Section 2.3.2 then presents structured three-way merge, and exemplifies why it is more precise than unstructured merge.

2.3.1 Two-way and three-way unstructured merge

In a merge of two revisions left and right, it is typically the case that both revisions are derived from a base revision, as described in Section 2.1.4. In a two-way merge, only the two current revisions of the file are considered. This way, it is possible to determine where the revisions differ, but it is not possible to determine which revision introduced what change [4]. In a sense, every difference is a conflict. As a concrete example, consider Figure 2.5c again. The parts that the revisions have in common can be seen in the matched blocks. The parts outside of the matched blocks are where the revisions differ, but there is not enough information to tell how the differences came to be. For example, there is no way to tell if the left revision added the print statement before the loop, or if the right revision removed it.

To be able to draw conclusions about where changes occurred, a reference point is needed, namely the base revision of the file [4]. By comparing the left and right revisions to their common ancestor, it is possible to align the segments that all three revisions have in common. This is shown in Figure 2.7d. Note how considering only the left and base column of Figure 2.7d gives the LCS matching of left and base, and analogously for base and right by covering up the left column. Aligned with the base revision, it is clear which edits occurred in which revision. Everything in a matched block is unchanged in both revisions, and anything not in a matched block has been changed in at least one revision. For example, the print statement before the loop has clearly been removed in the right revision as it is present in both base and left, but not in right.

A problem occurs when the left and right revisions disagree on an edit, which is the case in the final unmatched block. This means that the same textual area has been changed in both the left and right revisions, which results in

```
1 def arith_sum(n):
2   tot = 0
3   print(tot)
4   for i in 1..n:
5     tot += i
6   return tot
```

(a) *Left revision*

```
1 def arith_sum(n):
2   tot = 0
3   print(tot)
4   for i in 1..n:
5     tot = tot + i
6   return tot
```

(b) *Base revision*

```
1 def arith_sum(n):
2   tot = 0
3   for i in 1..n:
4     tot = tot + i
5   print(tot)
6   return tot
```

(c) *Right revision*

```
------------------------------------------------------------ start
def arith_sum(n):    def arith_sum(n):    def arith_sum(n):
  tot = 0              tot = 0              tot = 0
------------------------------------------------------------ end
  print(tot)           print(tot)
------------------------------------------------------------ start
  for i in 1..n:       for i in 1..n:       for i in 1..n:
------------------------------------------------------------ end
    tot += i             tot = tot + i        tot = tot + i
                                              print(tot)
------------------------------------------------------------ start
  return tot           return tot           return tot
------------------------------------------------------------ end
```

(d) *The left and right revisions aligned against the base revision. The start and end markers in the right margin mark the start and end of matching blocks.*

Figure 2.7: *Example three-way file merge*

a merge conflict [33]. Figure 2.8a shows what this typically looks like in GIT[9]. The `<<<<<<<` marker marks the start of the left revision's contribution, `=======` separates the left and right revisions' contributions, and `>>>>>>>` marks the end of the right revision's contribution. When the word merge conflict is used, it most often refers to a conflict block on this form, which is sometimes referred to as a conflict *hunk* [33].

The conflict in Figure 2.8a shows a common characteristic and weakness of line-based merge, namely merge conflicts that cross syntactical boundaries. Not only is this conflict unnecessary, as adding a print statement after a for loop and changing a statement inside the for loop are hardly conflicting operations, but the fact that the conflict crosses a syntactical boundary makes it potentially more confusing. Figure 2.8b shows the expected outcome of this merge, which it seems reasonable should be possible to reach automatically.

Another common problem for line-based merge are *ordering conflicts*, which is when the left and right revisions add unordered elements in the same position [11]. The archetypical ordering conflict is caused by both current revisions inserting methods in the same textual area. In many languages, methods

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

```
1  def arith_sum(n):
2    tot = 0
3    for i in 1..n:
4  <<<<<<< Left revision
5      tot += i
6  =======
7      tot = tot + i
8    print(tot)
9  >>>>>>> Right revision
10   return tot
```

(a) *Line-based merge result, containing a conflict*

```
1  def arith_sum(n):
2    tot = 0
3    for i in 1..n:
4      tot += i
5    print(tot)
6    return tot
```

(b) *Expected result*

Figure 2.8: *Actual and expected merge results from the scenario in Figure 2.7*

are unordered, so it would be possible to automatically resolve such conflicts by simply inserting the methods in any order. However, line-based merge does not know anything about syntactical structure, and so cannot determine that a conflict is an ordering conflict.

2.3.2 Three-way structured merge

With a structured merge algorithm, the conflict encountered in Section 2.3.1 can be automatically resolved. The same principles from the line-based merge apply here, in that individual nodes in the left and right revisions are aligned with the base revision, which allows edits to be inferred. The fundamental difference is that the AST is more granular than lines of text [12], leading to edits that are adjacent or even overlapping in a line-based merge to not be in a structured merge. An AST node also contains more information about the code than a line of text, leading to the possibility of making more informed decisions about the merge. For example, the added information of a method node being unordered lets a structured merge automatically resolve ordering conflicts [7].

A general approach to performing a structured merge is to match nodes across revisions, and then merge the contents and child lists of matched nodes [8, 9, 16]. Merging the child list of a single node is conceptually similar to a textual merge of a whole file, with the unit being an AST node instead of a line of text. While a text file is just a flat list of lines or characters, an AST consists of many child lists that can be merged more or less independently, which largely explains why structured merge is so much more granular than textual

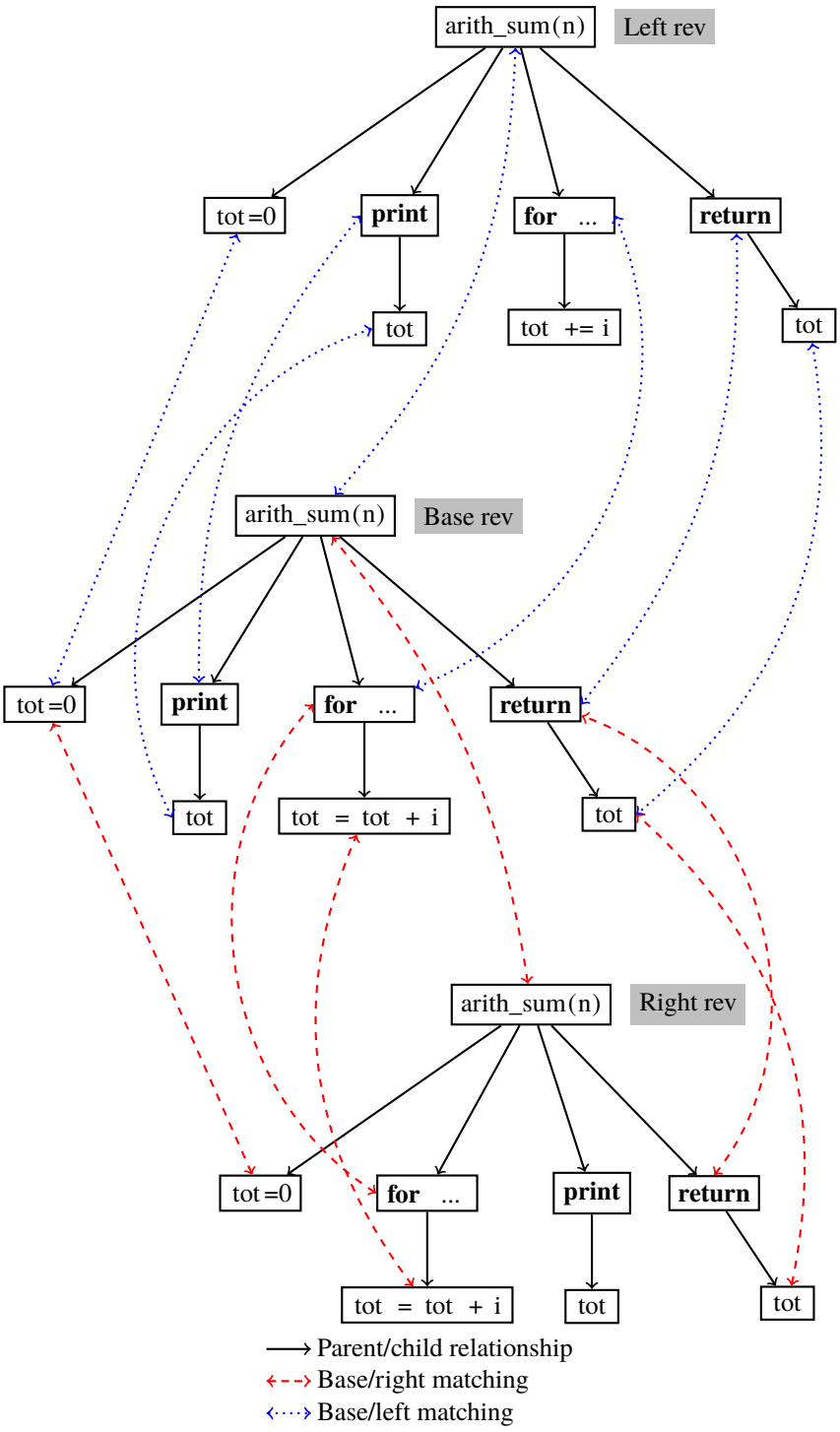


Figure 2.9: *Simplified ASTs of the file revisions from Figure 2.7*

merge. Conflicts occur when different revisions enact edit operations such as insertions or deletions to the same or adjacent nodes in a single child list, much in the same way that edits to adjacent or overlapping lines cause conflicts in textual merge. A detailed description of one approach to structured merge is given in Chapter 4, which covers the 3DM-MERGE algorithm.

Figure 2.9 shows three simplified[10] ASTs, one for each of the left, base and right revisions from Figure 2.7, as well as the base-to-left and base-to-right tree matchings. Edit operations can again be inferred. For example, if a node in base is matched to only one of left or right, it has been deleted in the other. Such is the case for the **print** node before the for loop in the left revision. A node in left or right that is not matched to base is an addition. Such is the case for the **print** node after the for loop in the right revision.

Note how the code elements that cause the conflict in the line-based merge in Figure 2.8a, namely `tot += i` from the left revision and **print** from the right revision, are clearly not adjacent in this model as they belong to different child lists. While the general case requires a systematic approach to merging, it should be plain to see that the changes in Figure 2.9 can be reconciled, as there are no adjacent or overlapping edits. This highlights a great strength of a structured merge, namely that it respects the syntactical boundaries of the code [4]. Even when conflicts do occur in structured merge, they do not cross syntactical boundaries [19], leading to less confusing conflicts.

The tree matchings in Figure 2.9 carry another interesting property. The **print** node in the right revision is unmatched, even though it is precisely equivalent to the **print** node in the base revision, only in a different position. In other words, the node has been moved. A rudimentary tree matching algorithm may not be able to link these two nodes, resulting in the matchings shown. This results in the move of the **print** node being modelled as a deletion at the site before the loop, and an insertion after the loop. In this case, that does not cause a problem, but consider a similar scenario where the left revision slightly alters the print statement to **print**(`"tot:␣" + tot`). Then the deletion in the right revision conflicts with the edit in the left revision, causing a merge that cannot detect moves to report a conflict like the one in Figure 2.10a. A move-enabled merge algorithm can detect the move, and compute the merge shown in Figure 2.10b. Note that a merge being move-enabled means two things: the diff algorithm it uses can detect moved nodes, and the merge algorithm can take advantage of this additional information.

In cases where the left and right revisions both introduce similar or identi-

[10] Simplified in the sense that many separate AST nodes have been collapsed into composite nodes for readability

```
1  def arith_sum(n):
2    tot = 0
3  <<<<<<< Left revision
4    print("tot: " + tot)
5  =======
6  >>>>>>> Right revision
7    for i in 1..n:
8      tot += i
9      tot = tot + i
10   print(tot)
11   return tot
```

(a) *Non move-enabled merge*

```
1  def arith_sum(n):
2    tot = 0
3    for i in 1..n:
4      tot += i
5    print("tot: " + tot)
6    return tot
```

(b) *Move-enabled merge*

Figure 2.10: *Example of move-enabled merge*

cal nodes in the same location in the tree, a left-to-right matching may also be used to identify and reconcile such similar additions [9]. A left-to-right matching is not shown in Figure 2.9, as it would render the figure unintelligible. The concept is further discussed in Section 4.2.

Chapter 3

State of the art on merging

There are multiple viable approaches to lessen the effort involved with merging revisions in VCS. Developing merge tools that provide better merges than the state-of-the-practice unstructured merge tools is one approach. Advances in this field are presented in Section 3.1. It should be noted that Section 3.1 deals solely with diff-based merge tools. There are other merge techniques, such as operation-based merge [4, 34], but they are tangential to this thesis and are therefore not covered in the interest of brevity. Another approach is to provide merge resolution helpers, that upon a conflict can help developers find ways to resolve it. A selection of such tools are presented in Section 3.2. A third approach is to attempt to detect conflicts at an early stage or even predict potential conflicts before they manifest by continuously monitoring developers' parallel work. Several variations of such systems are presented in Section 3.3. These three approaches are largely orthogonal, and making improvements to one often benefits the others. Section 3.4 then presents a selection of studies on the general properties of branching and merging, and Section 3.5 presents studies that benchmark merge tools.

3.1 Structured and semistructured merge

This section outlines related work on structured and semistructured merging. Section 3.1.1 presents structured diff algorithms, Section 3.1.2 presents complete structured merge tools and Section 3.1.3 presents related work on semistructured merge.

3.1.1 Structured diff algorithms

While unstructured diff algorithms operate on raw text, structured diff algorithms operate on a structure that the text encodes [4]. In the case of source code, common representations used by structured diff algorithms are *abstract syntax trees* (AST) [12], or more generalized graph representations [4, 35].

Chawathe et al. [31] present a structured diff algorithm for ordered trees that can handle insertions, deletions, updates and moves of tree nodes, as well as an implementation called LADIFF. They show that tree differencing of two trees T_1 and T_2 is a two-step process. First, a matching of similar nodes between T_1 and T_2 must be computed. Such a matching should pair nodes $t_1 \in T_1$ and $t_2 \in T_2$ if they either match exactly, or are similar. Then, based on the computed matching, an edit script that transforms T_1 into T_2 may be computed. The trees considered by LADIFF are not ASTs per se, but rather a notion of ordered trees to represent structured text documents such as LaTeX and HTML. This leads to assumptions that lessen the generality of the algorithms. In particular, the authors assume that for any leaf node $t_1 \in T_1$, there is at most one leaf node $t_2 \in T_2$ such that t_2 is highly similar to t_1. This may be true for many forms of structured text documents, but the authors note that it may not hold in some domains where formality tends to necessitate repetition, such as in legal texts.

Fluri et al. [32] present CHANGEDISTILLER, a structured diff algorithm that is largely based on LADIFF. CHANGEDISTILLER is however primarily designed for source code, rather than structured text documents. Fluri et al. [32] argue that the primary shortcoming of LADIFF is the strong assumption of the unique matchings of similar leaf nodes. This assumption does not hold for source code in general, as similar and identical leaf nodes are common[1]. CHANGEDISTILLER improves upon the original algorithm, most notably by introducing a better matching algorithm for leaf nodes that does not rely on the unique similarity assumption. However, CHANGEDISTILLER still represents leaf nodes as text, so although designed specifically for source code differencing, it does not truly operate on fully resolved ASTs.

Falleri et al. [12] present GUMTREE, a structured diff algorithm designed specifically for source code differencing on fully resolved ASTs. Much like previous works [31, 32], GUMTREE can handle insertions, deletions, updates and moves of tree nodes. It achieves this primarily by means of its advanced tree matching, which consists of two algorithms. First, a top-down algorithm greedily scans for subtrees $t_1 \in T_1,\ t_2 \in T_2$ such that t_1 and t_2 are isomorphic.

[1] In the case of code duplication, entire subtrees may be identical

This initial phase is sufficient to match trees that are structurally unaltered, but even minor modifications to a tree's structure causes it not to be matched. To address this, a bottom-up algorithm follows. It starts out scanning for pairs of unmatched nodes that share a considerable amount of matched descendants, as computed by the top-down algorithm. When two nodes are successfully matched in this way, their descendants are further matched to each other using an existing edit script algorithm that considers insertions, updates and deletions. When matchings between T_1 and T_2 have been computed, GUMTREE uses the edit script algorithm by Chawathe et al. [31] to compute an edit script. GUMTREE is evaluated both qualitatively and quantitatively, and the authors conclude that GUMTREE presents significant improvements both over unstructured diff, and competing structured diff algorithms such as CHANGEDISTILLER.

Apiwattanapong, Orso, and Harrold [35] present CALCDIFF, a graph-based structured diff algorithm. CALCDIFF specifically targets object-oriented programming languages, and operates on a *control flow graph* (CFG) of the program that is enhanced in various ways to better model object-oriented concepts. The algorithm is designed mostly with advanced code analysis in mind rather than source code merging, such as being able to estimate changes in test suite coverage from changes in the production source code. The authors also present an implementation called JDIFF for JAVA, which is experimentally evaluated in a series of experiments to among other things determine how well JDIFF predicts coverage changes.

3.1.2 Structured merge tools

Structured merge tools make use of a structured diff algorithm to identify changes in different revisions, and based on that information use varying strategies for computing the merge. Such tools have been in development at least since the early 1990s [8], and there are implementations for several programming languages [9, 16].

Leßenich, Apel, and Lengauer [7] and Leßenich [9] present JDIME, an AST-based structured merge tool for JAVA. JDIME implements a three-way structured merge algorithm, complete with its own tree matching and merge strategies. The tree matching is simple but efficient, matching nodes across two trees in a top-down scan and stopping as soon as two nodes do not match. Therefore, it cannot detect moved or updated elements. JDIME then computes a merge based on the tree differences. Both during tree matching and merging, JDIME differentiates ordered and unordered elements. For example, if the exact same method appears in two different places in the files being merged, JDIME still

matches them, as methods are unordered. This allows JDIME to resolve ordering conflicts, but it is not the same as detecting arbitrarily moved nodes as it acts only on a single syntactical level, and only works for unordered elements. Statements in a method are treated as ordered nodes, and their positions distinguish them. In addition to being a structured merge tool, JDIME also implements auto-tuning by first running in unstructured mode, and only switching over to structured mode when a conflict is detected. According to the authors, this improves performance by 10 times on average, while largely maintaining precision [7].

In a more recent publication, Leßenich et al. [14] present an extension to JDIME to handle occurrences of moved and renamed code elements by making use of a novel lookahead mechanism. The mechanism uses heuristics based on known or assumed patterns of how developers refactor code to lookahead a pre-determined distance whenever the top-down search finds a mismatch. While this makes for an efficient implementation, it is constrained by the patterns it searches for and the maximum depth it is allowed to search.

Asenov et al. [36] present a novel approach to structured merging by creating a generic textual representation of ASTs and then merging with a standard line-based merge algorithm. AST matchings can be computed in two ways, either by using stored node identifiers that uniquely identify nodes across versions, or by using a structured diff algorithm to compute the matchings across versions. The technique is shown to work well with GUMTREE as the diff algorithm in the absence of stored node identifiers, which suggests that GUMTREE is a feasible diff algorithm for use in a merge tool.

Lindholm [16] presents 3DM, a three-way structured merge tool for XML documents. While the accompanying implementation is specialized for XML, the merge algorithm is generalizable to any form of tree structure and exhibits some highly desirable traits. First, it can handle insertions, deletions, moves and updates of nodes. This also means that in order to fully utilize 3DM's merge algorithm, a move-aware diff algorithm such as GUMTREE must be used. Second, it is fairly easy to extend by allowing or disallowing certain node contexts. Third, it is commutative[2], meaning that merging revision `A` into revision `B` yields the same result as merging revision `B` into revision `A`. Fourth, it can be implemented in $\mathcal{O}(n*log(n))$ time, n being the total amount of nodes, making it a highly efficient merge.

Rönnau, Pauli, and Borghoff [37] and Rönnau, Philipp, and Borghoff [38] present an approach for merging XML documents using version-deltas (i.e. diffs) that does not rely on three-way merging. The novelty with this approach

[2]Assuming that the diff algorithm used is also commutative

is to compute deltas on one version of a document and apply them to another version of the same document. The authors argue that this approach is suitable in cases where all three versions (left, right and base revisions) are not available[3], perhaps due to bandwidth limitations [37]. In order to determine where and if edit operations should be applied, a *context fingerprint* is calculated based on surrounding nodes. The fingerprint is then used instead of absolute paths, making it possible to apply deltas computed on one version of a document to another version of it. The algorithm can deal with insertions, deletions, updates and moves, and can also detect conflicts, for example by noting that two different edits have the same context fingerprint. The authors do however note that a three-way merge, such as 3DM, offers higher accuracy at the price of higher computational complexity [39].

3.1.3 Semistructured merge tools

Semistructured merge tools represent an attempt to find a middle-ground between structured and unstructured merging in terms of accuracy and running time performance [21]. The idea is to partially resolve ASTs or graphs and merge high-level code elements such as import statements and method headers structurally, but resort to unstructured merge in fine-grained code elements such as method bodies.

Apel et al. [40] are the first to propose semistructured merge with the introduction of FSTMERGE, a framework for building semistructured merge tools. By providing FSTMERGE with an annotated grammar, it can be specialized for a given language. Apel et al. [40] demonstrate the idea by specializing FSTMERGE for the JAVA programming language.

Apel et al. [21] follow up on their previous work in [40] by performing a more robust, quantitative evaluation on JAVA, C# and PYTHON projects. They show that compared to unstructured merge, semistructured merge can significantly reduce merge conflict size and quantity overall. This is primarily attributed to semistructured merge being able to resolve so-called *ordering conflicts*. These occur when unstructured merge reports a merge conflict when for example different import statements or different method declarations have been inserted in the same textual area by different revisions. As the order of such constructs do not matter, semistructured merge can resolve the conflicts automatically by simply choosing an order in which to insert the changes. It is however noted that in some cases, such as when methods have been renamed,

[3]It should be noted that when using a DVCS such as GIT, all versions of all files are always available, making this reasoning irrelevant

semistructured merge results in more and larger conflicts than unstructured merge.

Cavalcanti, Accioly, and Borba [20] replicate the experiments by Apel et al. [21] on a significantly larger sample. Their results strengthen the findings of Apel et al. [21], and also confirm that many conflicts encountered in unstructured merge are ordering conflicts. In addition, Cavalcanti, Accioly, and Borba [20] discuss merge tool accuracy in terms of *false positives* and *false negatives*. A false positive occurs when a merge conflict is reported, but it does not actually represent a real conflict and could be solved automatically given enough information. Ordering conflicts are archetypical false positives. A false negative occurs when a merge tool resolves a merge in which the merged revisions actually contain incompatible changes. Duplicated methods across revisions of a JAVA file being merged with unstructured merge can for example give rise to false negatives if the duplicated methods are in different parts of the file.

Cavalcanti, Borba, and Accioly [11] expand upon their previous work in [20] by benchmarking semistructured merge on an even larger sample, as well as introducing the semistructured merge tool JFSTMERGE. It builds upon a more efficient version of FSTMERGE [21, 40], and also adds post-processing to deal with common problems for semistructured merge, such as renamed methods. The results of the benchmark again strengthen the findings of earlier works [20, 21] in that semistructured merge significantly reduces merge conflicts when compared to unstructured merge. False negatives are also more carefully examined than before, and are found to occur with roughly the same frequencies across semistructured and unstructured merge. Cavalcanti, Borba, and Accioly [11] do however argue that the false negatives caused by semistructured merge are often harder to resolve then the ones caused by unstructured merge.

Tavares et al. [10] use a modified version of FSTMERGE to create two slightly different merge tools for JAVASCRIPT, which are evaluated with the same methodology as in [11]. FSTMERGE is modified as the authors argue[4] that the original version does not adequately support certain characteristics of JAVASCRIPT, such as being able to mix ordered elements (e.g. statements) and unordered elements (e.g. function declarations) on the same syntactical level. They find that the best variation of their tool reduces merge conflicts when compared to unstructured merge. However, the reduction is not as dramatic as in previous works [11, 20, 21], and Tavares et al. [10] conclude that substantial improvements are required to make semistructured merge attractive for JAVASCRIPT developers.

A recent study by Cavalcanti et al. [19] compares structured and semistruc-

[4]Backed by an experiment on a single project

tured merge. It shows that structured merge results in fewer false positives than semistructured merge, but also results in more false negatives. The authors also argue that while semistructured merge does produce more false positives, these are easier to resolve than the additional false negatives of structured merge. Ultimately, they conclude that structured and semistructured merge provide similar accuracy in real-life scenarios, but semistructured merge does so at lower computational cost and implementation complexity [19].

Shen et al. [41] present INTELLIMERGE, a graph-based merge tool for JAVA. The graph used, called a *Program Element Graph* (PEG), represents the overall structure of the program. A node may denote for example a method or a field, and an edge between two nodes may denote a method invocation or a field read or write. Compared to a fully resolved CFG, the PEG only resolves the high-level elements of the code, and fine-grained elements such as method bodies are not resolved. This simplification is an important aspect of making INTELLIMERGE viable for practical use as it allows its running time to remain feasible, something that graph-based merging techniques typically fail to achieve [12, 41]. When the PEG has been constructed, nodes from different revisions are matched together, using GUMTREE to compute method body similarities. This matching is 1-to-1, and is used to detect for example moved and renamed methods and fields. A second n-to-m matching between method nodes is computed to find methods that have been extracted from or inlined into other methods. This step is specifically designed to ease merging of refactorings, which is one of the primary goals of INTELLIMERGE. The fact that a single node can be matched to multiple other nodes can help resolve situations where for example the common code of multiple methods has been extracted into a single helper method. The PEG is then merged structurally, but unresolved elements such as method bodies are merged textually, making it a semistructured approach. Finally, the merged PEG is pretty-printed into source code files. Shen et al. [41] also benchmark INTELLIMERGE, and find that it improves upon JFSTMERGE. In line with previous works [11, 20, 21], JFSTMERGE is in turn found to significantly improve upon unstructured merge. INTELLIMERGE is also notably faster than JFSTMERGE, with a median running time of around 539 ms, compared to 1426 ms for JFSTMERGE.

3.2 Merge conflict resolution helpers

Some authors focus their efforts on assisting in resolving conflicts when they do occur, rather than try to resolve them automatically. The most straightforward of such tools are visualizers that simply present the merge conflicts in

a more intuitive manner. These include tools like KDIFF3[5], MELD[6] and WINMERGE[7]. Some *integrated development environments* (IDE) also provide integrated merge conflict visualizers [42].

Nieminen [43] presents CORED, a tool and workflow for collaborative resolution of merge conflicts. The premise of the tool is that whenever developers encounter a conflict, they can upload the conflicting files to a server and edit the affected files in a collaborative online environment. Other developers can then be invited to a specific merge resolution session by supplying them with a unique URL. This addresses the problem of a single developer often not possessing all of the required knowledge to resolve a difficult conflict.

Costa et al. [44] present TIPMERGE, a tool that inspects the commit history of two branches and identifies experts who may be suitable to merge said branches. TIPMERGE is an appropriate complement to CORED, as it can automatically provide insight into which developers are most suited for performing the merge, and these developers can then collaborate in the CORED environment.

Nishimura and Maruyama [45] present MERGEHELPER, an interactive tool for resolving merge conflicts by replaying the edit operations performed on the conflicting branches. Among other things, it can detect which commits introduced conflicting changes, produce artificial merges between intermediary commits as well as replay the entire edit history of a branch. It does however have a few major limitations, one being that the edit history must be separately recorded by the ECLIPSE IDE, and another being an inability to detect renamings of methods.

Zhu and He [18] present AUTOMERGE, a merge resolution helper that is not tied in to any particular development environment. It is built on top of JDIME and acts just like a structured merge tool as long as no merge conflicts are encountered. When merge conflicts are encountered, it attempts to synthesize possible resolutions using version space algebra (VSA). When a number of candidate resolutions have been discovered, AUTOMERGE presents these to the developer, who can then choose an appropriate resolution.

Zhu, He, and Yu [46] improve upon AUTOMERGE by introducing a more advanced tree matching algorithm. While the original version of AUTOMERGE [18] reuses the entirety of JDIME's tree matching and merging, the improved version introduces a new matching quality function that is used to evaluate how well two nodes match. Zhu, He, and Yu [46] show that the improved tree matching yields significant improvements in a quantitative experiment.

[5] http://kdiff3.sourceforge.net/

[6] `https://meldmerge.org/`

[7] `https://winmerge.org/`

Sousa, Dillig, and Lahiri [47] present SAFEMERGE, a tool for verifying *semantic conflict-freedom* in a three-way program merge [47]. SAFEMERGE takes as input the two revisions that were merged, their shared ancestor, as well as the outcome of the merge, and attempts to establish a proof that the merge respects all semantic changes introduced by the merged revisions. SAFEMERGE thereby helps resolving semantic conflicts by alerting developers of their presence, and could possibly be combined with a structured merge tool to help counteract the false negatives that such tools are prone to exhibit. For this use case, SAFEMERGE is however limited by the fact that it in its current state cannot pinpoint the location of a semantic conflict, but only alert the user that a proof of semantic conflict-freedom cannot be produced. SAFEMERGE could also be useful in evaluating a merge tool, as it can verify semantic conflict-freedom in absence of a reference merge result. However, SAFEMERGE is itself validated on a comparatively small sample size of 52 merge scenarios from only 5 projects, which is insufficient to demonstrate its general applicability and precision.

3.3 Merge conflict prediction and workspace awareness

Workspace awareness tools attempt to make developers aware of their colleagues' parallel changes, especially if the changes could or will result in some form of conflict [6]. The core idea is that if conflicts are detected early, or even predicted before they manifest, they can be more easily resolved [48].

PALANTIR is an early example of a workspace awareness tool for JAVA projects, which continuously monitors commits on developers' local branches and tries to integrate them with each other [48]. It informs all developers concerned when they are working on the same or closely related pieces of code, and alerts developers when their changes conflict with each other. The tool is built on the idea that conflicts materialize slowly over many commits, and by informing developers as soon as a conflict materializes potential damage and hard work to resolve the conflict can be minimized. PALANTIR is tailored for SUBVERSION and requires ECLIPSE, making it a fairly locked-in piece of software.

Kasi and Sarma [6] follow up on the work on PALANTIR with a new tool, called CASSANDRA. As even conflicts that are detected early can be cumbersome to resolve, CASSANDRA goes a step further than PALANTIR by recommending an order in which tasks should be completed to avoid conflicts altogether. In essence, while PALANTIR alerts developers of conflicts at an early stage, CASSANDRA aims to help developers not encounter conflicts at all.

SYDE is another workspace awareness tool that functions similarly to PALANTIR and CASSANDRA, but detects merge conflicts on the AST level for JAVA programs [49]. That SYDE uses ASTs to represent the source code highlights the fact that improved diff and merge tools are complementary to workspace awareness tools.

CRYSTAL is similar to CASSANDRA, and tries to predict future conflicts by continuously integrating developers' changes [5]. Among other things, it tries to merge branches that developers are working on, as well as build and test these merges. The authors call this *speculative analysis*, as the actions performed by CRYSTAL (such as merging branch `A` into branch `B`) may not actually be performed by developers in the future. While comprehensive in scope, this is also computationally expensive, and the computational load increases rapidly with an increasing amount of branches.

WECODE is similar to CRYSTAL, and provides many of the same services [50]. It allows JAVA developers to join a *team*, which has an associated *merged system*. WECODE continuously merges individual team members' changes (both committed and uncommitted) into the merged system using structured merge, builds the merged system to detect build conflicts, and then runs JUNIT test cases to detect test conflicts. The ideas are generalizable, but the implementation is heavily tied into the JAVA programming language, the ECLIPSE IDE and the JUNIT testing framework. Like CRYSTAL, this is a comprehensive system that provides many services, but at a high computational cost.

A more recent trend is to do lightweight feature analysis in order to try to detect possible conflicts, rather than to actually try merging. The aim is aligned with that of tools such as CRYSTAL and WECODE, but with the added goal of reducing the computational cost of doing speculative analysis. It has however proven difficult to build accurate models, with previous works finding either little or no correlation between their chosen predictive factors and merge, build and test outcomes [51, 52]. There is evidence that better-than-chance predictors can be found [53, 54], but the idea of lightweight merge conflict prediction is clearly still in its early stages. Conflict-free merges do however appear to be possible to detect with both high precision and recall [54].

3.4 Properties of branching and merging

Several studies investigate the general properties of branching and merging. This section presents a selection of such studies.

Zimmermann [55] presents one of the first studies on version control history and merge conflict characteristics by mining the CVS repositories of 4 large projects. Their primary findings are that conflicts occur frequently, as often as in 46.62% of all merge scenarios for the project with the highest conflict frequency, and 22.75% for the project with the lowest conflict frequency.

Bird et al. [1] compare studying version history from a CVCS to doing the same for a DVCS, using SVN as an archetypical example of a CVCS and GIT as the corresponding DVCS. They highlight that in GIT, more information is typically available to researchers. For example, experimental and incomplete code is often committed to the repository, which is something that rarely occurs in SVN. In the latter, commits are usually only created once the code has been thoroughly evaluated. Therefore, version history in a DVCS potentially provides researchers with a more complete picture of the development process. There are however many drawbacks to be aware of, the most important of which is that the version history in GIT is more easily altered than in SVN, making it more likely to occur. For example, merges can be made invisible, branches can be deleted and commits can be altered, squashed together or removed entirely. Although GIT stores information about such history alterations, it only does so in the local repository where the change occurred. Bird et al. [1] also note that although branching is possible in SVN, it is significantly more common in GIT due to the process being more lightweight. They even claim that branching in GIT is essentially unavoidable due to the creation of implicit branches, as explained in Section 2.1.3.

Bird and Zimmermann [2] perform a survey study to evaluate the benefits and drawbacks of using branches in software development. They find that branching and merging accounts for 5.45 hours of development time per month on average across all survey respondents, but also that the 95th percentile averages more than 15 hours. This discrepancy is explained by some developers being assigned as integrators to deal with the majority of merge integration. It is also found that social communication is important to resolving complex merge conflicts, and that incorrectly resolved conflicts frequently cause errors at later stages. It is however noted that such errors often occur in non-source code files, such as XML files and build manifests. The authors conclude that branching carries a clear cost, and that some branches cost more than others, causing varying amounts of delays or even entirely halting development. They suggest that branches should be treated with care, and that activities such as refactoring and optimization of branches should occur frequently within larger projects.

McKee et al. [56] perform a survey study investigating how software de-

velopers experience merge conflicts. Among other things, they conclude that developers perceive the complexity of the conflicting lines of code along with knowledge of said code to be the two most influential factors in the perception of merge difficulty. Other factors also play a role, such as the amount of conflicting lines and number of files in the conflict. This initial perception of the conflict is important, as it affects how the conflicts are approached. McKee et al. [56] also note that current tooling does not fully meet developers' needs, such as being able to easily and intuitively explore revision history.

Nelson et al. [57] extend the work in [56] by further analyzing the results, and by conducting an additional survey investigating how merge conflicts are monitored for and handled by developers. They find that many developers do not monitor for merge conflicts at all, and of those that do, most use tools that can only alert them of conflicts after they have already occurred. They also suggest that workspace awareness tools such as PALANTIR [48] and CRYSTAL [5] are not used in practice, either due to the tools not being mature enough or the fact that developers simply are not aware of them. They also find that most developers interrupt their current work to investigate conflicts, regardless of if the conflict is caused by mismatching whitespace or overlapping logical changes. A common solution is however to defer conflicts, thus delaying their resolutions, which may impact the entire team and potentially even require a reimplementation of the changes.

Menezes et al. [58] study merge conflicts in open source JAVA projects. To find merge conflicts, they gather projects from GITHUB and replay all merge commits using GIT's default line-based merge driver. They first perform a small-scale manual analysis of all merge conflicts in 5 repositories, and then scale this up to an automated analysis of 2,731 repositories. This yields three primary findings. First, for 87% of all conflict hunks, no new code is written to resolve the conflict. Second, 60% of all merges with conflicts involve multiple conflict hunks, and dependencies between conflict hunks occur in 29% of all conflicting merges. Third, there are often patterns to how developers resolve conflicts, but the patterns vary from project to project, and even from developer to developer. Based on the variability observed in merge resolutions to similar conflicts, Menezes et al. [58] postulate that no single merge technique will ever be able to fully automate merging. They instead suggest that merge tools should be built around a base technique, and offer extensibility in the form of plugin systems to allow for tailoring of merge tools. Such plugins could for example offer heuristic solutions to types of merge conflicts common to a particular project.

Mahmoudi, Nadi, and Tsantalis [13] investigate the correlation between

code refactorings and the presence of merge conflicts. They argue that if certain code refactorings are to blame for a significant amount of merge conflicts, then more effort should be invested into developing tools that can resolve such conflicts. With an empirical evaluation, they investigate the prevalence of 15 out of 70 types of refactorings as classified by Fowler [59]. Only 15 types are investigated due to tooling limitations. They find that 22% of merge conflicts involve refactorings, and also that conflicts caused by refactorings are larger in general, and thereby more difficult to resolve. They also find that certain refactorings are more problematic than others. In particular, the `Extract Method`[8] refactoring is one of the most common causes of conflicts relative to its frequency of occurrence.

Owhadi-Kareshk and Nadi [60] present MERGANSER, a tool for extracting and replaying merge scenarios, as well as gathering metadata about the related projects. The authors recognize that many merge studies need to replay merges, and are therefore often accompanied by custom tools tailored for those particular evaluations. Such tooling is typically not easily reusable, motivating the creation of a reusable merge replay tool. MERGANSER can be instructed to collect or not collect certain data, such as build and test status, making it amenable to different kinds of studies. However, this configurability is limited to the built-in capabilities of the tool, and there is no system in place to extend these capabilities. MERGANSER is also limited to replaying entire merge scenarios, offering no support for replaying individual file merges. As such, MERGANSER is a suitable solution for investigating general properties of merges across different projects, but may not be sufficiently customizable for more detailed studies, such as merge tool evaluations.

There are a few key takeaways from this section that. First, it is evident that merge conflicts occur frequently and are non-trivial to deal with [1, 2, 55, 56, 57]. Second, resolution patterns for similar merge conflicts differ between teams and individual developers, showing a need for customizable merge tools [13]. Third, refactorings are a common cause for merge conflicts, calling for tools that can deal with the most conflict-prone refactorings [13]. Finally, there is a general lack of reusable tooling to perform merge studies, and development of such tooling has only recently gotten started [60].

[8] The extraction of part of a method into a method of its own

3.5 Benchmarking merge tools

There is no one standardized way of benchmarking merge tools, and different authors typically adopt different approaches. To ease reproducibility, many studies provide replication packages that include the data for the benchmarks, as well as scripts and tooling for analysis [10, 11, 19, 20, 21]. While the replication packages allow the experiments to be replicated, they are typically tailored to the tools they are designed to evaluate, and are often not easy to reuse for other studies [60]. A selection of such existing benchmarks are summarized in Table 3.1, each of which is presented in more detail in the rest of this section.

Apel et al. [21] perform one of the first merge tool benchmarks on 180 merge scenarios from 24 SUBVERSION projects sourced from SOURCEFORGE. In this benchmark, the authors primarily investigate merge conflicts in regards to quantity and size. An important quirk of this benchmark is caused by the fact that for a long time, SUBVERSION did not support any standard log entry format for merges. Thus, merge scenarios are identified manually, and some plausible merge scenarios are simply constructed. They conclude that semistructured merge significantly improves upon the capabilities of unstructured merge.

Cavalcanti, Accioly, and Borba [20] is a replication study of Apel et al. [21]. They use the same scripts and tools for evaluation, but the merge scenario mining step differs. Instead of sourcing SUBVERSION projects from SOURCEFORGE, GITHUB is used to find 60 GIT projects. From an initial selection of over 60,000 automatically identified merge scenarios, a subset of 4,678 scenarios are selected for analysis based on them causing merge conflicts with unstructured merge. This may be significant, as semistructured merge could produce merge conflicts where unstructured merge does not. The results reported by Cavalcanti, Accioly, and Borba [20] strengthen the findings of Apel et al. [21].

Leßenich, Apel, and Lengauer [7] benchmark their structured merge tool JDIME on 434 real merge scenarios from 50 JAVA projects. In this benchmark, both precision and running time is measured. The estimated precision of a tool is purely based on the amount of conflicts reported by it. The running time is measured for each scenario and tool just once. They find that structured merge produces significantly fewer merge conflicts than unstructured merge, but at the expense of being up to 109 times slower, and 23 times slower on average. They also find that running unstructured merge first, and switching over to structured merge only when conflicts appear improves performance by 10 times on average, and yields roughly the same conflict as when running purely structured merge.

Table 3.1: *Summary of studies with merge tool benchmarks*

Study	Scenarios	Languages	Replication pkg.
Apel et al. [21]	180	PYTHON, C#, JAVA	Yes[1]
Leßenich, Apel, and Lengauer [7]	434	JAVA	Yes[2]
Cavalcanti, Accioly, and Borba [20]	4,678	PYTHON, C#, JAVA	Yes[3]
Cavalcanti, Borba, and Accioly [11]	30,030	JAVA	Yes[4]
Cavalcanti et al. [19]	43,509	JAVA	Yes[5]
Tavares et al. [10]	10,345	JAVASCRIPT	Only results[6]
Zhu and He [18]	101	JAVA	Yes[7]
Zhu, He, and Yu [46]	3,687	JAVA	No
Shen et al. [41]	1,070	JAVA	Only dataset[8]

[1] `http://www.fosd.de/SSMerge/`
[2] `http://fosd.net/JDime`
[3] `https://twiki.cin.ufpe.br/twiki/bin/view/SPG/SemistructuredMergeReplication`
[4] `https://spgroup.github.io/s3m/eis.html`
[5] `https://spgroup.github.io/s3m/svj.html`
[6] `https://ase2019author.github.io`
[7] `https://thufv.github.io/automerge/docs/artifact/`
[8] `https://github.com/Symbolk/IntelliMerge-data`

Cavalcanti, Borba, and Accioly [11] perform a relative benchmarking of unstructured and semistructured merge on 34,030 real merge scenarios from 50 JAVA projects using GIT. These 50 projects are manually selected from the 100 most starred JAVA projects on GITHUB in an effort to find a diverse sample [61]. The benchmark is relative as only merge scenarios where the tools give different verdicts (i.e. one reports conflicts and the other does not) are analyzed. The authors argue that unstructured and semistructured merge are similar enough that this approach is sufficient to compare them to each other.

In a study setup similar to that of Cavalcanti, Borba, and Accioly [11], Cavalcanti et al. [19] compare semistructured and structured merge on 43,509 real merge scenarios from 508 JAVA projects on GITHUB. The authors report that the selected sample exhibits a wide range of diversity[9] in terms of, among other things, domain, size and amount of contributors. All of the selected projects use MAVEN, which allows for consistent building and testing, and TRAVIS CI, which allows the authors to check real build statuses. The latter allows for merge scenarios where one or more of the revisions in the merge do not build on their own to be filtered out, which could otherwise cause an overestimation of false negatives. As in their previous work [11], only merge scenarios where the tools differ are further analyzed.

Tavares et al. [10] perform a relative benchmarking of unstructured merge

[9] As defined by Nagappan, Zimmermann, and Bird [61]

and semistructured merge on 10,345 merge scenarios from 50 JAVASCRIPT projects. The study setup largely follows that of the previous work by Cavalcanti, Borba, and Accioly [11] and Cavalcanti et al. [19], and only scenarios in which the tools give different verdicts are analyzed.

Zhu and He [18] benchmark their tool AUTOMERGE on 101 merge scenarios from 10 different projects. From the merge scenarios, they find 138 file merges that cause a total of 244 conflicts for JDIME. They also identify the file that was actually committed by the developers, and refer to this as the *expected* resolution. The file merges are replayed with AUTOMERGE, which is forced to synthesize merge resolutions as JDIME encounters a conflict on each merge. The authors then assign a score for each file merge based on how many merge resolutions AUTOMERGE must synthesize before it finds one that matches the expected resolution according to JDIME's AST-based diff.

Zhu, He, and Yu [46] benchmark an improved version of AUTOMERGE that has a more involved tree-matching algorithm than JDIME. The benchmark is performed on 3,687 merge scenarios from JAVA projects collected from GITHUB, each merge being replayed by JDIME and AUTOMERGE. The merge results are then compared to the developers' actual merge with the goal of determining how well the tools reproduce the merges. The revisions are checked for equality using JDIME's AST-based diff[10], and are considered equal if there is no difference.

Shen et al. [41] benchmark their tool INTELLIMERGE, GIT's default unstructured merge and JFSTMERGE on 1,070 merge scenarios collected from JAVA projects on GITHUB. These merge scenarios are specifically selected to include refactorings, as determined by automatic analysis. The merge scenarios are replayed for each tool and the merge results are textually compared to the developers' actual merge commits, with the amount of differing lines acting as an estimation of merge correctness. The authors note that using the developers' actual merge commits as ground truth is a flawed approach, as they are sometimes objectively incorrect. It is also biased in favor of GIT's unstructured merge, as that is the tool that is in use by most projects. Merge conflicts are also evaluated in regards to the amount of conflict hunks, and the total amount of conflicting lines.

We do not reuse any of the discussed benchmarks in this thesis, due to them being too tailored to the specific studies. We do however reuse methodology and ideas, which is discussed in Section 6.3 and Section 6.4.

[10]This is not explicitly stated in the paper, but has been confirmed in e-mail conversation with the authors

Chapter 4

Primer on 3DM-merge

3DM is a diff and merge tool for XML documents, created by Lindholm [16]. The novel part about it is the merge algorithm, which is a generalized algorithm for ordered trees[1]. As 3DM is the name of the entire tool, we refer to the merge algorithm in [16] as 3DM-MERGE.

This chapter describes 3DM-MERGE in detail, and is structured as follows. Section 4.1 presents the data structure that 3DM-MERGE operates upon. Section 4.2 and Section 4.3 describe important theoretical concepts related to 3DM-MERGE, and Section 4.4 presents the merge algorithm itself.

Note that this chapter both contains terms and concepts as defined by Lindholm [16], and as defined by the authors of this thesis. Anything stated as fact is as defined by Lindholm [16]. Our own definitions amount mostly to extended terminology, and are always prefaced with a "we" or "our".

4.1 PCS triples, content tuples and the change set

3DM-MERGE operates on a data structure that Lindholm [16] refers to as a *change set*. The change set is composed of two primary data types, both of which are detailed in this section.

[1] A tree in which the child list of each node is an ordered list of nodes, rather than an unordered set of nodes

4.1.1 PCS triples

The first data type is the *parent-child-successor* (PCS) triple, which is used to encode the structure of a tree. A PCS triple is written on the form $pcs(parent, child, successor)$, where $parent$ is the parent node of both $child$ and $successor$, and $successor$ immediately follows $child$ in $parent$'s child list. To make the terminology more clear, we refer to the $child$ node of the triple as the $predecessor$, as it precedes the successor.

Intuitively, all PCS triples with $parent = p$ encode the child list of p. With this representation, the structure of a tree with n nodes can be encoded as a set of $O(n)$ PCS triples. We refer to the nodes in PCS triples that have a corresponding node in the real tree as *concrete* nodes. There are also three kinds of virtual nodes: a virtual root ($\perp$), a virtual start of a child list ($\dashv$) and a virtual end of a child list ($\vdash$). For reasons explained in Section 4.1.3, the $\dashv$ and $\vdash$ virtual nodes are uniquely defined for each child list. That is to say, the $\dashv$ nodes in $pcs(p, \dashv, \cdots)$ and $pcs(p', \dashv, \cdots)$ are not one and the same, and can be thought of as $\dashv_p$ and $\dashv_{p'}$, respectively.

The PCS concept is best explained by example. Consider the tree in Figure 4.1. It can be converted to a PCS set by creating a PCS child list for each node. For example, the child list for node c would consist of the two triples $pcs(c, \dashv, d)$ and $pcs(c, d, \vdash)$. The entire set of PCS triples can be seen in Table 4.1.

Note how a PCS triple contains just enough information to connect it to its surrounding triples. For example, $pcs(a, \dashv, b)$ has a as parent and b as successor. The next PCS in a's child list still has a as a parent, but b as a predecessor

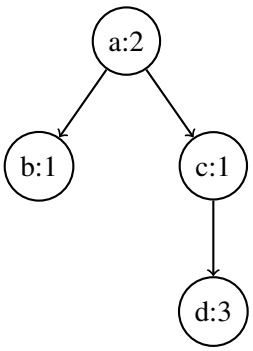


Figure 4.1: *Schematic drawing of an ordered tree. The labels should be interpreted as* `<ID>:<CONTENT>`

.

Table 4.1: *PCS set of Figure 4.1, ordered into child lists for clarity*

Node	PCS child list
$\perp$	$pcs(\perp, \dashv, a),\ pcs(\perp, a, \vdash)$
a	$pcs(a, \dashv, b),\ pcs(a, b, c),\ pcs(a, c, \vdash)$
b	$pcs(b, \dashv, \vdash)$
c	$pcs(c, \dashv, d),\ pcs(c, d, \vdash)$
d	$pcs(d, \dashv, \vdash)$

Table 4.2: *Content set of the tree in Figure 4.1*

$$c(a, 2),\ c(b, 1),\ c(c, 1),\ c(d, 3)$$

instead of successor, meaning $pcs(a, b, c)$. By the same reasoning, the next PCS is $pcs(a, c, \dashv)$, which marks the end of a's child list. Of course, one can apply the logic in reverse to traverse the child list in the opposite direction. It is theoretically similar to a doubly linked list.

As the virtual nodes for each concrete node are known, starting to traverse an arbitrary node x's child list is a matter of finding the PCS triple where x is the parent, and $\dashv$ is the predecessor. For example, traversal of the PCS set can be started by finding the triple $pcs(\perp, \dashv, \cdots)$, and traversal of the c node's child list can be started by finding the triple $pcs(c, \dashv, \cdots)$. Combined with the linked list-like child list traversal, this gives a method for performing a top down traversal of the tree based entirely on the PCS triples.

4.1.2 Content tuples

The second data type is the *content* tuple. A node v with content m is represented as the tuple $c(v, m)$. Table 4.2 shows the content set of Figure 4.1. What the content of a tree node actually represents is a matter of implementation detail. In general, if a value x is associated with a node v, but x is itself not a proper tree node, then x is part of v's content, i.e. $x \in m$. Note that m should be interpreted as a multiset of values, rather than as a single value.

As mentioned at the beginning of this section, the PCS triples and content tuples that encode a tree T are collectively known as the change set of T. The change set of the tree in Figure 4.1 is therefore the union of the PCS triples in

Table 4.1 and the content tuples in Table 4.2.

4.1.3 Consistency

An important concept related to change sets is *consistency* [16]. A change set is consistent if every node v fulfils all of the following criteria.

1. v has at most one parent x. We call this the *unique parent criterion*.
2. v has at most one predecessor y. We call this the *unique predecessor criterion*.
3. v has at most one successor z. We call this the *unique successor criterion*.
4. v has at most one piece of associated content m. We call this the *unique content criterion*.

A corollary of the unique parent criterion is that in any PCS triple where v appears as either predecessor or successor, x must be the parent. Combined with the unique predecessor criterion, this implies that v may only appear as a successor once, in a PCS triple on the form $pcs(x, y, v)$. A corollary of a combination of the unique parent and successor criteria is that v may only appear as a predecessor once, in a PCS triple on the form $pcs(x, v, z)$. Note that a node can appear as a parent in an arbitrary number of PCS triples without violating any of the criteria, meaning that child lists are unbounded in size.

Given these criteria, it is evident that the virtual $\dashv$ and $\vdash$ nodes marking the beginning and end of a child list must be unique for each child list. Otherwise, two unrelated child lists $[pcs(p, \dashv, a), pcs(p, a, \vdash)]$ and $[pcs(p', \dashv, a'), pcs(p', a', \vdash)]$ would violate the unique successor criterion in regards to $\dashv$, and the unique predecessor criterion in regards to $\vdash$.

4.1.4 Well-formedness

The consistency criteria presented in Section 4.1.3 guarantee that any change set that fulfills them is unambiguous [16]. We argue that this is insufficient to ensure that the change set actually encodes a valid tree. For example, the change set $\{pcs(a, b, \vdash), c(a, m)\}$ is consistent, but the child list of a is clearly incomplete, having no predecessor to b and no start of the list. Additionally, a has no parent, and b has neither content nor a child list. This change set is ill-formed, and it is not be possible to build a tree from it. The consistency

criteria allow this as they only set upper limits on any given node's presence in change set elements, but fail to declare lower limits.

We define a *well-formed* change set as a subset of consistent change sets by imposing such lower limits. First, each concrete node v must fulfill the following criteria.

1. v has precisely one parent x
2. v has precisely one predecessor y
3. v has precisely one successor z
4. v has precisely one piece of associated content m
5. v appears in precisely one triple on the form $pcs(v, \dashv, \cdots)$
6. v appears in precisely one triple on the form $pcs(v, \cdots, \vdash)$

This ensures that each concrete node has content and that there are no incomplete or disjoint child lists. To ensure that the tree has a well-defined root, criteria 5 and 6 also apply to the $\perp$ node.

We also restrict virtual nodes such that they can only appear where they make sense. In other words, $\perp$ may only appear as a parent, $\dashv$ may only appear as a predecessor, and $\vdash$ may only appear as a successor. Although these criteria on virtual nodes are implicit in their definitions, we state them explicitly for added clarity. A change set that fulfills both the criteria for concrete nodes and the criteria for virtual nodes is well-formed, and a well-formed change set can always be interpreted into a tree.

4.2 Matchings and class representatives

3DM-MERGE makes use of a matching between tree nodes to determine where trees to be merged are similar [16]. The matching is used to group nodes across revisions that are considered to be the same nodes. Each node is mapped to a so-called *class representative*, and all nodes assigned to the same class representative are considered equivalent by 3DM-MERGE.

Consider the base, left and right revisions in a merge as trees T_0, T_1 and T_2, respectively. Assume that every node in a given tree is always matched to itself, and to at most one node in each of the other trees. The class representative of a node $v \in T_i$ is then the matching node in tree T_j with the smallest possible j. As every node is matched to itself, it is given that $j \leq i$. More formally, a

node $v \in T_i$ has class representative $w \in T_j$ if the following three criteria are met: 1) v is matched to w, 2) $j \leq i$ and 3) there is no other node $u \in T_k$ where v is matched to u and $k < j$.

Figure 4.2 shows an example three-way merge scenario with pairwise matchings between all three revisions. The corresponding class representatives mapping is shown in Table 4.3. Note how matchings to the base revision take precedence over matchings between the left and right revisions, as prescribed by criterion 3. For example, a_1 and a_2 are matched, yet the class representative for both is a_0. Mappings to the base revision represent the absence of change, and so the more that can be mapped to the base revision, the less work the merge algorithm has to perform.

4.3 Inconsistencies and conflicts

3DM-MERGE can automatically detect four forms of inconsistencies in the change set, one for each distinct violation of the consistency criteria presented in Section 4.1.3. We refer to inconsistencies by the criteria they violate.

1. A *parent inconsistency* violates the unique parent criterion
2. A *predecessor inconsistency* violates the unique predecessor criterion
3. A *successor inconsistency* violates the unique successor criterion
4. A *content inconsistency* violates the unique content criterion

We collectively refer to parent, predecessor and successor inconsistencies as *structural* inconsistencies. Such inconsistencies can only occur among PCS triples. Content inconsistencies can only occur among content tuples. Structural inconsistencies are more intricate to deal with, and are therefore the focus of the further discussion.

It should be noted that some criteria can be violated simultaneously by a single pair of PCS triples. For example, the two triples $pcs(x, y, z)$ and $pcs(x', y', z)$ present both a parent inconsistency by z having parents x and x', and a predecessor inconsistency by z having predecessors y and y'.

Inconsistencies are encountered in any non-trivial merge, as any difference between the base revision and the current revisions results in an inconsistency. For example, consider in isolation the child lists of the root nodes of the trees in Figure 4.2. Collapsing the full PCS triples on the form $pcs(a_0, x, y)$ to just (x, y) to reduce verbosity yields the child lists shown in Table 4.4. Note how

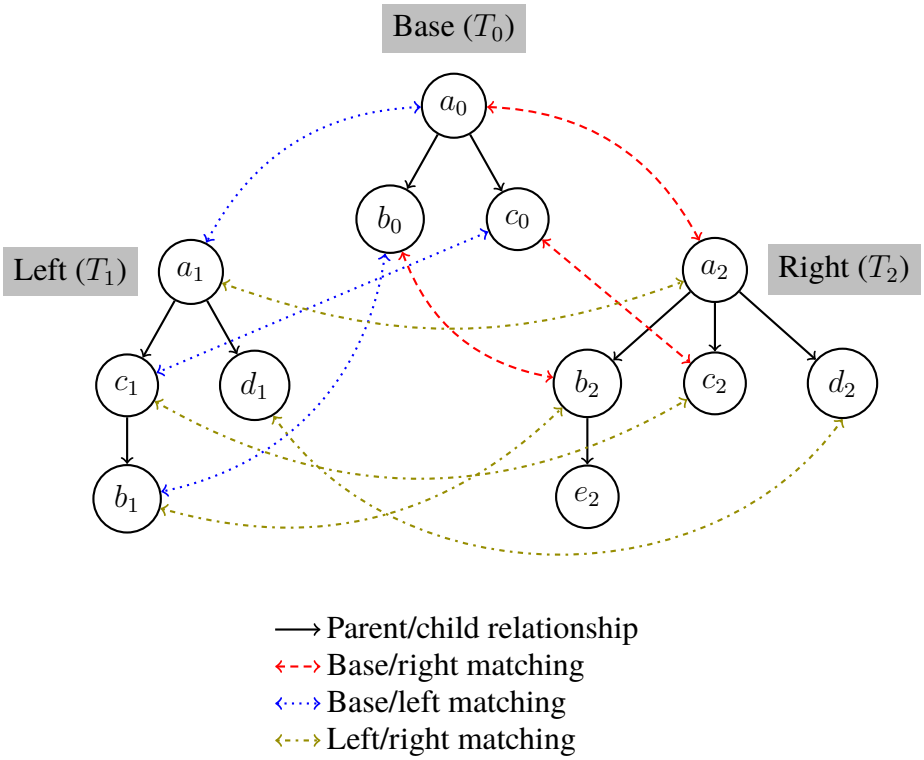


Figure 4.2: *Schematic drawing of a pairwise tree matching between the base, left and right revisions of a tree. The subscript of a node label indicates which tree the node belongs to.*

Table 4.3: *The class representatives mapping for the nodes in Figure 4.2*

Node	a_0	b_0	c_0	a_1	b_1	c_1	d_1	a_2	b_2	c_2	d_2	e_2
Class representative	a_0	b_0	c_0	a_0	b_0	c_0	d_1	a_0	b_0	c_0	d_1	e_2

Table 4.4: *Child lists of the root nodes of the trees in Figure 4.2, as well as of the expected merge tree. All nodes are presented as their class representatives, and $pcs(a_0, x, y)$ is collapsed into (x, y) for brevity.*

Revision	Simplified PCS child list of root node a_0
Left	$(\dashv, c_0), (c_0, d_1), (d_1, \vdash)$
Base	$(\dashv, b_0), (b_0, c_0), (c_0, \vdash)$
Right	$(\dashv, b_0), (b_0, c_0), (c_0, d_1), (d_1, \vdash)$
Merge	$(\dashv, c_0), (c_0, d_1), (d_1, \vdash)$

the individual child lists are consistent, but there are several inconsistencies across child lists.

Recall from Section 2.1.4 that a merge should include all changes made to the base revision. The merged child list should therefore preserve both the deletion of the b node in the left revision, and the concurrent addition of the d node in both the left and right revisions. Such a merge can be achieved by considering all inconsistencies, and removing any inconsistent elements that are present in the base revision. For example, $(\dashv, c_0)$ from the left revision is inconsistent with $(\dashv, b_0)$ from the right and base revisions. As $(\dashv, b_0)$ is present in the base revision, it is discarded, thus preserving the edit in the left revision. We refer to any inconsistency where one element is present in the base revision as a *soft* inconsistency. Soft inconsistencies can always be resolved by discarding the element from the base revision. This is the fundamental principle upon which 3DM-MERGE operates, and applying it to the remaining inconsistencies results in the merged child list seen at the bottom of Table 4.4. The merge algorithm is explained in more detail in Section 4.4, and is to a large degree a systematic approach to finding and resolving soft inconsistencies.

If none of the conflicting elements involved in an inconsistency are present in the base revision, then the inconsistency is caused by conflicting changes between the left and right revisions. We refer to this as a *hard* inconsistency. Consider for example if d_1 in the left tree was not matched to d_2 in the right tree in Figure 4.2. This would mean that the left and right revisions had inserted different nodes in the same place, which would then give rise to two inconsistencies, one between (c_0, d_1) in left and (c_0, d_2) in right, as well as between $(d_1, \vdash)$ in left and $(d_2, \vdash)$ in right. None of these elements are present in the base revision. These are hard inconsistencies, as the elements dictate different successors to c_0, and different predecessors to $\vdash$. Removing any one element in a hard inconsistency leads to a loss of information on changes applied to the

left or right revision.

We refer to the underlying operations that cause a hard inconsistency as a *conflict*. Lindholm [16] defines six conflict types. Each conflict is named on the form *x/y conflict*, which means that operation x in one revision interferes with operation y in another revision. Four of these conflicts are related to the positioning of nodes, namely the move/move, move/insert, move/delete and delete/delete conflicts. We refer to these as *structural* conflicts. Another conflict is the update/update conflict, which is equivalent to a hard content inconsistency, and hence we call it a *content* conflict. The final conflict type is the delete/edit conflict, which occurs when a node is deleted in one revision, but edited in the other, either in terms of the node's content or by way of structural or content changes in its subtree. The delete/edit conflict is unique in that it is the only conflict type in which a structural change may interfere with a content change. In addition to the conflicts defined by Lindholm [16], we define two more structural conflicts following the same naming scheme, namely the insert/insert and insert/delete conflicts.

Identifying which type of conflict gave rise to a hard inconsistency is necessary in order to determine how to solve or represent the merge conflict. The difficulty lies in the fact that in isolation, structural inconsistencies are often ambiguous. For example, the predecessor inconsistency between $pcs(x, y, z)$ and $pcs(x, y', z)$ could have been caused by the left revision inserting y ahead of z, and the right revision inserting y' in the same place, making it an insert/insert conflict. But it could also have been caused by the left revision deleting what was previously ahead of z, and the right revision inserting y', which would make it an insert/delete conflict. It could also have been caused by the left and right revisions both deleting nodes ahead of z, but differing amounts, making it a delete/delete conflict. There is simply not enough information embedded in the single inconsistency to determine which of these conflict types, if any, was the cause. Additionally, hard inconsistencies rarely come alone, and a single conflict often results in several hard inconsistencies. Conflicts in relation to SPORK are discussed in more detail in Section 5.6.

4.4 The merge algorithm

The 3DM-MERGE algorithm as presented by Lindholm [16] can be seen in Listing 4.1. The syntax is slightly altered to better suit the style of the pseudocode in the rest of this thesis, but the algorithm is the same.

3DM-MERGE consists of a series of steps. First, it converts the trees to be merged into change sets, replacing all nodes with their class representatives

(lines 2-4). Second, it forms a *"raw"* merge from the union of all change sets (line 7). While the change sets are always consistent individually, the raw merge is only consistent if all trees are identical, as discussed in Section 4.3.

The final stage of 3DM-MERGE, and the core of the algorithm, consists of iteratively transforming the raw merge into a consistent change set (lines 9-25). It does so by iterating through each element of the raw merge and removing soft inconsistencies according to the principles discussed in Section 4.3. If a hard inconsistency is encountered, 3DM-MERGE calls the conflict function, which Lindholm [16] does not discuss in detail. Our interpretation is that it registers the hard inconsistency between the two elements.

Listing 4.1: *The 3DM-MERGE algorithm by Lindholm [16]*

```
1  def merge(base, left, right: Tree, m: Matching):
2      base_change_set = class_rep_change_set(base, m)
3      left_change_set = class_rep_change_set(left, m)
4      right_change_set = class_rep_change_set(right, m)
5
6      # Δ is the initially "raw" merge,
7      # the set union of all change sets
8      Δ = base_change_set + left_change_set +
           right_change_set
9
10     for δ in Δ:
11         δ' = None # holder for inconsistent changes
12         δ' = get_other_content(Δ, δ)
13         if δ' == None:
14             δ' = get_other_root(Δ, δ)
15         if δ' == None:
16             δ' = get_other_predecessor(Δ, δ)
17         if δ' == None:
18             δ' = get_other_successor(Δ, δ)
19
20         if δ' != None:
21             if δ' in base_change_set:
22                 Δ = Δ - {δ'}
23             elif δ in base_change_set:
24                 Δ = Δ - {δ}
25             else:
26                 conflict(δ, δ')
27
28     return Δ
```

To illustrate how 3DM-MERGE operates, we present an example dry run of the loop using the merge scenario from Figure 4.2. As a simplification, we omit content tuples, meaning that the change set of a tree becomes equivalent to its

Table 4.5: *PCS sets from the tree revisions in Figure 4.2. Nodes are replaced by their class representatives, and the child list of each node is written on a line of its own to aid readability.*

Revision	PCS set
Left	$(\bot, \dashv, a_0), (\bot, a_0, \vdash),$ $(a_0, \dashv, c_0), (a_0, c_0, d_1), (a_0, d_1, \vdash),$ $(c_0, \dashv, b_0), (c_0, b_0, \vdash),$ $(d_1, \dashv, \vdash)$
Base	$(\bot, \dashv, a_0), (\bot, a_0, \vdash),$ $(a_0, \dashv, b_0), (a_0, b_0, c_0), (a_0, c_0, \vdash),$ $(b_0, \dashv, \vdash),$ $(c_0, \dashv, \vdash)$
Right	$(\bot, \dashv, a_0), (\bot, a_0, \vdash),$ $(a_0, \dashv, c_0), (a_0, c_0, d_1), (a_0, d_1, \vdash),$ $(b_0, \dashv, e_2), (b_0, e_2, \vdash),$ $(c_0, \dashv, b_0), (c_0, b_0, \vdash),$ $(d_1, \dashv, \vdash),$ $(e_2, \dashv, \vdash)$

PCS set. Table 4.5 shows the individual PCS sets of the tree revisions after the nodes have been mapped to their class representatives as per the mapping in Table 4.3. Table 4.6 illustrates the iterations of the primary loop, and can be found at the end of this section. The merged result can then be interpreted into the tree shown in Figure 4.3 by traversing each non-empty PCS child lists[2] top down, starting with the $\bot$ node's child list. So long as the change set is well formed, it is always possible to build the merged tree.

There are several details to make note of in Table 4.6. First of all, the initial raw merge seen in the very first segment is not much larger than the individual change sets. This is due to the fact that the revisions have a lot in common, and so the mapping to class representatives causes many of the PCS triples across revisions to become identical. Therefore, merging the parts of the trees that are the same across revisions is trivially handled by the initial set union that creates the raw merge.

The example also shows how a move of a node does not interfere with operations on that same node's child list. In this case, b is moved in the left

[2]Leaf nodes have child lists on the form $pcs(leaf, \dashv, \vdash)$, which may be ignored when building the tree

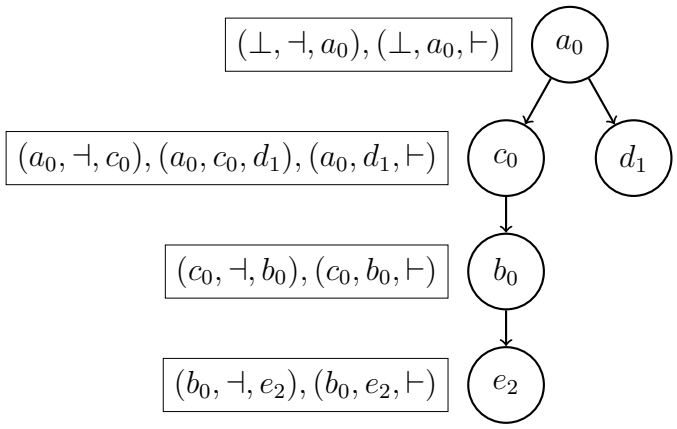


Figure 4.3: *Tree built from the merged change set of Table 4.6. Each child list is flanked by the corresponding set of PCS triples. Note that the leaf node child lists* $(d_1, \dashv, \vdash)$ *and* $(e_2, \dashv, \vdash)$ *are not explicitly illustrated, as they are empty by definition.*

revision, while the right revision inserts a new child node e. As a single PCS triple is only aware of its immediate parent, and not of any more distant ancestors, there is no hard inconsistency related to these two operations.

Another important consideration is that the same PCS triple is often involved in multiple inconsistencies, as illustrated by the two parent inconsistencies that $(c_0, \dashv, b_0)$ is involved in. Furthermore, the iteration order is undefined, and the iteration in Table 4.6 occurs in no particular order. The triples in Table 4.6 are ordered for the sake of readability, but the change set is not ordered. The undefined iteration order and PCS triples involved in multiple inconsistencies can in combination cause non-deterministic behavior in 3DM-MERGE, which is explained in Section 5.6.

Table 4.6: *Dry run of 3DM-MERGE on the merge scenario in Figure 4.2. The column headers correspond to the symbols used in Listing 4.1. Each rule-separated segment represents one iteration of the primary loop. Iteration over Δ occurs in no particular order, and iterations where no inconsistency is found are not shown. The Δ column reflects the state of Δ before δ or δ' in the same segment is removed.*

δ	δ'	Δ	Comment
$(a_0, \dashv, b_0)$	$(c_0, \dashv, b_0)$	$(\bot, \dashv, a_0), (\bot, a_0, \vdash),$ $(a_0, \dashv, b_0), (a_0, b_0, c_0), (a_0, c_0, \vdash),$ $(a_0, \dashv, c_0), (a_0, c_0, d_1), (a_0, d_1, \vdash),$ $(b_0, \dashv, \vdash), (b_0, \dashv, e_2), (b_0, e_2, \vdash),$ $(c_0, \dashv, \vdash), (c_0, \dashv, b_0), (c_0, b_0, \vdash),$ $(d_1, \dashv, \vdash),$ $(e_2, \dashv, \vdash)$	$\delta \in base$
(a_0, b_0, c_0)	$(c_0, \dashv, b_0)$	$(\bot, \dashv, a_0), (\bot, a_0, \vdash),$ $(a_0, b_0, c_0), (a_0, c_0, \vdash),$ $(a_0, \dashv, c_0), (a_0, c_0, d_1), (a_0, d_1, \vdash),$ $(b_0, \dashv, \vdash), (b_0, \dashv, e_2), (b_0, e_2, \vdash),$ $(c_0, \dashv, \vdash), (c_0, \dashv, b_0), (c_0, b_0, \vdash),$ $(d_1, \dashv, \vdash),$ $(e_2, \dashv, \vdash)$	$\delta \in base$
$(a_0, d_1, \vdash)$	$(a_0, c_0, \vdash)$	$(\bot, \dashv, a_0), (\bot, a_0, \vdash),$ $(a_0, c_0, \vdash),$ $(a_0, \dashv, c_0), (a_0, c_0, d_1), (a_0, d_1, \vdash),$ $(b_0, \dashv, \vdash), (b_0, \dashv, e_2), (b_0, e_2, \vdash),$ $(c_0, \dashv, \vdash), (c_0, \dashv, b_0), (c_0, b_0, \vdash),$ $(d_1, \dashv, \vdash),$ $(e_2, \dashv, \vdash)$	$\delta' \in base$
$(b_0, \dashv, e_2)$	$(b_0, \dashv, \vdash)$	$(\bot, \dashv, a_0), (\bot, a_0, \vdash),$ $(a_0, \dashv, c_0), (a_0, c_0, d_1), (a_0, d_1, \vdash),$ $(b_0, \dashv, \vdash), (b_0, \dashv, e_2), (b_0, e_2, \vdash),$ $(c_0, \dashv, \vdash), (c_0, \dashv, b_0), (c_0, b_0, \vdash),$ $(d_1, \dashv, \vdash),$ $(e_2, \dashv, \vdash)$	$\delta' \in base$
$(c_0, \dashv, \vdash)$	$(c_0, b_0, \vdash)$	$(\bot, \dashv, a_0), (\bot, a_0, \vdash),$ $(a_0, \dashv, c_0), (a_0, c_0, d_1), (a_0, d_1, \vdash),$ $(b_0, \dashv, e_2), (b_0, e_2, \vdash),$ $(c_0, \dashv, \vdash), (c_0, \dashv, b_0), (c_0, b_0, \vdash),$ $(d_1, \dashv, \vdash),$ $(e_2, \dashv, \vdash)$	$\delta \in base$
		$(\bot, \dashv, a_0), (\bot, a_0, \vdash),$ $(a_0, \dashv, c_0), (a_0, c_0, d_1), (a_0, d_1, \vdash),$ $(b_0, \dashv, e_2), (b_0, e_2, \vdash),$ $(c_0, \dashv, b_0), (c_0, b_0, \vdash),$ $(d_1, \dashv, \vdash),$ $(e_2, \dashv, \vdash)$	Finished merge

Chapter 5

Spork

This chapter outlines the architecture of SPORK along with its most important behaviors. Section 5.1 through Section 5.5 describe the 5 distinct phases by which SPORK merges JAVA files. Figure 5.1 contains an illustration of the 5 phases, showing how data flows between them and where the SPOON and GUMTREE libraries are most heavily utilized. The phases are as follows.

1. Parse each JAVA file to a SPOON AST as described in Section 5.1
2. Compute tree matchings as described in Section 5.2
3. Compute the merge as described in Section 5.3
4. Build a merged SPOON AST as described in Section 5.4
5. Pretty-print the SPOON AST as described in Section 5.5

Section 5.6 then provides a detailed description of how conflicts are expressed in SPORK's implementation of 3DM-MERGE, and how they are handled. Section 5.7 closes the chapter with a description of SPORK's fallback mechanisms that activate when there are failures in the merge process.

5.1 Parsing

SPORK uses the SPOON library [17] to parse JAVA source files into an ASTs. At this point, SPORK also identifies the style of indentation used in the source file. It attempts to detect both the indentation character, as either tab or space, and the amount of indentation. This information is required in order to pretty-print with the correct indentation.

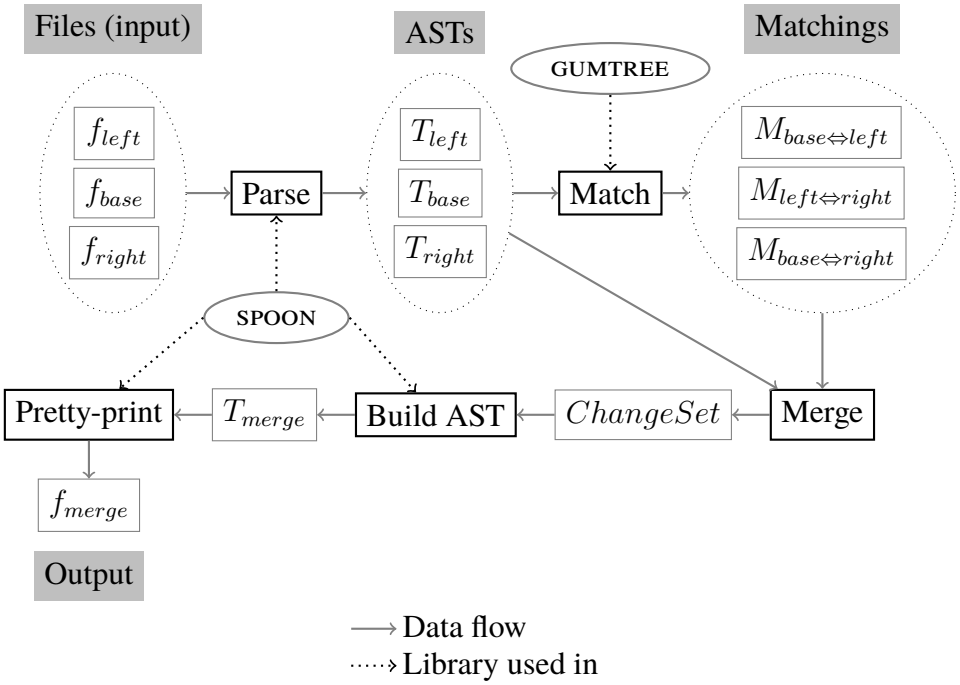


Figure 5.1: *Schematic drawing of SPORK's architecture. Thin-lined rectangles represent data, filled rectangles are labels describing the closest data, and related data is grouped within dotted outlines. Thick-lined rectangles represent phases and ellipses represent libraries and.*

The SPOON AST is then provided to GUMTREE-SPOON[1], which builds the corresponding GUMTREE AST. Each node in the GUMTREE AST has a reference to the corresponding node in the SPOON AST. This is important in the tree matching step, which is described next.

5.2 Tree matching

GUMTREE[2], which is discussed in Section 3.1.1, is used to compute pairwise tree matchings between the base, left and right revisions. In other words, it links nodes that are considered similar across revisions, as explained in Section 2.2.2 and Section 2.3.2. The actual pairwise matchings that are computed are conceptually the same as those shown in Figure 4.2.

The base-to-left and base-to-right matchings allow SPORK to align the two changed revisions with the base, and the left-to-right matching is primarily used to detect identical or near-identical additions. This approach is based both on the theory of file merging as presented in Section 2.3, and on the work by Leßenich [9] on JDIME.

The matchings computed by GUMTREE are based on its own AST model, and SPORK converts these to matchings between the original SPOON ASTs before proceeding with the merge. This makes it is easier to later build the merged SPOON AST, as opposed to merging GUMTREE nodes and then interpreting these to a SPOON AST.

Certain matchings that GUMTREE computes cause issues for 3DM-MERGE. 3DM-MERGE has no notion of valid parent-child relationships, all it knows about syntactical structure is what is inherent in the PCS triples. If two nodes are matched, then 3DM-MERGE merges both their contents and child lists, which is problematic if the two nodes do not accept precisely the same types of content or children. For example, an array type reference can be matched to a non-array type reference. The former has a feature that the other one lacks, namely the dimension of the array, which in SPOON is modeled as nested array type references. If this match is allowed, SPORK may end up trying to add the nested array type references to the normal type reference, which results in a crash. To remedy this, SPORK ignores such matchings when converting the GUMTREE matchings to SPOON matchings.

[1] `https://github.com/spoonlabs/gumtree-spoon-ast-diff`
[2] `https://github.com/gumtreediff/gumtree`

5.3 Merging

Merging is the primary technical contribution in SPORK, as most of the functionality is implemented directly in the tool itself. The implementation is based on 3DM-MERGE, which is described in Chapter 4.

5.3.1 Mapping to class representatives

The mapping to class representatives largely follows the theoretical ideas presented in Section 4.2. First, the nodes from the base revision are mapped to themselves, following the definition of a class representative. The nodes in the left and right revisions are then mapped to the base revision if there is a corresponding matching in the base-to-left or base-to-right matchings, or to themselves if there is no such matching.

SPORK does not necessarily add all of the node matchings in the left-to-right matching to the class representatives mapping. The left-to-right matching is prone to contain spurious matchings, and carelessly adding these to the class representatives mapping can cause unnecessary conflicts. Figure 5.2 shows an example merge scenario where such a spurious match is likely. In this case, GUMTREE can likely match the parameter c in the left revision's add function to the parameter c in the right revision's mul function. This causes a conflict in the merge[3], which is undesirable as the additions are clearly unrelated.

```
1 def add(a, b, c):
2     return a + b + c
3
4 def mul(a, b):
5     return a * b
```

***(a)** Left revision*

```
1 def add(a, b):
2     return a + b
3
4 def mul(a, b):
5     return a * b
```

***(b)** Base revision*

```
1 def add(a, b):
2     return a + b
3
4 def mul(a, b, c):
5     return a * b * c
```

***(c)** Right revision*

***Figure 5.2:** Merge scenario that can lead to spurious left-to-right matches*

To address this problem, SPORK implements two heuristics to decide whether or not to add a match between the nodes v_{left} in the left revision and v_{right} in the right revision to the class representatives mapping. First, none of v_{left} and v_{right} may be matched to a node v_{base} in the base revision, as a matching to the base should take precedence[4]. Second, the parents of v_{left} and v_{right} must

[3]Due to reasons explained in Section 5.7 this particular conflict is ignored by SPORK, but at a high computational cost

[4]The definition of class representatives actually allows a node v_{right} to be mapped to a node v_{left}, even if v_{left} is mapped to a node v_{base}

already be mapped to the same class representative. This prevents a matching such as the one discussed in regards to Figure 5.2 from making its way into the class representatives mapping, as the added parameters have different parent nodes.

Adding eligible left-to-right matches to the class representatives is performed with a top-down scan of the left tree, which is shown in the pseudocode in Listing 5.1. The fact that the scan is top-down is important, as it allows arbitrarily complex subtrees to be incrementally mapped as long as their roots have parents that are already mapped to each other.

Listing 5.1: *Pseudocode for augmenting the class representatives mapping with left-to-right tree matches*

```
1  def augment_mapping(
2          left_tree: Tree,
3          class_reps: Map[Tree, Tree],
4  ):
5      for left_node in top_down_scan(left_tree):
6          if class_reps[left_node] != left_node:
7              return # this node is already mapped to base
8          if left_node not in left_right_match:
9              return # this node is not matched to right
10
11         right_node = left_right_match[left_node]
12         if class_reps[right_node] != right_node:
13             return # the matched right node is already
                   mapped to base
14
15         if class_reps[left_node.parent] == class_reps[
               right_node.parent]:
16             class_reps[right_node] = left_node
```

5.3.2 Converting an AST to a change set

Converting a tree into a change set can be done by traversing the tree and creating a PCS child list for each node, as well as extracting each node's content. There are no requirements on the traversal order, but the relative ordering of predecessors and successors in the PCS triples must reflect the actual order of any given child list, as described in Section 4.1. Pseudocode for building a change set from a tree is presented in Listing 5.2.

It is worth noting that PCS triples and content tuples in SPORK are straight adaptations of their theoretical counterparts presented in Section 4.1. A PCS triple really is a triple of nodes, and a content tuple really is a tuple of a node

and a set of content. Both data types also carry some additional metadata, most importantly which revision they were created from.

Listing 5.2: *Pseudocode for building a change set from a tree*

```
1  def tree_to_pcs(tree: Tree):
2      content_set = {}
3      pcs_set = {
4          Pcs(⊥, ⊣, tree.root),
5          Pcs(⊥, tree.root, ⊢),
6      }
7
8      for parent in traverse(tree):
9          content_set += Content(parent, parent.content)
10         pred = ⊣
11         for succ in parent.get_children():
12             pcs_set += Pcs(parent, pred, succ)
13             pred = succ
14
15         pcs_set += Pcs(parent, pred, ⊢)
16
17     return ChangeSet(pcs_set, content_set)
```

For some complex nodes, naively building a PCS child list of their direct children as shown in Listing 5.2 is insufficient to achieve the high degree of separation between different kinds of syntactical elements that is typical of structured merge, as discussed in Section 2.3.2. For example, consider the method declaration in Listing 5.3. Its direct children are the return type, the parameters, the thrown type and the body. In SPOON, the parameters and thrown types are kept in separate collections, but they are both considered to be direct children of the method declaration, and so appear in its child list. Figure 5.3 shows a schematic AST with a naively built child list. The end of the list of parameters is adjacent to the beginning of the list of thrown types, so there is no separation between these two distinct syntactical elements. As such, structural modifications to the end of the parameter list in one revision can conflic with structural modifications to the beginning of the thrown types list in another revision, which is undesirable.

Listing 5.3: *A JAVA method declaration, with the body omitted*

```
int div(int lhs, int rhs) throws ArithmeticException { ... }
```

To avoid collections of elements of different types within a child list to conflict with each other, SPORK inserts *intermediate virtual nodes* when building the PCS structure. A schematic example of this is shown in Figure 5.4, where the `parameters` and `thrown` virtual nodes separate the previously adjacent

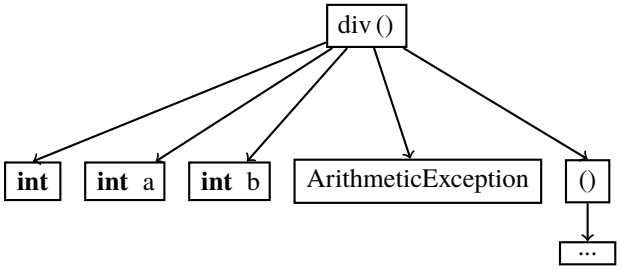


Figure 5.3: *Schematic drawing of naively built child list for the method declaration in Listing 5.3*

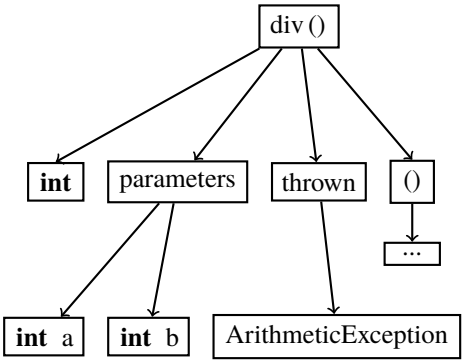


Figure 5.4: *Schematic drawing of the method declaration in Listing 5.3 with intermediate virtual nodes for the parameters and thrown types*

parameters and thrown types. It is important that all applicable intermediate virtual nodes are inserted even if the method node has no parameters or thrown types, as otherwise conflicts can occur due to insertions and deletions of the virtual nodes themselves. Note that the body and return type have no need of intermediate virtual nodes, as they are always represented by a single node each. When merging, SPORK treats intermediate virtual nodes like concrete nodes, except that it does not attempt to access their contents.

5.3.3 The SPORK-3DM algorithm

SPORK uses a modified version of the 3DM-MERGE algorithm presented in Section 4.4. In order to avoid confusing the two algorithms, we refer to the one implemented in SPORK as SPORK-3DM. Listing 5.4 presents psudeocode for SPORK-3DM.

Listing 5.4: *The SPORK-3DM algorithm*

```
1  def spork_3dm(base, left, right: Tree, m: Matching):
2      base_change_set = class_rep_change_set(base, m)
3      left_change_set = class_rep_change_set(left, m)
4      right_change_set = class_rep_change_set(right, m)
5
6      delta = base_change_set + left_change_set +
           right_change_set
7
8      for pcs in pcs_set(delta):
9          merge_content(pcs, base_change_set, delta)
10
11         others = get_other_roots(delta, pcs) \
12             + get_other_predecessors(delta, pcs) \
13             + get_other_successors(delta, pcs)
14
15         for other in others:
16             if other in base_change_set:
17                 delta = delta - {other}
18             elif pcs in base_change_set:
19                 delta = delta - {pcs}
20             else:
21                 conflict(pcs, other)
22
23     return delta
```

There are a three key differences between 3DM-MERGE and SPORK-3DM. Two of these are directly related to the get_other_$*$ functions. First, in 3DM-MERGE, the next get_other_$*$ function is only called if the previous one did not return an inconsistent element, while in SPORK-3DM all three functions are always called and the results are accumulated. Second, in 3DM-MERGE each function returns a single item, while in SPORK-3DM they return all items that match the criteria[5]. This means that SPORK-3DM immediately processes all inconsistencies a PCS triple is involved in. Without these alterations, SPORK exhibits flaky inconsistency detection in certain cases of move conflicts, which is discussed in more detail in Section 5.6.5.

The third difference is that SPORK-3DM clearly separates the processing of PCS triples and content tuples, while 3DM-MERGE does not. In 3DM-MERGE, the variable δ can be either a PCS triple or a content tuple, which makes it difficult to interpret the intended semantics of the get_other_$*$ functions as both input and output can be of two distinct data types. SPORK-3DM does not have

[5]For example, get_other_predecessors returns all PCSes with predecessor inconsistencies with respect to the argument PCS

a variable δ, but instead has pcs, which can only reference a PCS triple.

The merge_content function is responsible for merging content, and it does so separately for each child node in a PCS triple. Content is merged by the same principles as in the primary algorithm: keep the base revision's content if all content tuples are equal, keep the modified version if only one of the left and right revisions differ from the base, or do nothing if both the left and right revisions differ from the base. Any node that has more than one content tuple after the merge is involved in a hard content inconsistency as far as SPORK-3DM is concerned.

Such content inconsistencies are not necessarily true conflicts, and can often be solved automatically. Recall from Section 4.1 that a content tuple is represented as $c(v, m)$, where m is a multiset of values. The merge_content function only checks if m has been modified, but not which parts of it. Consider a method node that has both its name and its modifiers in m. This can cause the merge_content function to flag a hard inconsistency if one revision changes the name while the other changes the modifiers, even though such edits are not conflicting. Non-trivial content inconsistencies are resolved in the phase discussed in Section 5.4.

5.4 Converting from change set to SPOON AST

SPORK converts a change set into a SPOON AST in three subphases. First, it converts the change set to an intermediate tree structure, which is easier to traverse and reason about than the change set. This is described in Section 5.4.1. Second, it builds a SPOON AST from the intermediate tree, which is described in Section 5.4.2. Third, it locates and merges duplicated type members in the built SPOON AST, as described in Section 5.4.3.

5.4.1 Building an intermediate tree

The first subphase consists of converting the change set to an intermediate representation, called a `SporkTree`. All virtual nodes are filtered out during this process, including the intermediate virtual nodes discussed in Section 5.3.2. This leaves a tree representation that is easier to traverse and reason about as a tree than the change set is.

Converting a change set to a tree is one of the cases where the virtual nodes presented in Section 4.1 play an important role. The $\perp$ node is known to have the actual root node as its only child, and therefore serves as a known starting point for traversal. Similarly, the $\dashv$ node serves as the known start of a

child list, and the first PCS triple of the child list of a node v in a well-formed tree is always $pcs(v, \dashv, \ldots)$. Finally, the $\vdash$ node marks the end of a child list. Pseudocode for an algorithm that converts a change set to a tree is presented in Listing 5.5.

Listing 5.5: *Algorithm for building a tree from a change set. Note that* `to_spork_node` *only converts the root of* `tree`*, discarding any child nodes.*

```
def to_tree(tree: Tree, change_set: ChangeSet):
    spork_root = to_spork_node(tree)
    spork_root.set_content(change_set.get_content(tree))

    pcs = change_set.get_pcs(parent=tree, pred=⊣)
    while pcs.succ != ⊢:
        pcs = change_set.get_pcs(
            parent=tree, pred=pcs.succ
        )
        current_child = pcs.pred
        spork_root.add_child(
            # recursively build the child list
            pcs_to_tree(current_child, change_set)
        )

    return spork_root
```

Note that the algorithm shown in Listing 5.5 only describes the general idea of converting a well-formed change set to a tree. Accounting for intermediate virtual nodes and conflicts requires additional sophistication.

Structural conflicts are detected and handled when building the `SporkTree`. In some situations, conflicts can be automatically resolved by invoking a *conflict handler*. When a structural conflict is encountered and SPORK can identify the conflicting changes[6], all registered conflict handlers are called in a pre-determined order until one finds a solution, or until there are no more handlers. If no handler is able to resolve the conflict, a structural conflict is inserted into the `SporkTree`.

SPORK currently has two structural conflict handlers. The first of these resolves method ordering conflicts, such as when two different methods have been added to the same part of the file. The second handler optimistically resolves a conflict in which one of the conflicting sides is empty by choosing the non-empty contribution.

The `SporkTree` also keeps track of which revisions contribute to each subtree. For example, one subtree may have been added in the left revision,

[6]This is not always possible, see Section 5.6

another may remain unchanged from the base revision, while another still may be a mix of potentially all revisions. Knowing which subtrees consist of a single revision is useful both when building the SPOON AST, and in pretty-printing. An important detail is that it is not sufficient to determine that all nodes in a subtree stem from a single revision, as this may cause deletions, moves and content updates to be missed. For example, if a subtree is modified only with deletions, then all of its nodes still originate from the base revision, although there are fewer of them. To determine which revisions contribute to a particular subtree, the revisions of the PCS triples and content tuples must be inspected, and not the individual nodes.

5.4.2 Building a SPOON AST

The second subphase is to build a SPOON AST from the `SporkTree`. Each node in the `SporkTree` contains all the information required to create the corresponding SPOON node. It is therefore simply a matter of performing a pre-order traversal of the `SporkTree`, building the corresponding SPOON nodes, and attaching their contents. Whenever a single-revision subtree is encountered, the entire SPOON subtree is copied as-is instead of traversing it and copying each node.

The non-trivial content inconsistencies mentioned in Section 5.3.3 are also processed during this subphase. For each content tuple $c(v, m)$, the set m is split into its constituting values, which can for example be the name of a method and the method modifiers. This more granular representation is then checked for inconsistent edits in the left and right revisions. If inconsistent edits are found, there is still a chance that the conflict can be resolved. For example, the left revision changing the visibility of a method from public to private, and the right revision adding the final keyword is at this level of granularity detected as an inconsistent change, when it in fact is not.

As with structural conflicts, SPORK defines conflict handlers for different kinds of content. Each handler is specific to a type of content, such as method modifiers or comment text. If an inconsistent edit is found in a type of content for which there is a conflict handler, SPORK invokes the conflict handler, which may be able to resolve the conflict. For any type of content that does not have a defined conflict handler, inconsistent changes automatically lead to a conflict. The content conflict handlers depend on implementation details in SPOON, and are therefore not of great interest in regards this thesis. It is however relevant to note that the conflict handler for comment text performs a line-based three-way merge of the comment revisions, meaning that SPORK respects edits to

comments.

In the case that a structural conflict is encountered in the `SporkTree`, a dummy node is created in the SPOON AST. Both sides of the conflict are then attached to the dummy node, making them available to the pretty-printer in the final stage of SPORK's execution.

5.4.3 Merging duplicated type members

In cases where both the left and right revisions add non-identical versions of the same type member, a failure to match these results in type member duplication in the merge process. For example, a class may end up with two methods with the same signature, or two fields with the same name, which are semantic conflicts.

SPORK's final conflict handling strategy addresses this problem. It first collects all type declarations, which includes for example class and interface declarations. For each type declaration, SPORK searches for pairs of duplicated type members. For every pair it finds, SPORK starts the entire merge process over again for the two type members, using an empty node as the base revision[7]. Both of the duplicated type members are then removed from the type declaration, and the merged type member is inserted in their place.

5.5 Pretty-printing

SPORK uses SPOON's default pretty-printer to produce the final result of the merge, namely a JAVA source file. There are however two significant extensions to the printer in SPORK.

The first extension is reusing the original source code for single-revision subtrees. We refer to this as *reprinting* an element. This allows SPORK to retain most of the formatting from the file revisions, and only portions that have been changed need to be pretty-printed. Currently, reprinting is limited to fields, methods and comments, as reprinting arbitrary elements sometimes causes issues with separators such as semicolons and commas. Reprinting arbitrary elements is possible, but it requires additional work due to numerous corner cases.

The second extension is printing of conflicts. This occurs by separate mechanisms for structural conflicts and content conflicts. For structural conflicts, SPORK uses reprinting to print both sides of a conflict in a dummy node.

[7] This effectively makes it a two-way merge of the type members

For content conflicts, the output stream is monitored for markers inserted by the SPOON AST builder. When a marker is encountered, it is replaced with a conflict hunk.

5.6 Merge conflicts

This section presents how conflicts are expressed in the change set, and how SPORK handles or fails to handle different conflict situations. The conflicts are illustrated with dry-runs of 3DM-MERGE, as it is less verbose to illustrate than SPORK-3DM is. As mentioned in Section 5.4.2, content conflicts are dependent on implementation details in SPOON, and so this section deals primarily with structural conflicts.

5.6.1 Insert/insert conflicts

An insert/insert conflict occurs when both revisions insert one or more nodes in the same position in the AST. This results in a conflict, as 3DM-MERGE cannot determine the order the inserted nodes should be inserted in relative to each other. In the PCS structure, this gives rise to two possible paths through the child list from the start of the conflict to the end of the conflict.

Consider the three revisions of some arbitrary node p's child list in Table 5.1. Both the left and right revisions insert nodes after a, which results in an insert/insert conflict. Table 5.2 shows the merge process using 3DM-MERGE, and finds two distinct hard inconsistencies: the successor inconsistency between (a, b) and (a, d), and the predecessor inconsistency between $(d, \vdash)$ and $(c, \vdash)$[8]. An insert/insert conflict always gives rise to this pattern, in that it starts with a successor inconsistency and ends with a predecessor inconsistency.

SPORK can handle insert/insert conflicts. It recovers the two sides of the conflict by traversing both paths indicated by the initial successor conflict. Here, the left side of the conflict is found by traversing $(a, b), (b, c), (c, \vdash)$, and the right side by traversing $(a, d), (d, \vdash)$. Note how both these paths are present in the final merge state of Table 5.2.

5.6.2 Delete/delete conflicts

A delete/delete conflict occurs when the left and right revisions delete adjacent nodes. Consider the three revisions of a child list in Table 5.3. The merge

[8]This inconsistency is discovered twice, but inconsistencies are symmetric so it has no impact

Table 5.1: *An insert/insert conflict in the child list of p. $pcs(p, a, b)$ is abbreviated as (a, b).*

Revision	Child list	Simplified PCS child list
Left	abc	$(\dashv, a), (a, b), (b, c), (c, \vdash)$
Base	a	$(\dashv, a), (a, \vdash)$
Right	ad	$(\dashv, a), (a, d), (d, \vdash)$

Table 5.2: *3DM-MERGE process on revisions from Table 5.1.*

δ	δ'	Δ	Comment
$(\dashv, a)$	-	$\{(\dashv, a), (a, \vdash), (a, b), (b, c), (c, \vdash), (a, d), (d, \vdash)\}$	-
(a, b)	$(a, \vdash)$	$\{(\dashv, a), (a, \vdash), (a, b), (b, c), (c, \vdash), (a, d), (d, \vdash)\}$	$\delta' \in base$
(b, c)	-	$\{(\dashv, a), (a, b), (b, c), (c, \vdash), (a, d), (d, \vdash)\}$	-
$(c, \vdash)$	$(d, \vdash)$	$\{(\dashv, a), (a, b), (b, c), (c, \vdash), (a, d), (d, \vdash)\}$	conflict
(a, d)	(a, b)	$\{(\dashv, a), (a, b), (b, c), (c, \vdash), (a, d), (d, \vdash)\}$	conflict
$(d, \vdash)$	$(c, \vdash)$	$\{(\dashv, a), (a, b), (b, c), (c, \vdash), (a, d), (d, \vdash)\}$	conflict

Table 5.3: *A delete/delete conflict in the child list of p. $pcs(p, a, b)$ is abbreviated as (a, b).*

Revision	Child list	Simplified PCS child list
Left	a	$(\dashv, a), (a, \vdash)$
Base	ab	$(\dashv, a), (a, b)(b, \vdash)$
Right	b	$(\dashv, b), (b, \vdash)$

Table 5.4: *3DM-MERGE process on revisions from Table 5.3*

δ	δ'	Δ	Comment
$(\dashv, a)$	$(\dashv, b)$	$(\dashv, a), (a, b), (b, \vdash), (\dashv, b), (a, \vdash)$	$\delta \in base$
(a, b)	$(\dashv, b)$	$(a, b), (b, \vdash), (\dashv, b), (a, \vdash)$	$\delta \in base$
$(b, \vdash)$	$(a, \vdash)$	$(b, \vdash), (\dashv, b), (a, \vdash)$	$\delta \in base$
-	-	$(\dashv, b), (a, \vdash)$	Finished merge

process for these is presented in Table 5.4, and no inconsistencies are present in the merged child list $(\dashv, b), (a, \vdash)$. It is clearly consistent according to the consistency criteria presented in Section 4.1.3, and it is indeed unambiguous. However, the child list is evidently disjoint, as there is no path from b to a, so it does not actually encode the child list of an ordered tree. This shows that the consistency criteria are not strong enough to guarantee that the merge results in a change set that encodes a tree, and motivates the definition of a well-formed change set in Section 4.1.4.

While it is easy to detect a disjoint child list, SPORK currently cannot recover the conflicting sides. This is due to the fact that parts of the conflict have already been removed from the change set. Instead, an exception is thrown, and a resolution is left to the fallbacks described in Section 5.7.

5.6.3 Insert/delete conflicts

An insert/delete conflict occurs when one revision inserts a node where another revision deletes a node. Table 5.5 shows such a situation, where the left revision removes b and the right revision inserts c after b. 3DM-MERGE correctly identifies the hard inconsistency as shown in Table 5.6. As with the delete/delete conflict discussed in Section 5.6.2, it is difficult to retrieve the left and right sides of the conflict due to the fact that part of the conflict is not present in the final change set.

Insert/delete conflicts pose an additional problem, namely that naively traversing the PCS triples causes the deletion to override the insertion. Even though the merge is clearly inconsistent, a is the unambiguous successor to $\dashv$, meaning that a is the first node in the list. Furthermore, even though a is not the unambiguous predecessor of $\vdash$, $\vdash$ is in fact the unambiguous successor to a. A left-to-right traversal of the child list results in a merge where a is the only node, leading to a loss of the insertion of c.

SPORK is not able to recover the conflicting sides in the case of an insert/delete conflict, and therefore cannot deal with the conflict in structured mode. Instead, it keeps track of the hard inconsistencies that should be processed for each child list, and throws an exception if any inconsistency is left unprocessed. In the case of Table 5.6, the inconsistency between $(a, \vdash)$ and $(c, \vdash)$ is not processed by a simple left-to-right traversal, and so an exception is thrown. This leaves the fallback mechanisms discussed in Section 5.7 to deal with merging the child list.

Table 5.5: *An insert/delete conflict in the child list of p. $pcs(p, a, b)$ is abbreviated as (a, b).*

Revision	Child list	Simplified PCS child list
Left	a	$(\dashv, a), (a, \vdash)$
Base	ab	$(\dashv, a), (a, b)(b, \vdash)$
Right	abc	$(\dashv, a), (a, b), (b, c), (c, \vdash)$

Table 5.6: *3DM-MERGE process on revisions from Table 5.3*

δ	δ'	Δ	Comment
$(\dashv, a)$	-	$(\dashv, a), (a, b), (b, \vdash), (a, \vdash), (b, c), (c, \vdash)$	
(a, b)	$(a, \vdash)$	$(\dashv, a), (a, b), (b, \vdash), (a, \vdash), (b, c), (c, \vdash)$	$\delta \in base$
$(b, \vdash)$	$(a, \vdash)$	$(\dashv, a), (a, \vdash), (b, c), (c, \vdash)$	$\delta \in base$
$(a, \vdash)$	$(c, \vdash)$	$(\dashv, a), (a, \vdash), (b, c), (c, \vdash)$	conflict
(b, c)	-	$(\dashv, a), (a, \vdash), (b, c), (c, \vdash)$	
$(c, \vdash)$	$(a, \vdash)$	$(\dashv, a), (a, \vdash), (b, c), (c, \vdash)$	conflict

5.6.4 Delete/edit conflicts

A delete/edit conflict occurs when a subtree is deleted in one revision and edited in another, either structurally or through edits to node contents. Both 3DM-MERGE and SPORK-3DM miss this kind of conflict, as inconsistencies in PCS triples are only detected through edits to adjacent or overlapping nodes, and inconsistencies in content tuples are only detected through other edits to the same tuples. The core merge algorithm fundamentally cannot detect this kind of conflict as it does not result in an inconsistency in the change set elements, and so it must be handled separately.

Lindholm [16] suggests that delete/edit conflicts can be detected by scanning deleted subtrees for changes. This can be implemented in SPORK, but actually acting on a delete/edit conflict requires more information than the fact that it exists. It is not necessarily easy to infer where the conflict took place as the context of the root of the deleted subtree has itself been deleted from the change set, and implementing support for this in SPORK requires significant amounts of work. Therefore, SPORK currently does not search for delete/edit conflicts, and any occurrences of such conflicts are automatically resolved by ignoring the edits.

5.6.5 Move conflicts

Move conflicts are theoretically and practically troublesome. Intuitively, a move conflict occurs when the left and right revisions both manipulate the same node or the context around it such that the node ends up in two different places in the computed merge. Table 5.7 shows an example of a pure move/-move conflict, where the left revision moves b to the start of the child list, and the right revision moves b to the end of the child list.

In the presence of move conflicts, the vanilla 3DM-MERGE algorithm can become dependent on iteration order. Consider the dry-run of 3DM-MERGE presented in Table 5.8. In the first two iterations, (b, a) and $(b, \vdash)$ are processed in turn and found to be in soft inconsistencies with change set elements from the base revision. However, as these two elements have both been processed as δ and used to remove soft inconsistencies, the hard inconsistency between the two is not discovered as neither is processed again. If the elements from the base revision are processed before (b, a) and $(b, \vdash)$, then the hard inconsistency between (b, a) and $(b, \vdash)$ may be detected.

If one instead considers the SPORK-3DM algorithm, this problem does not apply, as all inconsistent change set elements are resolved in each iteration. When (b, a) is processed, all of $(\dashv, a), (\dashv, b), (b, c)$ are immediately found as inconsistent, so there is no risk of failing to detect inconsistencies.

While SPORK has the facilities for detecting inconsistencies caused by move conflicts, it in general has little ability to deal with them. There are several reasons for this. First of all, expressing a move conflict in plain-text form in a way that is compatible with GIT is not straightforward. Second, it is difficult to detect that an inconsistency is caused by a move conflict, the one exception being parent conflicts, which are discussed in Section 5.7.1. A move conflict can occur not only when both revisions move a node, but also when one revision moves a node, and the other inserts or deletes nodes adjacent to the moved node, effectively locking it in place through predecessor/successor contexts. We do not currently know of a general-purpose way of dealing with the variations of move conflicts that can occur, and it gets even more involved when there are unrelated conflicts present in the same child list, such as insert/insert or delete/delete conflicts. As such, SPORK's current move conflict handling is limited to the set of fallback mechanisms described in Section 5.7.

Table 5.7: *A move/move conflict in the child list of p. $pcs(p, a, b)$ is abbreviated as (a, b).*

Revision	Child list	Simplified PCS child list
Left	bac	$(\dashv, b), (b, a), (a, c), (c, \vdash)$
Base	abc	$(\dashv, a), (a, b), (b, c), (c, \vdash)$
Right	acb	$(\dashv, a), (a, c), (c, b), (b, \vdash)$

Table 5.8: *3DM-MERGE process on revisions from Table 5.7*

δ	δ'	Δ	Comment
(b, a)	$(\dashv, a)$	$(\dashv, a), (a, b), (b, c), (c, \vdash), (\dashv, b), (b, a), (a, c), (c, b), (b, \vdash)$	$\delta' \in base$
$(b, \vdash)$	$(c, \vdash)$	$(a, b), (b, c), (c, \vdash), (\dashv, b), (b, a), (a, c), (c, b), (b, \vdash)$	$\delta' \in base$
(a, b)	(c, b)	$(a, b), (b, c), (\dashv, b), (b, a), (a, c), (c, b), (b, \vdash)$	$\delta \in base$
(b, c)	(a, c)	$(b, c), (\dashv, b), (b, a), (a, c), (c, b), (b, \vdash)$	$\delta \in base$
(c, b)	$(\dashv, b)$	$(b, c), (\dashv, b), (b, a), (a, c), (c, b), (b, \vdash)$	conflict
(a, c)	(b, c)	$(b, c), (\dashv, b), (b, a), (a, c), (c, b), (b, \vdash)$	$\delta' \in base$
$(\dashv, b)$	(c, b)	$(\dashv, b), (b, a), (a, c), (c, b), (b, \vdash)$	conflict

5.7 Fallback mechanisms

Due to the issues surrounding move and delete conflicts discussed in Section 5.6, there are situations in which SPORK's pure structured merge may fail to produce a result. There are also situations where syntactical errors in the input files may cause parsing failures that are outside of SPORK's control. To mitigate the effect of such failures, SPORK has three fallback mechanisms.

5.7.1 Hard parent inconsistency elimination

Hard parent inconsistencies are always caused by move conflicts. As SPORK has no way of expressing move conflicts, avoiding them carries a high priority. After SPORK-3DM has finished merging, any identified hard inconsistencies are scanned for parent inconsistencies. If a parent inconsistency is found, all nodes that are found to have multiple parents are mapped to themselves in the class representatives map, and the same is done for all of their descendant nodes. This effectively removes the move as far as SPORK-3DM is concerned, which instead sees it as an insertion and a deletion. The raw merge is then recomputed and SPORK-3DM is invoked again with the new class representatives map.

5.7.2 Local line-based fallback

Parent inconsistency elimination only covers inter-parent move conflicts. This leaves intra-parent move conflicts and various forms of delete conflicts as potential causes for merge failures. The presence of such conflicts may cause SPORK to be unable to recover a node's child list from the change set when building the intermediate `SporkTree`, as described in Section 5.4. SPORK then attempts to perform a line-based merge of the node instead. We refer to this as a *local* fallback, as it only affects the node for which the child list can not be recovered.

The fallback works as follows. If v is the current parent whose child list SPORK is trying to convert into a `SporkTree`, and there is an error recovering the child list, SPORK immediately stops trying and discards any partially built tree. The nodes from the left and right revisions that match v, i.e. v_{left} and v_{right} respectively, are then resolved, and their source code is merged by a line-based merge.

5.7.3 Global line-based fallback

If SPORK encounters a critical error that cannot be handled by any of the other fallbacks, then the entire input is processed through a line-based merge. We refer to this as the *global* fallback, as it covers all of SPORK's execution phases, from parsing to pretty-printing. While the local fallback is necessary because of shortcomings in SPORK itself, the global fallback is necessary as there are many failure modes that are outside of SPORK's control. For example, SPOON may fail to parse an input file, either because of a bug in SPOON or because of syntactical errors in the input file. The only known failure mode for the global fallback is when a file path provided to SPORK does not refer to an existing file.

Chapter 6

Experimental methodology

This chapter presents the methodology for the experimental evaluation. Section Section 6.1 reiterates the research questions and presents hypotheses related to them, and Section 6.2 details the dataset. Section 6.3 and Section 6.4 detail the two experiments that aim to address the research questions, and Section 6.5 outlines the statistical methods used to analyze the results. Section 6.6 closes the chapter with an overview of the hardware and software used in the experiments. All evaluation scripts are written in PYTHON and are available in the online appendix[1].

6.1 Research questions

The research questions for the thesis are as follows.

> RQ1: How does SPORK compare to JDIME in terms of amounts and sizes of conflicts?
>
> RQ2: How does SPORK compare to JDIME in terms of running time?
>
> RQ3: How does SPORK compare to JDIME in terms of preserving source code formatting?
>
> RQ4: How does SPORK compare to JDIME in terms of producing merges that are semantically equivalent to the merges committed by developers?

The following sections present rationale for including each of the above research questions, as well as related hypotheses.

[1] `https://github.com/KTH/spork/tree/master/replication`

6.1.1 Rationale for RQ1

Merge conflict prevalence is an important aspect of a merge tool, and a reduction in merge conflicts is one of the primary selling points of structured merge tools. Conflict size is a determining factor in how difficult developers perceive conflicts to be, with increasing size being associated with increasing difficulty [56]. In cases where structured merge and unstructured merge both exhibit conflicts, structured merge also tends to produce individually smaller but more numerous conflicts [7]. It is therefore of interest to investigate both size and quantity, as ideally both should be minimized.

While SPORK's move-enabled algorithms can avoid some conflicts that JDIME encounters, the development process of SPORK also indicates that move-enabled merge may encounter conflicts where non-move-enabled merge does not. The hypotheses to accompany RQ1 therefore only suggest that differences can be found.

H1a: SPORK and JDIME produce differing amounts of conflicts

H1b: SPORK and JDIME produce differing conflict sizes

6.1.2 Rationale for RQ2

Running time is an important aspect for any tool that is used in an interactive computing environment. The user has to wait for the tool to run to completion before being able to proceed, meaning that all else being equal, a faster tool is preferable to a slower one. Making SPORK faster than JDIME was a key development goal, and so the hypothesis reflects this.

H2: SPORK is faster than JDIME

6.1.3 Rationale for RQ3

Formatting is an important aspect of source code. Formatting features such as characters per line, indentation and even blank lines play an important role in code readability, which in turn plays an important role in maintainability [15]. Due to a structured merge concluding in a pretty-printing step, a structured merge tool may produce output that in terms of formatting barely resembles the input. We argue that it is not the place of a merge tool to impose formatting conventions, rather that should be left to dedicated formatting tools or the developers themselves. In addition, the state of the practice in both diff and

merge is line-based, and the less a structured merge tool alters code formatting the more compatible it is with existing tools.

SPORK's source code reprinting and indentation detection were both designed to reduce unnecessary changes to existing formatting, giving it an advantage over JDIME that lacks such features. This is reflected in the hypothesis to accompany RQ3.

> H3: SPORK preserves formatting to a greater extent than JDIME does

6.1.4 Rationale for RQ4

A merge tool should ideally produce a merge that is semantically correct. As there in general is no precise semantic specification for the operation of a given program, the developers' actual commit is used as a reference. The hypothesis related to RQ4 is as follows.

> H4: SPORK and JDIME differ in the extent to which they produce merges that are semantically equivalent to those committed by developers

6.2 Dataset

Projects for the experiments were selected from the REAPER dataset of GITHUB repositories [62]. The REAPER dataset consists of 1.8 million GITHUB projects that are rated according to a variety of criteria that are indicators of a well-engineered project, such as the use of CI and unit tests, amount of documentation and amount of core contributors. The REAPER dataset has also been used in other merge studies [19, 54].

The REAPER dataset is large and contains projects in a variety of languages. The dataset was therefore filtered, first on the project level, and then on the merge commit level within selected projects. The filters are summarized in Table 6.1. Even with all of these filters applied, the dataset was still too large to be possible to process in a reasonable amount of time, and so the final selection was randomized.

As the filtering was extensive and thereby a potential source of bias in the data, the following two sections describe it in detail. Section 6.2.1 describes the project-level filtering, and Section 6.2.2 describes the merge commit filtering.

Table 6.1: *Filtering steps applied to arrive at the dataset*

Filter	Filter level
Uses Java	Project
Classified as well-engineered	Project
$>$ 50 stars	Project
$>$ 2 core contributors	Project
Is not a fork	Project
Latest commit buildable with Maven[1]	Project
Merge has precisely two parents	Merge commit
Merge base is unambiguous[2]	Merge commit
Merge is non-trivial[2]	Merge commit
Merge contains no non-content conflicts	Merge commit
All merge scenario revisions can be built	Merge commit
No conflict marker in any file revision	File merge

[1] As of the 10th of May 2020 [2] As defined in Section 6.2.2

6.2.1 Filtering projects

The first set of filters was applied to the REAPER dataset itself. They required that the project used JAVA, was classified as well-engineered[2], had more than 50 stars[3], and a minimum of 2 core contributors[3,4]. A total of 1174 projects fulfilled all of the criteria.

For each project that passed these initial filters, the GITHUB API was queried in order to filter out projects that were forks. The reason for this was that the REAPER dataset was found to contain mislabeled forks. For example, `dblock-/oshi`[5] is listed as having 73 stars and good scores in most respects, but it is actually an unmaintained fork of `oshi/oshi`[6], and in fact has no stars. The same applies to for example `spinscale/dropwizard-jobs`[7], which is

[2]According to the `randomforest_org` classification in the REAPER dataset. This classifier has the lowest false positive rate, which decreases the probability of selecting a non-well-engineered project.

[3]The minimum values for stars and core contributors were meant to increase the chances of selecting popular projects with at least a few active committers. The precise values are arbitrarily chosen, but are necessary to choose to reduce the size of the dataset.

[4]The core contributors are defined as the smallest set of contributors responsible for at least 80% of all commits

[5]`https://github.com/dblock/oshi`

[6]`https://github.com/oshi/oshi`

[7]`https://github.com/spinscale/dropwizard-jobs`

a fork of `dropwizard-jobs/dropwizard-jobs`[8], so it is not an isolated issue. As such forks can be years behind the actual project in terms of commits, all forks were discarded as a precaution.

A final set of filtering on the project level was performed to find projects that could be built using MAVEN on the test system, as this was the tool chosen for performing the experiments that require code compilation. Each project was therefore scanned for a `pom.xml` file in the latest commit[9] of the default branch, indicating the use of MAVEN. If there was such a file, the project was cloned and built with MAVEN. If the build succeeded within at most 5 minutes, the project was added to a list of candidate projects. Two of these were used for the development of SPORK, and were therefore excluded. A list of the remaining 359 projects can be found in the online appendix[10].

6.2.2 Filtering merge commits

Both experiments required the base commit of each merge scenario to be located. In cases where a commit history has a criss-cross pattern[11], there can be multiple possible merge bases, leading to the merge base being *ambiguous*. In such cases, GIT resorts to a recursive merge strategy[12]. As this turns a merge into a multi-stage process, it complicates merge replay, as well as identification of the files involved in the merge. For these reasons, merge commits with an ambiguous merge base were excluded from the dataset. In some rare cases, the `git merge-base` command used to locate the merge base did not return a value for unknown reasons, and these commits were also excluded.

As noted by Cavalcanti et al. [19], most merges are trivial and involve disjoint sets of files, which means that no merge tool is required. The merge commits were therefore filtered to include only commits where at least one JAVA source file had been edited in both the left and right revisions.

There are only two types of conflicts in GIT that concern the content of a file: *content* conflicts and *add/add* conflicts [13]. The former occurs when the same file has been edited in both the left and right revisions, and the latter when the left and right revision have both added the same file, but with differing contents. Non-content conflict types in GIT are not relevant to this thesis. For example, a delete/modify conflict involves a file that has been deleted in one

[8] `https://github.com/dropwizard-jobs/dropwizard-jobs`

[9] As of the 10th of May 2020

[10] `https://github.com/KTH/spork/tree/master/replication`

[11] `https://git-scm.com/docs/git-merge-base`

[12] `https://git-scm.com/docs/git-merge#Documentation/git-merge.txt-recursive`

revision, and updated in another. A rename/rename conflict involves a file that has been renamed to different things in in each revision. Such conflicts cannot be resolved by either SPORK or JDIME as they only merge one file at a time, and have no context other than the revisions of that file. Therefore, scenarios containing such non-content conflicts were excluded.

Individual file merges in which any of the revisions contained conflict markers were also excluded, as pre-existing markers interfered with the automatic conflict analysis. Such cases are not common, but they do occur [41]. This also means that merge scenarios in which all non-trivial file merges contained conflict markers were excluded from the dataset entirely.

According to Cavalcanti et al. [19], it is often the case that old commits cannot be built, for example due to missing dependencies. If any of the base, left or right revisions do not build, the merge is unlikely to build through no fault of the merge tool. If the expected revision does not build, there is no point of reference. Therefore, a final filter was applied to vet out candidates that could not be built on the test system. This filter excluded merge scenarios in which any revision involved in the merge, including the merge commit itself, could not be built with MAVEN.

6.2.3 Executing the experiments

During the execution of the experiments, projects were randomly drawn from the candidate projects, and viable merge commits were extracted. The amount of time spent extracting merge commits from a single project was limited to 1 hour, and the amount of merge commits sampled from each project was limited to a maximum of 15. The time limit was to avoid getting stuck on a project with large amounts of merge commits, and the quantity limit was to avoid a situation where a few projects accounted for a majority of all merge commits. When the merge commits had been extracted from a project, both experiments were executed in sequence, after which the process started over from the merge extraction phase for the next project.

In total, 890 merge scenarios from 119 different projects were extracted, comprising a total of 1740 file merges. A summary of these projects, merge scenarios and file merges can be found in Appendix A.

6.3 Experiment protocol for RQ1, RQ2 and RQ3

The first experiment was designed to address RQ1, RQ2 and RQ3, which concern conflict features, running time and formatting preservation, respectively. The general idea of the experiment was to replay individual file merges with the different merge tools, time each execution, and then gather data about conflicts and formatting preservation. The experiment was inspired by the one performed by Shen et al. [41], but with an overall stricter formatting comparison to better capture important formatting features [15].

For each merge scenario, all individual file merges were extracted. This involved finding all revisions of a file, including the one that was actually committed as part of the merge, which is referred to as the *expected* revision. File revisions were located with GIT's merge functionality. It was therefore not always possible to find all revisions, for example if developers had manually interfered with file paths in a merge. If any of the base, left, right or expected revisions of a file could not be found, the file merge was skipped.

SPORK and JDIME were then then applied to the base, left and right revisions of each file merge, and the files they produced were stored. The file produced by a merge tool for a given file merge scenario is referred to as the *replayed* revision for that file merge and merge tool.

For the purpose of addressing RQ1, each replayed revision was scraped for conflict hunks. For each file, the amount of conflict hunks and the total amounts of lines in the conflicts were recorded. Scraping conflict hunks becomes difficult if there are pre-existing conflict markers in the files, which is why such file merges were excluded as described in Section 6.2.2.

To address RQ2, The execution of each file merge was also timed, with the running time being measured as the wall time from the moment the merge tool was invoked to the moment it exited. The file merges were run sequentially to avoid the merge tools interfering with each other. While any individual running time is not reliable due to a given file merge only being executed once, the large amount of executed scenarios should still give an accurate depiction of real-world performance. A timeout was set to 300 seconds, after which the merge was forcibly aborted. If a merge tool timed out, the running time was set to exactly 300 seconds. This allows timeouts to be easily distinguished, as all other running times are floating point values.

To address RQ3, a measure of formatting preservation had to be decided upon. It is reasonable to assume that the project's formatting style is well

represented by the expected file revision, and so it served as a ground truth for the correct merge. To determine how closely a replayed revision resembled its corresponding expected revision, the GITDIFF[13] command was used. GITDIFF produces a line-based edit script of insertions and deletions for two files, and the sum of insertions and deletions is the *diff size*. The precise command that was used is shown below.

```
git diff --no-index --numstat --ignore-space-at-eol \
         --ignore-cr-at-eol
```

6.4 Experiment protocol for RQ4

The second experiment was designed to evaluate the general correctness of the merge tools. The idea was to determine to what extent the merges produced by SPORK and JDIME corresponded to those actually committed by the developers in terms of semantics. It is important to note that even given a perfect semantic equality check, this approach is an approximation of merge correctness, as developers are fallible and may produce incorrect solutions to merge conflicts [41, 56]. In addition, the unstructured merge tools used by most projects may miss conflicts and thus produce semantically incorrect merges [7].

This experiment was far more complex than the one described in Section 6.3. The rationale for the experiment's design is presented in Section 6.4.1, and the actual experiment protocol is detailed in Section 6.4.2.

6.4.1 Experiment design

In designing this experiment, inspiration was drawn from the benchmarks presented in Section 3.5. The works comparing merge tools largely avoid attempting to evaluate the correctness of successful merges, focusing instead on reported conflicts and running times [7, 20, 21]. Shen et al. [41] additionally used textual comparisons between replayed revisions and their corresponding expected revisions to evaluate the precision and recall of merge tools. The premise of structured merge is however that text does not well reflect the software artifact [4, 7], and so drawing any conclusions about semantics from textual comparisons is inappropriate[14].

[13] `https://git-scm.com/docs/git-diff`

[14] Note that Shen et al. [41] do not claim to measure semantic precision and accuracy, but they do use the measurements to argue for the efficacy of a merge tool

Other closely related works are those by Cavalcanti et al. [19] and Tavares et al. [10]. The experiments revolved around building projects and executing their test suites, with the purpose of identifying false positives and false negatives. The authors only investigated scenarios in which one tool reported a conflict that the other one did not. These are all exceptional cases, again primarily focused on conflicts, and so reveal little about how the tools perform in general.

Due to the complexity of structured merge tools, SPORK and JDIME are highly likely to compute different merges even when none of the tools encounter a conflict. As the vast majority of merges computed by a structured merge tool are conflict free, evaluating the conflict free cases is arguably more important than evaluating those with conflicts. One approach for such an evaluation is to use the expected revision as the ground truth for the correct merge resolution. Although the developers' commit is not always correct [41, 56], it is reasonable to assume that in most cases, it is.

Even having a replayed and expected revision to compare, there is no generalized method for computing the semantic equivalence of two programs. The problem is in fact undecidable or intractable for all but the most trivial programs [63]. Zhu and He [18] and Zhu, He, and Yu [46] attempted to evaluate merge correctness in the general case by comparing the AST of a replayed revision to the corresponding expected revision, which can serve as an approximation of semantic equivalency. However, this AST-based approach hinged on the fact that the compared merge tools were two variations of JDIME, both employing the same AST, which is not the case for SPORK and JDIME. Using either SPORK's or JDIME's underlying AST model naturally favors that tool, and using either tool's diff algorithm would have the same effect. A straight equality check with an unrelated AST model is also insufficient. ASTs that are specifically designed for code analysis and transformation, such as JAVAPARSER[15], are too granular and formatting dependent. ASTs that are designed for code compilation, such as the one used in OPENJDK, lack the introspective capabilities required to easily determine equality, and even having that it is still affected by irrelevant features such as method ordering.

Instead, we opted for approximating semantic equality by comparing compiled bytecode using SOOTDIFF [64]. SOOTDIFF uses the SOOT framework[16] to transform JVM bytecode into the simpler JIMPLE format, which has a drastically reduced instruction set and a large amount of other simplifications. Using bytecode for comparison reduces variations caused by differing formatting,

[15] `https://javaparser.org/`

[16] https://github.com/Sable/soot

and JIMPLE's simplified format further reduces non-semantic variations in the bytecode. In addition to these benefits, SOOTDIFF also performs various optimizations to further eliminate insignificant bytecode differences, and then computes a diff. To the best of our knowledge, this is the furthest one can go in reducing non-semantic differences between two revisions of JAVA files, and source file variations such as formatting and method ordering have no impact[17]. SOOTDIFF is also entirely independent of both SPORK and JDIME. The experiment protocol is detailed in Section 6.4.2.

6.4.2 Experiment protocol

For each project, the experiment was conducted as follows. First, the project was checked out to the merge commit, referred to as the expected merge, and then the project was built. The non-trivial file merges that involved JAVA files were identified in the same way as described in Section 6.3, and the compiled class files corresponding to these source files were located in the compiled output. These were the expected class files that served as a reference for the merge scenario. In some cases, it was not possible to locate a class file corresponding to a changed source file. For example, a source file may not be part of the project's primary test or production codebase, and therefore is not compiled when the project is built. Any scenario where no expected class files could be located was skipped.

The following process was then repeated for both merge tools. First, the tool was configured as the merge driver for JAVA source files in GIT. The scenario was then replayed by checking out to the left commit and merging the right commit into it, without creating a new commit to avoid altering the state of the repository unnecessarily. This replayed merge was then built in the same way the expected merge was.

For each expected class file, the corresponding class file in the replayed merge was located. For each pair of corresponding replayed and expected class files, SOOTDIFF was used to compute the diff. A zero-size diff indicated equality. Expected class files that had no counterpart in the replayed merge automatically counted as unequal.

If a replayed class file compared equal to an expected class file, they were considered to be semantically equivalent. If all expected class files from a single expected source file compared equal to their replayed counterparts, the expected and replayed source files were considered to be semantically equiv-

[17]Non-semantic features such as method ordering do in fact impact JVM bytecode, as the constants pool often changes with different method ordering

alent. The expected source file had then been semantically *reproduced*. Similarly, if a merge tool reproduced all source files for a given merge scenario, the replayed merge in its entirety was considered to be semantically equivalent to the expected merge, which had then been reproduced.

As a final step for each merge scenario, the state of the repository and the working tree was reset. This ensured that the state of the repository was the same for each merge scenario and tool, and that no files produced by a previous merge or compile process were present for the next merge replay.

6.5 Statistical tests

Every experiment generated pairwise data, with one result for JDIME and one for SPORK. All of the data generated in the experiment described in Section 6.3 is ordinal in nature. To reiterate, this relates to four separate measurements: one for conflict sizes, one for conflict quantities, one for diff sizes, and one for running times. Each kind of measurement was analyzed separately.

To determine if the results for the tools differed for a given kind of measurement, the Wilcoxon signed-rank test was used. As all but the running time measurements yielded discrete values, there was a significant risk of zero-differences among sample pairs, which negatively affects the reliability of the Wilcoxon signed-rank test. Therefore, the rank adjustment proposed by Pratt [65] was applied, as well as the corresponding adjustment to the standard error proposed by Cureton [66].

As SPORK and JDIME were not allowed to fall back on line-based merge if they failed to produce a structured merge[18], both tools simply failed to produce output for some merges. File merges for which at least one tool failed to produce a result were excluded from all forms of statistical analysis.

The data from the bytecode comparison experiment described in Section 6.4 is arguably ordinal when considering the amount of equal source files per merge scenario. However, as many merge scenarios encompassed only a single expected source file, and many source files corresponded to just a single class file, the results per merge scenario and even per source file are binary to a large extent. All measurements were therefore considered to be binary: a source file was either reproduced or not reproduced, and a merge scenario was either reproduced or not reproduced. Recall that a source file counts as reproduced if it reproduced all related expected class files, and that a merge

[18] Note that in the case of SPORK, this refers to the global fallback. The local fallback was still allowed.

scenario counts as reproduced if all source files involved in it were reproduced.

It is also important to consider that there in general is a dependence between built class files in the same merge scenario. If a tool fails with just one source file in a scenario that contains many, the build may fail. This affects potentially all class files that should be built[19]. Two measures of differing granularity were therefore considered: the amount of reproduced source files, and the amount of reproduced merge scenarios. The former measure puts greater weight on merge scenarios with many involved source files, while the latter puts equal weight on all merge scenarios. McNemar's test was then used to determine if either tool more often succeeded in scenarios where the other failed.

6.6 Test system

The test system hardware consisted of an Intel Core i7 6700k clocked at 4.6 GHz, 32 GiB of RAM clocked at 2666 MHz and a SATA SSD with read and write speeds of 500 MB/s. An important note is that the CPU was manually locked to 4.6 GHz across all cores for the experiments, eliminating variability caused by intermittent clock boosts.

The test system ran ARCHLINUX with kernel 5.6.13, OPENJDK 1.8.0u242 and CPYTHON 3.8.2. JDIME was built from source using the latest version on the development branch[20], and SPORK v0.5.0[21] was used.

In addition to this, open source PYTHON projects were used to implement the experiments and analyze the results. The ones that were most crucial are SCIPY [67], NUMPY [68], Pandas [69] and GITPYTHON[22]. For a full list, refer to the online appendix[23].

[19] Some projects have multiple build stages, so it is in some circumstances possible to partially build a project

[20] `https://github.com/se-passau/jdime/commit/100aeecedcd36cd7e9165fd0495c7781344593e0`

[21] `https://github.com/kth/spork/releases/v0.5.0`

[22] `https://github.com/gitpython-developers/GitPython`

[23] `https://github.com/KTH/spork/tree/master/replication`

Chapter 7

Experimental results

This chapter presents the results from the experiments. Section 7.1 through Section 7.3 present the results from the experiment described in Section 6.3, and Section 7.4 presents the results from the experiment described in Section 6.4.

7.1 Conflict hunk quantities and sizes

This section presents results related to RQ1, which is recapitulated below.

> RQ1: How does SPORK compare to JDIME in terms of amounts and sizes of conflicts?

Section 7.1.1 presents results on conflict hunk quantities, and Section 7.1.2 presents results on conflict hunk sizes. In conjunction, these two sections address RQ1.

7.1.1 Conflict hunk quantities

Conflict quantities were investigated by comparing the amount of conflict hunks per file. File merges were only included in this comparison if both SPORK and JDIME successfully produced a merge, and at least one of them encountered a conflict. Out of the total of 1740 file merges, 319 file merges met these criteria.

JDIME encountered conflicts in 212 of the file merges, producing a total of 419 conflict hunks. SPORK encountered conflicts in 202 of the file merges, producing a total of 308 conflict hunks. Figure 7.1 shows a histogram of the distribution of conflict hunk quantities for SPORK and JDIME. Note that the

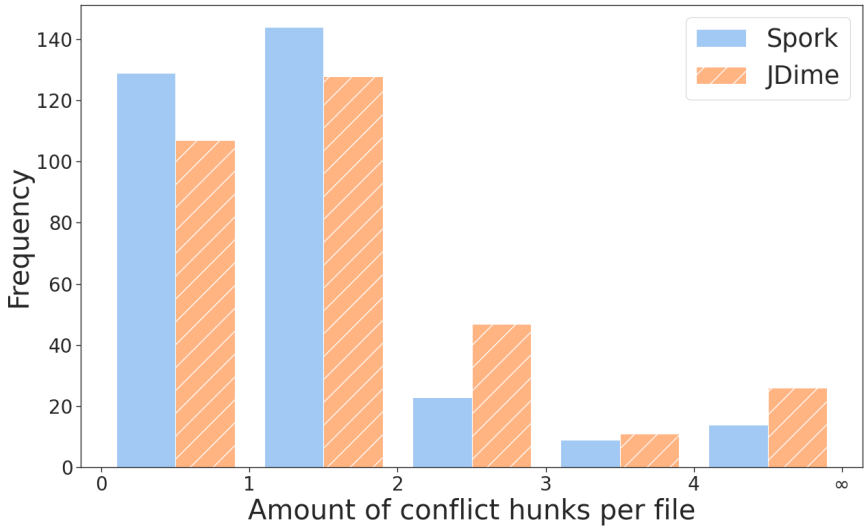


Figure 7.1: *Histogram of conflict hunk quantities per file for* SPORK *and* JDIME*. Lower is better. Each histogram bin contains the frequency of values in the range* $[L, R)$*, where* L *and* R *are the values to the left and right of the bin, respectively.*

first bin, that shows only the cases with 0 conflicts[1], has a special meaning; it shows the amount of files in which one of SPORK or JDIME reported at least one conflict, and the other tool reported none. Figure 7.1 shows a clear advantage for SPORK, with fewer overall conflict hunks.

A one-sided Wilcoxon signed-rank test was used to determine which tool produced fewer conflicts overall. $H0_{1a}$ was adopted as the null hypothesis, as defined below. The initial hypothesis H1a was refined to the stronger hypothesis H1a' in light of the clear trends in the data, and adopted as the alternative hypothesis.

> $H0_{1a}$: SPORK and JDIME produce the same amount of conflict hunks
>
> H1a': SPORK produces fewer conflicts than JDIME does

The test yielded a p-value of 3.6e-4. This lead to the null hypothesis being discarded, and the alternative hypothesis that SPORK produces fewer conflicts than JDIME was instead accepted.

An exceptional thing to note is that in 12 of the file merges, SPORK reported that it encountered a conflict, but it wrote no conflict hunk. There are at least two reasons for this behavior. First, SPOON incorporates a lot of information in

[1]The range $[0, 1)$ contains only 0

the AST that is not used for printing, such as implicit package references[2] and the types of variables. Conflicts that occur on such elements are not printed, and are typically caused by SPOON failing to resolve an implicit reference or type in one or more revisions. Out of the 12 considered file merges, 8 conflicts were caused by such pseudo conflicts, while in the remaining 4 SPORK failed to write a conflict due to bugs in the conflict handling mechanism.

There were also noteworthy differences between the kinds of conflicts detected by each tool. Spot checks of merge scenarios with conflicts were used to identify two forms of conflicts that only SPORK reported. First, SPORK sometimes reported false positives on near identical and even identical additions between the left and right revisions. The near identical addition conflicts were likely due to the conservative left-to-right matching in SPORK. Conflicts on identical additions were typically caused by SPOON inconsistently resolving implicit elements across revisions, causing the related explicit elements not to be matched, even though they were in fact identical in terms of source code. Second, SPORK performs line-based merge of comments, and this can naturally result in conflicts. It appears as if JDIME does not merge comments at all when running in structured mode, but instead always appears to pick the comment from the left revision, which makes conflicts an impossibility.

At the time of the experiments, the line-based merge in SPORK was also affected by a bug that caused a conflict if the base revision was the empty string, and precisely one of the left and right revisions was also the empty string. Then, a conflict between the left and right revisions would printed, even though the non-empty string was the unambiguous merge resolution. In particular, this caused problems when merging file headers[3] in the cases where the file header did not exist in the base revision, but was added in one of the left or right revision. This always resulted in a conflict, and out of the 308 conflict hunks written by SPORK, at least 45 were caused solely by the bug causing false positives on added file headers. This problem is magnified by the fact that these 45 conflicts were the only conflicts in their respective files. This means that in more than a fifth of the 202 files in which SPORK printed conflicts, it did so solely because of this bug.

The spot checks also revealed 4 forms of conflicts that only JDIME reported. First, JDIME correctly identified and reported several delete/edit conflicts, which SPORK is currently unable to do, as discussed in Section 5.6.4. Second, JDIME showed a tendency to produce spurious matches between unrelated fields added by the left and right revisions, which caused several false

[2] E.g. a type reference `String` having the implicit package reference `java.lang`

[3] A block comment at the very start of a file

```
1  public class Adde {
2    int add(int a, int b) {
3      return a + b;
4    }
5  }
```

(a) *Base revision*

```
1  public class Adder {
2    int add(int a, int b) {
3      return a + b;
4    }
5  }
```

(b) *Left revision*

```
1  public class Adde {
2    public int add(int a, int b) {
3      return a + b;
4    }
5  }
```

(c) *Right revision*

```
1  public class Adder {
2    public int add(int a, int b) {
3      return a + b;
4    }
5  }
```

(d) *SPORK's merge*

```
1  <<<<<<< left/Adder.java
2  public class Adder {
3    int add(int a, int b) {
4      return a + b;
5    }
6  }
7  =======
8  public class Adde {
9    public int add(int a, int b) {
10     return a + b;
11   }
12 }
13 >>>>>>> right/Adder.java
```

(e) *JDIME's merge*

Figure 7.2: *Artificial merge scenario involving a class renaming, along with the merges computed by SPORK and JDIME*

positive conflicts that SPORK avoided. Third, in a few isolated cases, JDIME reported the entire file as a conflict, the cause of which was never clearly established. Finally, JDIME always reports conflicts on renamings, which SPORK rarely does due to its move-enabled algorithms, and this resulted in several additional conflicts for JDIME. This is to be expected as JDIME's lookahead feature was not enabled [14], but even with it enabled, JDIME either still encountered the conflict, or even crashed.

Figure 7.2 shows an artificial merge scenario exemplifying a class renaming that occurred multiple times in the experiments. The left revision fixes a typo in the class name, and the right revision modifies the class body by adding a visibility modifier to the add method. SPORK has no trouble resolving this merge due to GUMTREE correctly computing matches between all class nodes, despite the slightly differing names. JDIME on the other hand does not find this match, and thus produces a conflict. This occurs even with looka-

head mechanism in JDIME enabled. With method renamings, the lookahead mechanism appears to cause crashes instead of conflicts.

7.1.2 Conflict hunk sizes

As there is no simple way to correlate conflict hunks to each other, the measure used in this section is the mean conflict hunk size per file, with the size of each hunk measured as the amount conflicting lines. File merges were only included in the comparison if at least one of SPORK or JDIME wrote at least one conflict hunk, which only applied to 91 file merges.

SPORK produced conflicts with a median conflict size of 2.14 lines, while JDIME's corresponding median is 2.00 lines, making for an absolute difference of 0.14 lines to JDIME's advantage. Figure 7.3 presents a histogram of SPORK's and JDIME's mean conflict hunk sizes, which indicates that JDIME performed better overall, having more of the smallest and fewer of the largest conflicts. In a pairwise comparison of the mean conflict size per file, JDIME yielded smaller conflict sizes in 40 cases, and SPORK in 20 cases, while both tools produced equally large conflicts in the remaining 31 cases.

As with the conflict quantities, a one-sided Wilcoxon signed-rank test was applied to the results to determine which tool produced fewer conflicts overall. $H0_{1b}$ was adopted as the null hypothesis, as defined below. The initial hypothesis H1b was refined with the stronger hypothesis H1b', and adopted as the alternative hypothesis.

> $H0_{1b}$: SPORK and JDIME produce equally large conflicts
>
> H1b': JDIME produces smaller conflicts than SPORK does

The test yielded a p-value of 5.6e-3, meaning that the null hypothesis was discarded. The alternative hypothesis that JDIME produces smaller conflicts than SPORK does was instead accepted.

There are at least three reasons as to why JDIME produced smaller conflicts. First, SPORK sometimes has trouble reconciling similar additions in the left and right revisions, which should be handled by the left-to-right matching as described in Section 5.2. This difficulty stems from the fact that SPORK uses a conservative matcher[4] when matching left-to-right, and is conservative even when including these matchings in the class representatives, as described in Section 5.3.1. This can cause large blocks such as method bodies and try

[4]The `XY` matcher in the GUMTREE library, as opposed to the more powerful GUMTREE matcher which is used for the base-to-left and base-to-right

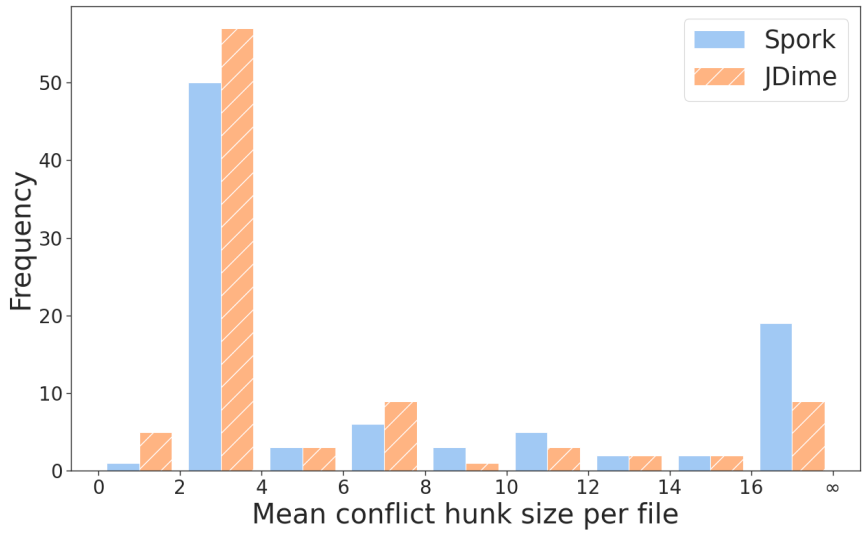


Figure 7.3: *Histogram of mean conflict sizes for* SPORK *and* JDIME*. Lower is better. Each histogram bin contains the frequency of values in the range* $[L, R)$*, where* L *and* R *are the values to the left and right of the bin, respectively.*

blocks of dozens of lines of code to fail to be matched, leading to large conflicts. A less conservative left-to-right matching in SPORK solves this problem, but increases the amount of spurious matchings, which in turn increases move conflicts.

Second, several of SPORK's conflict hunks were produced by a line-based merge in the local fallback, which resulted in several large conflicts. In the most extreme case, the local fallback activated on the child list of a class declaration, such that the class body was merged with line-based merge. In this case, the left revision contained LF line endings, while the left revision contained CRLF line endings, resulting in a conflict hunk of the entire class body.

Third, SPORK and JDIME have different notions of insert conflicts. For example, if the left revision inserts abc in the same place that the right revision inserts d, SPORK considers all of abc to conflict with d. JDIME on the other hand considers only c and d to be conflicting, resulting in a smaller conflict.

7.2 Running times

This section presents results related to RQ2, which is recapitulated below.

> RQ2: How does SPORK compare to JDIME in terms of running time?

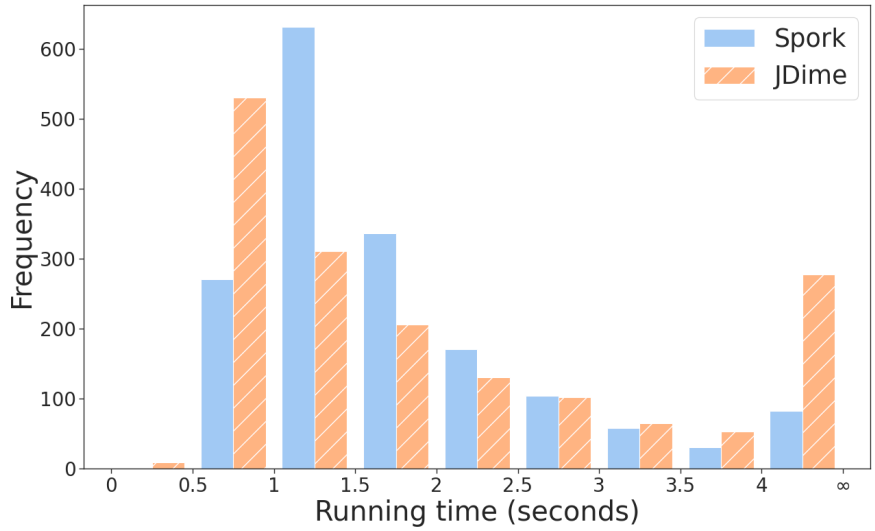


Figure 7.4: *Histogram of file merge running times for for* SPORK *and* JDIME*. Lower is better. Each histogram bin contains the frequency of values in the range* $[L, R)$*, where* L *and* R *are the values to the left and right of the bin, respectively.*

Out of the total of 1740 file merges, there are 1685 file merges for which both SPORK and JDIME produced a merged file. Only the 1685 file merges in which both tools produced results were considered for this comparison. The failures are discussed in more detail in Section 7.5, but it is relevant for these results to note that 16 of JDIME's failures were caused by timing out at 300 seconds, whereas SPORK exhibited no timeouts.

In order to be compatible with GIT, a merge tool must report the amount of conflicts encountered as the exit status[5]. JDIME only does this if it is instructed to gather statistics, but this negatively impacts running time performance. The running times presented in this section were measured with statistics collection disabled in JDIME.

In the median case, SPORK and JDIME performed almost identically, with median running times of 1.45 seconds for SPORK and 1.48 seconds for JDIME. The difference of 0.03 seconds is negligible from a user perspective.

The histogram of running times in Figure 7.4 does however indicate stark differences in the distribution of running times. There is a clear trend in the data showing that JDIME has more both of the smallest and of the largest running times. In terms of the smallest running times, JDIME finished 539 of the

[5] `https://git-scm.com/docs/git-merge-file`

file merges in less than 1 second, while SPORK finished 271 of the file merges in the same time span. JDIME also achieved a minimum running time of 0.45 seconds, which is significantly smaller than SPORK's minimum of 0.66 seconds. In terms of the largest running times, JDIME finished 278 of the file merges in 4 seconds or more, while SPORK finished 83 of the file merges in the same time span. JDIME also achieved a maximum running time of 294.12 seconds, which is an order of magnitude larger than SPORK's maximum of 16.38 seconds.

This trend is not only visible in the extreme cases. Considering JDIME's median of 1.48 seconds as a point of separation, there are 781 file merges in which both JDIME and SPORK finished in at most that time, and 904 file merges in which at least one tool exceeded that time. Out of the 781 file merges where both tools finished in at most 1.48 seconds, JDIME was faster in 709 cases, and SPORK was faster in 72 cases. In the 904 cases where at least one tool finished slower than 1.48 seconds, JDIME was faster in 186 cases, and SPORK was faster in 718 cases. Although JDIME was faster in a majority of the file merges, the differences increase in magnitude with increasing running time. This is reflected in the fact that JDIME's total running time was 5290 seconds[6], while SPORK's total running time was 3120 seconds.

Manual re-executions of a small selection of file merges revealed differences in the time that the tools spend parsing, and the time they spend merging. In all manual tests, JDIME spent less time parsing input than SPORK did, but SPORK spent less time merging. For the smaller merges, time spent parsing can constitute a significant portion of the total running time, which hurts SPORK's performance due to slow parsing. In several of the smallest merges, JDIME managed to complete the merge, including printing the output, before SPORK was done parsing the input. However, as the total running time of the merges increases, the fraction of the total time spent parsing decreases, allowing SPORK to pull ahead, likely due to faster merging.

A one-sided Wilcoxon signed-rank test was applied to the results to determine which tool achieved better running times overall. $H0_2$ was adopted as the null hypothesis, as defined below, and H2 was adopted as the alternative hypothesis.

> $H0_2$: There is no difference between SPORK's and JDIME's running times
>
> H2: SPORK is faster than JDIME

The test yielded a p-value of 7.39e-14, meaning that the null hypothesis was

[6] Accounting for the 16 timeouts, JDIME's real total running time was in fact 10090 seconds

rejected. Therefore, the alternative hypothesis that SPORK is faster than JDIME was accepted.

7.3 Preservation of formatting

This section presents results related to RQ3, which is recapitulated below.

> RQ3: How does SPORK compare to JDIME in terms of preserving source code formatting?

Out of the total of 1740 file merges, there are 1366 file merges for which both SPORK and JDIME produced conflict free results. Only these 1366 file merges were considered for this comparison. Formatting preservation was estimated with GITDIFF, as described in Section 6.3. Recall that the measure is the diff size between the expected and replayed revision, which can be interpreted as the amount of lines by which the two files differ.

SPORK produced file merges with a median diff size of 67, while JDIME produced file merges with a median diff size of 319, making for a difference of 252 in SPORK's favor. Figure 7.5 shows a histogram of the diff sizes for SPORK and JDIME. It clearly corroborates the difference in medians, showing that SPORK preserved formatting to a greater extent. This notion is further

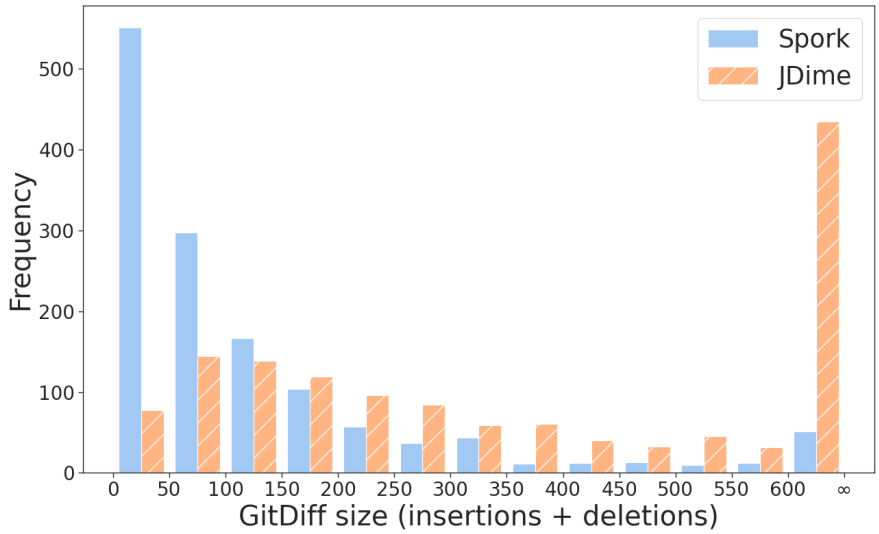


Figure 7.5: *Histogram of GITDIFF sizes for SPORK and JDIME. Lower is better. Each histogram bin contains the frequency of values in the range $[L, R)$, where L and R are the values to the left and right of the bin, respectively.*

strengthened by a pairwise comparison of each file merge scenario, showing that SPORK produced a merge with a lesser diff size in 1298 out of 1366 cases.

The small diff sizes for SPORK's merges can largely be attributed to source code reprinting, as explained in Section 5.5, as well as its ability to guess indentation style. JDIME appears to always print with an indentation of two spaces, and rarely captures formatting well even when the indentation happens to be correct.

SPORK's current reprinting implementation relies on entire methods in the merge stemming from single revisions. In cases where most methods were altered by the merge, JDIME often produced smaller diff sizes, which explains why SPORK is not better in all cases. In fact, disabling reprinting appears to give JDIME a substantial advantage. This is mostly due to the fact that SPOON's default pretty-printer often inserts redundant parentheses around expressions, which has been noted to be a problem in other applications as well [70].

A one-sided Wilcoxon signed-rank test was applied to the results to determine which tool produced smaller diff sizes overall. $H0_3$ was adopted as the null hypothesis, as defined below, and H3 was adopted as the alternative hypothesis.

> $H0_3$: There is no difference between the GITDIFF sizes of file merges produced by SPORK and JDIME
>
> H3: SPORK preserves formatting to a greater extent than JDIME does

The test yielded a p-value of 5.2e-208, meaning that the null hypothesis was rejected. Therefore, the alternative hypothesis that SPORK preserves formatting to a greater extent than JDIME does was instead accepted.

7.4 Reproduction of merges

This section presents results related to RQ4, which is recapitulated below.

> RQ4: How does SPORK compare to JDIME in terms of producing merges that are semantically equivalent to the merges committed by developers?

Out of the 890 merge scenarios, only 5 are such that no expected class files could be located. These 5 scenarios yielded no data for the experiment, leaving 885 viable merge scenarios comprising 1734 file merges.

Table 7.1: *Contingency table showing the amount of source files for which JDIME's and SPORK's merges yield the same or opposite results*

	JDIME fail	JDIME success
SPORK fail	621	176
SPORK success	154	783

Table 7.2: *Contingency table showing the amount of merge scenarios for which JDIME's and SPORK's merges yield the same or opposite results*

	JDIME fail	JDIME success
SPORK fail	235	81
SPORK success	70	499

The first measure is the amount of reproduced source files, as defined in Section 6.4.2. Out of the 1734 source files, SPORK reproduced 937 while JDIME reproduced 959. Table 7.1 shows the extent to which SPORK and JDIME reproduced the same source files. The main diagonal shows the amount of source files for which both tools produced merges that were judged equally, and the antidiagonal shows the amount of discordant cases.

McNemar's test was applied to the results to determine if either of the discordant cases is more likely to occur than the other. $H0_4$ was adopted as the null hypothesis, as defined below, and H4 was adopted as the alternative hypothesis.

> $H0_4$: It is equally likely that SPORK reproduces a merge that JDIME does not, as it is that JDIME reproduces a merge that SPORK does not
>
> H4: SPORK and JDIME differ in the extent to which they produce merges that are semantically equivalent to those committed by developers

The test yielded a p-value of 0.225, meaning that the null hypothesis could not be discarded. As such, tool performance is not distinguishable.

The second measure is the amount of reproduced merge scenarios. Out of a total of 885 merge scenarios, JDIME reproduced 580 scenarios while SPORK reproduced 569. Table 7.2 shows the extent to which SPORK and JDIME reproduced the same merge scenarios. The interpretation of the diagonals is the same as that for Table 7.1.

McNemar's test was applied again, with the same null and alternative hypotheses as in the previous test. The test yielded a p-value of 0.371, meaning that the null hypothesis could not be discarded. Again, tool performance is not distinguishable.

It is clear that SPORK and JDIME mostly perform the same, but that on occasion one tool can successfully perform a merge that the other cannot. Based on the data, there is however no conclusive evidence that either tool is overall more correct than the other.

7.5 Merge failures

Out of the total of 1740 file merges, SPORK failed to compute a merge for 34 cases due to crashes. The most prominent cause for failures was parse errors in SPOON, accounting for 15 failures. All but 1 of these originated from a single project, and for most of them SPOON built an empty tree for unknown reasons. Of the remaining failures, 6 were caused by failures in the fallback mechanisms, 6 more were caused by various other failures in conflict handling, and an additional 7 were caused by miscellaneous bugs in SPORK.

On the same dataset, JDIME failed to compute 23 merges. Only 7 of these failures were crashes, the remaining 16 being timeouts[7]. The 16 file merges that caused timeouts originated from 9 different projects. The remaining 7 failures were all caused by JDIME assigning too many or too few children to an AST node, which is likely a bug in JDIME.

Only 2 out of the 57 failures across both tools occurred on overlapping files. In the case of SPORK, these were parsing errors, and in the case of JDIME they were part of the 7 failures that were not timeouts.

[7] >300 second running time

Chapter 8

Discussion

This section presents a discussion of the results presented in Chapter 7. Section 8.1 discusses findings from the development of SPORK and the experimental results. Section 8.2 then details threats to the validity of the experimental results and Section 8.3 presents suggestions for future research. Section 8.4 concludes the chapter with a discussion of ethical considerations for the thesis.

8.1 Findings

The results indicate that SPORK matches or exceeds JDIME's performance in most respects, only objectively falling behind in the size of conflict hunks. First of all, SPORK performs significantly better than JDIME does in the above-median running times and is faster overall. This is an important finding, as the fact that SPORK is overall faster than JDIME is a strong indication that move-enabled structured merge is viable in terms of running time. JDIME is clearly faster for small merges, possibly due to its faster parser, while in the median case, neither tool can be said to be faster than the other. As SPORK is not heavily optimized there is room for further improvement, but making notable improvements to the below-median running times likely requires performance enhancements in SPOON's parsing, rather than in SPORK itself. Parallelizing the parsing is however one avenue worth exploring that does not necessarily require alterations to SPOON. However, given that most merge scenarios that require file merging only require it for 1 or 2 files, the sub-second variations in below-median running times between SPORK and JDIME are unlikely to amount to much of a difference in general. As such, improving upon the below-median running times may not be all too important. Limiting the larger running times is arguably more important, as having to wait 10 seconds or 100 seconds for a

merge is a noticeable difference.

It is possible that SPORK's tighter running times are due to SPORK-3DM scaling better than JDIME's merge algorithm. However, there are many other differences between the two merge tools that could play a role, and the manual spot checks are far from sufficient to draw any definitive conclusions. In particular, the differing ASTs likely impact running time to some extent. The spot checks indicate that JDIME's AST is faster to build from source code, but this does not imply that it is also faster to traverse, or that building the merged tree is faster. It is also likely that GUMTREE has different running time characteristics than JDIME's diff algorithm. This uncertainty highlights the fact that the results reflect the tools as the sum of their parts, rather than the individual pieces, and attributing differences between the tools to any individual component should be done conservatively.

The differing parsers and ASTs definitely have impact in areas other than running time. As noted in Section 7.1.1, SPORK has a tendency to encounter conflicts due to SPOON resolving implicit elements inconsistently. We have throughout the development of SPORK encountered multiple issues with implicit elements causing everything from crashes, to incorrectly printed conflicts, to conflicts that are reported but not printed. Many of these issues required fixes in the SPOON library, and we ended up contributing 7 patches to SPOON[1].

The results show that SPORK is significantly better at preserving formatting than JDIME is. It is evident that this is due to SPORK's source code reprinting and indentation detection, two features which JDIME lacks. Both of these features were developed as a reaction to SPORK's poor formatting preservation in early testing, where it performed far worse than JDIME did. Preserving source code formatting is undeniably important, and this was made even more prominent in preliminary test runs of the bytecode comparison experiment. In these test runs, Both SPORK and JDIME failed to build all merges in several projects due to failing style checks. Style checks were later disabled as formatting was not interesting to that particular experiment, but it served as a clear indication that formatting is important to developers.

When it comes to conflict handling, the results must be interpreted carefully. While SPORK does produce significantly fewer conflict hunks, we cannot confidently say if this is due to it properly resolving conflicts, or if it just fails to handle them. For example, as explained in Section 5.6.4, SPORK currently ignores delete/edit conflicts, while JDIME does not, and we also found a small

[1] `https://github.com/inria/spoon/commits?author=slarse&after=fa343e79b6e523bb938dcd4560b38d27c184adf9+0`

amount of conflicts that SPORK detected but failed to print as conflict hunks. In addition, given that a single bug in SPORK's line-based merge caused 45 extra conflict hunks, it is not unreasonable for some other bug to have the opposite effect. While fewer conflict hunks looks better on paper, a more thorough analysis of false positives and negatives as proposed by Cavalcanti et al. [19] is required to draw any definitive conclusions. The results that show that JDIME produces smaller conflict hunks are more reliable, and the spot checks indicate that this is indeed the case. Overall, JDIME's conflict handling appears to be more reliable than SPORK's, which requires refinement.

The results in Section 7.4 indicate that JDIME and SPORK are on par when it comes to reproducing merges semantically. Although no tool can be said to outperform the other overall, it is interesting to note that there are a large amount of discordant cases. This indicates that both tools are able to handle situations that the other cannot, which can be seen as corroborating the proposition by Menezes et al. [58] that no single merge tool can deal with all situations.

There is however a notable shortcoming with approximating semantic equality with a bytecode comparison, namely that the latter is far stricter. While two class files having equivalent bytecode does conclusively prove that they are semantically equivalent[2], the converse is not true in theory. That is to say, two class files that have differing bytecode may very well be semantically identical. For example, consecutive operations may be commutative, such that even when ordered differently in the bytecode, they have the same effect. We have taken every step known to us to minimize non-semantic bytecode differences, but the problem is undeniably still present.

Finally, move-enabled diff and merge has both benefits and drawbacks. The most tangible benefit that can be discerned from the results is how SPORK can resolve merges involving renamings automatically. Although more involved examples involving moves of code blocks can be easily constructed, no such examples were found in the results[3]. Also, while SPORK encounters fewer conflicts than JDIME does, we cannot confidently attribute this to the move-enabled diff and merge algorithms due to the large amount of differences between the merge tools.

The largest drawback of move-enabled merge is the introduction of move conflicts. Not only can they be difficult to detect, it is also not clear how to express a move conflict in textual form, or if it even is desirable to do so.

[2]Barring bugs in SOOTDIFF

[3]This is not to say that such examples are definitely absent from the results, as the manual inspection of the results consisted of spot checks

In addition, it has proven difficult to distinguish between move conflicts and delete conflicts in SPORK-3DM, which makes taking appropriate action even more difficult.

8.2 Threats to validity

There are three primary threats to validity: the diversity and representativeness [61] of the dataset, the execution of the experiments and the tuning of the merge tools.

The diversity of the dataset is one of the most critical aspects when it comes to the extent to which the results generalize. The dataset exhibits a large amount of variety in terms of project domains, sizes, amount of core contributors and stars, as can be seen in the project summary in Appendix A. While this makes it probable that the results generalize well within the JAVA programming language, there are limiting factors. It is for example possible that closed source projects have different merge practices from the open source projects in the dataset, and so generalizing beyond open source is unsubstantiated. The extensive filtering described in Section 6.2 may also have an impact. For example, it is possible that projects that use a build system other than MAVEN have some particular merging practices. Even limiting the analysis to open source projects on GITHUB may mean that the results do not generalize to other platforms, such as GITLAB.

An important factor that hurts representativeness is the amount of merges that can be extracted with the methodology given in Chapter 6, the *discoverable* population, contrasted with the amount of merges that have actually taken place, the *true* population. For example, file merges may occur during rebasing, but as rebasing rewrites the version history [1], any such merges cannot be found in the merge extraction. Rebasing and squashing may also hide previously performed merges. Additionally, recursive merges were categorically excluded. It is therefore important to note that the experiments sampled from a subset of all merge scenarios that actually occurred within the selected projects.

Furthermore, as the amount of merge scenarios from each project were restricted such that no project could contribute more than 15 merge scenarios, the dataset most certainly is not representative of the population of merge scenarios in terms of proportions. This is however necessary, as a) the true population cannot be determined, and b) the extensive filtering performed to find buildable scenarios means that the population eligible for sampling is not representative even of the discoverable population. By limiting the amount of

merge scenarios from each project to 15 scenarios, amounting to 1.68% of the total of 890 scenarios, the dataset is instead representative of the population of projects, rather than the population of merge scenarios. The same does not apply directly to file merges however, as a single merge scenario can yield many file merges. The project with the most file merges contributed 122 file merges to the experiment, which amounted to 7% of the total 1740 file merges. As SPORK and JDIME perform differently in different projects, this may impact the generalizability of the results on individual file merges.

As for the execution of the experiments, there are two significant threats to validity. First, the individual running time measurements are not reliable due to the inherent variability of measuring wall time on a computer. Each file merge should ideally be timed multiple times for each merge tool. Second, the bytecode comparison experiment described in Section 6.4 is both complex, making the possibility for implementation errors relatively high, and also relies on building projects, which makes it highly environment dependent. In fact, it is even time sensitive, as the dependencies of the projects in the dataset that can be accessed today may not be available tomorrow. This may make fully reproducing the experiment with the same dataset difficult, or even impossible. While this is not a direct threat to validity, it is a threat to future validation of the results.

Finally, both SPORK and JDIME can be tuned in various ways. The results are only valid for one set of tuning parameters, and these are not necessarily optimal. The experiments were executed with JDIME's default settings. This for example means that its lookahead heuristics for identifying renamed methods and shifted code were not enabled, which if enabled could have helped avoid some conflicts at the cost of increased running time [14]. To summarize, it is possible that with different tuning, the results would have turned out differently.

8.3 Future research

There are two important aspects to further improve upon in SPORK. The most important part to improve upon is the handling of delete and move conflicts. The fact that SPORK currently can only deal with these via the local fallback, which is line-based, limits its capabilities to perform structured merging. The local fallback and the conservative left-to-right matching were the primary reasons for SPORK producing larger conflict hunk sizes than JDIME, and both of these are consequences of an inability to deal with delete and move conflicts.

The second point of improvement is to refine SPORK's pretty-printing pro-

cess. The results on the GITDIFF sizes show that SPORK's implementation of source code reprinting has a large impact on code formatting preservation, even though it only acts on single-revision comments, fields and methods. Preliminary tests show that increasing the granularity to the level of individual expressions and statements further reduces the GITDIFF size significantly, especially in file merges where many methods are edited.

There is also a need to rewrite SPORK's pretty-printer, which is currently implemented on top of SPOON's default printer. Due to implementation details of SPOON's printer, SPORK's extension of it is not always called when a node is to be printed, which results in corner cases where a conflict is simply cannot be printed. It is likely that a complete rewrite of the printer is required to fully solve the problem.

Another interesting avenue of research is to attempt to evaluate how move-enabled merge compares to non-move-enabled equivalents. As SPORK-3DM's merge behavior is decided by the tree matchings computed by the diff algorithm, simply swapping in a non-move-enabled diff algorithm makes SPORK as a whole non-move-enabled. For example, it is possible to implement JDIME's tree matching algorithm in the GUMTREE library, making switching between a move-enabled and a non-move-enabled matching in SPORK as simple as providing a different argument to GUMTREE's matcher factory. Due to SPORK-3DM simply adapting to whatever tree matchings it is provided, and the GUMTREE library being so generic, SPORK provides an excellent platform to compare move-enabled and non-move-enabled structured merge.

Due to the difficulties encountered in validating the general correctness of the merge tools, we find that there is a great need for independently developed tooling that can verify the semantic correctness of a merge. For example, in Section 3.2 we discuss SAFEMERGE, which we think could be used for this purpose, but is neither well enough maintained nor validated to be used in its current state.

There is also a great need for further work on tooling for general merge tool evaluation. While MERGANSER [60] is a step in the right direction, it did not fulfill our needs in terms of customizability, making it more efficient for us to develop our own tooling. MERGANSER is an off-the-shelf CLI for running an experiment, and it sacrifices customizability for ease of use. In the case of this thesis, a highly flexible framework for setting up customized experiments would have been of greater use. Much like a single merge technique is unlikely to be the optimal performer in all cases, a single merge testing application being able to handle all forms of merge experiments without modification is an unlikely proposition.

8.4 Ethical considerations

There are two aspects of the thesis that in general have relevant ethical considerations: data collection and energy consumption. Data collection is a controversial topic in computer science, but all of the data used in the experiments is purposefully published on GITHUB. In addition, everything but source files was discarded, which further reduces the likelihood that the dataset contains personal or otherwise sensitive information.

As for energy consumption, the added computational complexity of structured merge is guaranteed to multiply the energy needed to compute a merge, as compared to unstructured merge. However, compared to other processes in software development, such as integration and system tests, the impact should be miniscule. In fact, better merging can lead to energy savings in the long run, as it should lead to less manual intervention and human error. The experiments did draw a lot of power, but even with the liberal estimation that the test system pulled 150W of power from the wall socket for 7 full days, it only amounts to around 25kWh[4]. This is not a lot of energy for a scientific experiment. It should be noted that there is a hidden energy cost in pulling resources from remote repositories for the experiments, but due to how MAVEN caches data, it should be relatively small. Overall, the work presented in this thesis has little negative impact, with the energy consumption for the experiments being the only objectively negative aspect.

[4] $(7 * 24[h]) * 150[W] = 25,200[Wh]$

Chapter 9

Conclusions

In this thesis, we implemented and evaluated SPORK, a move-enabled structured merge tool for the JAVA programming language. By empirical evaluation, we showed that SPORK provides similar performance to the state-of-the-art structured merge tool JDIME. SPORK also makes significant improvements on above-median running times and preservation of source code formatting, two factors that are likely important for wider adoption of structured merge among developers.

To the best of our knowledge, SPORK represents the first attempt to implement a fully structured merge tool for JAVA using move-enabled diff and merge algorithms. The results show that SPORK's move-enabled algorithms allows it deal with renamings, and clearly demonstrate that the GUMTREE diff algorithm and the SPORK-3DM merge algorithm are fast and accurate enough to be used in a merge tool for JAVA. SPORK and the GUMTREE library also provide a viable platform to further evaluate different tree matching algorithms in a merge tool, without requiring other parts of the merge tool to be modified.

It is also apparent that move conflicts introduce several challenges that do not exist when dealing with a merge that lacks move detection capabilities. Move conflicts are challenging to identify in SPORK-3DM, and even when properly identified, there is no obvious way to express a move conflict in text. Much due to unresolved issues with move conflicts, SPORK's conflict handling is not as robust as JDIME's is.

We also developed a comprehensive benchmarking suite that can either be reused as-is, or be expanded for other purposes or programming languages. Whereas previous research has mostly focused on investigating conflict behavior, our approach also considers the general performance of merge tools when conflicts are absent. This can aid both in developing new merge tools, as well

as in evaluating existing ones.

In conclusion, the results of this thesis are promising for the prospect of move-enabled structured merge. Although SPORK is unpolished in certain aspects, it clearly contends with the state of the art in structured merging, and there are likely significant gains to be made from further development.

Bibliography

[1] Christian Bird et al. “The promises and perils of mining git”. In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009. DOI: 10.1109/msr.2009.5069475.

[2] Christian Bird and Thomas Zimmermann. “Assessing the value of branches with what-if analysis”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*. ACM Press, 2012. DOI: 10.1145/2393596.2393648.

[3] Bram Adams and Shane McIntosh. “Modern Release Engineering in a Nutshell – Why Researchers Should Care”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016. DOI: 10.1109/saner.2016.108.

[4] T. Mens. “A state-of-the-art survey on software merging”. In: *IEEE Transactions on Software Engineering* 28.5 (2002), pp. 449–462. DOI: 10.1109/tse.2002.1000449.

[5] Yuriy Brun et al. “Early Detection of Collaboration Conflicts and Risks”. In: *IEEE Transactions on Software Engineering* 39.10 (2013), pp. 1358–1375. DOI: 10.1109/tse.2013.28.

[6] Bakhtiar Khan Kasi and Anita Sarma. “Cassandra: Proactive Conflict Minimization through Optimized Task Scheduling”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, 732–741. ISBN: 9781467330763.

[7] Olaf Leßenich, Sven Apel, and Christian Lengauer. “Balancing precision and performance in structured merge”. In: *Automated Software Engineering* 22.3 (2014), pp. 367–397. DOI: 10.1007/s10515-014-0151-5.

[8] Bernhard Westfechtel. “Structure-oriented merging of revisions of software documents”. In: *Proceedings of the 3rd international workshop on Software configuration management* -. ACM Press, 1991. DOI: 10.1145/111062.111071.

[9] Olaf Leßenich. “Adjustable Syntactic Merge of Java Programs”. MA thesis. Universität Passau, 2012.

[10] Alberto Trindade Tavares et al. “Semistructured Merge in JavaScript Systems”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019. DOI: 10.1109/ase.2019.00098.

[11] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. “Evaluating and improving semistructured merge”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–27. DOI: 10.1145/3133883.

[12] Jean-Rémy Falleri et al. “Fine-grained and accurate source code differencing”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*. ACM Press, 2014. DOI: 10.1145/2642937.2642982.

[13] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. “Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts”. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019. DOI: 10.1109/saner.2019.8668012.

[14] Olaf Leßenich et al. “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*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 543–553. ISBN: 978-1-5386-2684-9. URL: http://dl.acm.org/citation.cfm?id=3155562.3155631.

[15] Raymond P.L. Buse and Westley R. Weimer. “A metric for software readability”. In: *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*. ACM Press, 2008. DOI: 10.1145/1390630.1390647.

[16] Tancred Lindholm. “A three-way merge for XML documents”. In: *Proceedings of the 2004 ACM symposium on Document engineering - DocEng '04*. ACM Press, 2004. DOI: 10.1145/1030397.1030399.

[17] Renaud Pawlak et al. “SPOON: A library for implementing analyses and transformations of Java source code”. In: *Software: Practice and Experience* 46.9 (2015), pp. 1155–1179. DOI: 10.1002/spe.2346.

[18] Fengmin Zhu and Fei He. “Conflict resolution for structured merge via version space algebra”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (Oct. 2018), pp. 1–25. DOI: 10.1145/3276536. URL: https://doi.org/10.1145/3276536.

[19] Guilherme Cavalcanti et al. “The Impact of Structure on Software Merging: Semistructured versus Structured Merge”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1002–1013.

[20] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. “Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment”. In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015. DOI: 10.1109/esem.2015.7321191.

[21] Sven Apel et al. “Semistructured merge”. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. ACM Press, 2011. DOI: 10.1145/2025113.2025141.

[22] Konrad Hinsen, Konstantin Läufer, and George K. Thiruvathukal. “Essential Tools: Version Control Systems”. In: *Computing in Science & Engineering* 11.6 (2009), pp. 84–91. DOI: 10.1109/mcse.2009.194.

[23] Wouter Swierstra and Andres Löh. “The Semantics of Version Control”. In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2014. Portland, Oregon, USA: Association for Computing Machinery, 2014, 43–54. ISBN: 9781450332101. DOI: 10.1145/2661136.2661137. URL: https://doi-org.focus.lib.kth.se/10.1145/2661136.2661137.

[24] Bryan O'Sullivan. “Making sense of revision-control systems”. In: *Communications of the ACM* 52.9 (2009), p. 56. DOI: 10.1145/1562164.1562183.

[25] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

[26] Caius Brindescu et al. “How do centralized and distributed version control systems impact software changes?” In: *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, 2014. DOI: 10.1145/2568225.2568322.

[27] Kıvanç Muşlu et al. “Transition from centralized to decentralized version control systems: a case study on reasons, barriers, and outcomes”. In: *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, 2014. DOI: 10.1145/2568225.2568284.

[28] Brian de Alwis and Jonathan Sillito. “Why are software projects moving from centralized to decentralized version control systems?” In: *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. IEEE, 2009. DOI: 10.1109/chase.2009.5071408.

[29] Shaun Phillips, Jonathan Sillito, and Rob Walker. “Branching and merging”. In: *Proceeding of the 4th international workshop on Cooperative and human aspects of software engineering - CHASE '11*. ACM Press, 2011. DOI: 10.1145/1984642.1984645.

[30] Eugene W. Myers. “AnO(ND) difference algorithm and its variations”. In: *Algorithmica* 1.1-4 (1986), pp. 251–266. DOI: 10.1007/bf01840446.

[31] Sudarshan S. Chawathe et al. “Change detection in hierarchically structured information”. In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD '96*. ACM Press, 1996. DOI: 10.1145/233269.233366.

[32] Beat Fluri et al. “Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction”. In: *IEEE Transactions on Software Engineering* 33.11 (2007), pp. 725–743. DOI: 10.1109/tse.2007.70731.

[33] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and merging files*. 2002. URL: https://docs.freebsd.org/info/diff/diff.pdf (visited on 06/08/2020).

[34] Danny Dig et al. “Refactoring-Aware Configuration Management for Object-Oriented Programs”. In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007. DOI: 10.1109/icse.2007.71.

[35] T. Apiwattanapong, A. Orso, and M.J. Harrold. “A differencing algorithm for object-oriented programs”. In: *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* IEEE, 2004. DOI: 10.1109/ase.2004.1342719.

[36] Dimitar Asenov et al. “Precise Version Control of Trees with Line-Based Version Control Systems”. In: *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, 2017, pp. 152–169. DOI: 10.1007/978-3-662-54494-5_9.

[37] Sebastian Rönnau, Christian Pauli, and Uwe M. Borghoff. “Merging changes in XML documents using reliable context fingerprints”. In: *Proceeding of the eighth ACM symposium on Document engineering - DocEng '08*. ACM Press, 2008. DOI: 10.1145/1410140.1410151.

[38] Sebastian Rönnau, Geraint Philipp, and Uwe M. Borghoff. “Efficient and reliable merging of XML documents”. In: *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09*. ACM Press, 2009. DOI: 10.1145/1645953.1646326.

[39] Sebastian Rönnau and Uwe M. Borghoff. “XCC: change control of XML documents”. In: *Computer Science - Research and Development* 27.2 (2010), pp. 95–111. DOI: 10.1007/s00450-010-0140-2.

[40] Sven Apel et al. “Semistructured Merge in Revision Control Systems.” In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. 2010, pp. 13–19.

[41] Bo Shen et al. “IntelliMerge: a refactoring-aware software merging technique”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–28. DOI: 10.1145/3360596.

[42] *Resolve Conflicts*. JetBrains s.r.o. 2019. URL: https://www.jetbrains.com/help/idea/resolving-conflicts.html (visited on 01/27/2019).

[43] Antti Nieminen. “Real-time Collaborative Resolving of Merge Conflicts”. In: *Proceedings of the 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE, 2012. DOI: 10.4108/icst.collaboratecom.2012.250435.

[44] Catarina Costa et al. “TIPMerge: recommending experts for integrating changes across branches”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. ACM Press, 2016. DOI: 10.1145/2950290.2950339.

[45] Yuichi Nishimura and Katsuhisa Maruyama. “Supporting Merge Conflict Resolution by Using Fine-Grained Code Change History”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016. DOI: 10.1109/saner.2016.46.

[46] Fengmin Zhu, Fei He, and Qianshan Yu. “Enhancing Precision of Structured Merge by Proper Tree Matching”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019. DOI: 10.1109/icse-companion.2019.00117.

[47] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. “Verified three-way program merge”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–29. DOI: 10.1145/3276535.

[48] A. Sarma, D. F. Redmiles, and A. van der Hoek. “Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes”. In: *IEEE Transactions on Software Engineering* 38.4 (2012), pp. 889–908. DOI: 10.1109/tse.2011.64.

[49] Lile Hattori and Michele Lanza. “Syde: A Tool for Collaborative Software Development”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press, 2010. DOI: 10.1145/1810295.1810339.

[50] Mário Luís Guimarães and António Rito Silva. “Improving Early Detection of Software Merge Conflicts”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE Press, 2012, 342–352. ISBN: 9781467310673.

[51] Paola Accioly et al. “Analyzing Conflict Predictors in Open-source Java Projects”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR ’18. Gothenburg, Sweden: ACM, 2018, pp. 576–586. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196437. URL: http://doi.acm.org/10.1145/3196398.3196437.

[52] Olaf Leßenich et al. “Indicators for merge conflicts in the wild: survey and empirical study”. In: *Automated Software Engineering* 25.2 (2017), pp. 279–313. DOI: 10.1007/s10515-017-0227-0.

[53] Klissiomara Dias, Paulo Borba, and Marcos Barreto. “Understanding Predictive Factors for Merge Conflicts”. In: *Information and Software Technology* (2020), p. 106256. DOI: 10.1016/j.infsof.2020.106256.

[54] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. “Predicting Merge Conflicts in Collaborative Software Development”. In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019. DOI: 10.1109/esem.2019.8870173.

[55] Thomas Zimmermann. “Mining Workspace Updates in CVS”. In: *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. IEEE, 2007. DOI: 10.1109/msr.2007.22.

[56] Shane McKee et al. “Software Practitioner Perspectives on Merge Conflicts and Resolutions”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017. DOI: 10.1109/icsme.2017.53.

[57] Nicholas Nelson et al. “The life-cycle of merge conflicts: processes, barriers, and strategies”. In: *Empirical Software Engineering* 24.5 (2019), pp. 2863–2906. DOI: 10.1007/s10664-018-9674-x.

[58] Gleiph Ghiotto Lima Menezes et al. “On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub”. In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1. DOI: 10.1109/tse.2018.2871083.

[59] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[60] Moein Owhadi-Kareshk and Sarah Nadi. “Scalable Software Merging Studies with MERGANSER”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019. DOI: 10.1109/msr.2019.00084.

[61] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. “Diversity in software engineering research”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, 2013. DOI: 10.1145/2491411.2491415.

[62] Nuthan Munaiah et al. “Curating GitHub for engineered software projects”. In: *Empirical Software Engineering* 22.6 (2017), pp. 3219–3253. DOI: 10.1007/s10664-017-9512-6.

[63] Oscar H. Ibarra and Brian S. Leininger. “The complexity of the equivalence problem for straight-line programs”. In: *Proceedings of the twelfth annual ACM symposium on Theory of computing - STOC '80*. ACM Press, 1980. DOI: 10.1145/800141.804675.

[64] Andreas Dann, Ben Hermann, and Eric Bodden. “SootDiff: bytecode comparison across different Java compilers”. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis - SOAP 2019*. ACM Press, 2019. DOI: 10.1145/3315568.3329966.

[65] John W. Pratt. “Remarks on Zeros and Ties in the Wilcoxon Signed Rank Procedures”. In: *Journal of the American Statistical Association* 54.287 (1959), pp. 655–667. DOI: 10.1080/01621459.1959.10501526.

[66] Edward E. Cureton. “The Normal Approximation to the Signed-Rank Sampling Distribution When Zero Differences are Present”. In: *Journal of the American Statistical Association* 62.319 (1967), pp. 1068–1069. DOI: 10.1080/01621459.1967.10500917.

[67] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature Methods* 17.3 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[68] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. “The NumPy Array: A Structure for Efficient Numerical Computation”. In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30. DOI: 10.1109/mcse.2011.37.

[69] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. SciPy, 2010. DOI: 10.25080/majora-92bf1922-00a.

[70] Haris Adzemovic. “A template-based approach to automatic program repair of Sonarqube static warnings”. MA thesis. KTH, School of Electrical Engineering and Computer Science (EECS), 2020, p. 65.

Appendix A

Projects used in experiments

Table A.1 shows an overview of the projects used in the experimental evaluation of `Spork` and `JDime`. Note that these values are dynamic and change over time, so there may be discrepancies between Table A.1 and what is found in the `Reaper` dataset, especially when it comes to stars and core contributors.

Table A.1: *Overview of the experiment dataset. The URL for any one of the projects is* `https://github.com/<PROJECT>`*. See the table footer for a column legend.*

Project	MS	FM	CLOC	Stars	CC
aaberg/sql2o	5	10	13078	986	3
addthis/stream-lib	11	15	13447	2141	13
afawcett/githubsfdeploy	1	1	1106	146	4
agoncal/agoncal-application-petstore-ee6	1	1	4231	199	3
alibaba/fastjson	15	16	240170	21720	6
amazon-archives/aws-dynamodb-session-tomcat	1	3	2031	96	9
apache/avro	1	1	123458	1591	21
apache/commons-codec	4	5	40905	257	4
apache/commons-imaging	1	1	67626	209	6
apache/curator	15	45	91451	2148	3
BaseXdb/basex	15	65	257527	471	9
bennidi/mbassador	1	4	8923	854	4
biojava/biojava	15	24	293155	361	12
Bluelock/camel-spring-amqp	6	11	2641	59	4
brianfrankcooper/YCSB	14	14	39987	3067	38

MS = Merge scenarios, FM = File merges, CLOC = commented lines of Java code, Stars = GitHub stars, CC = core contributors. CLOC, Stars and CC are calculated as of the 7th of June 2020

Project	MS	FM	CLOC	Stars	CC
broadgsa/gatk	3	5	204458	272	18
BroadleafCommerce/BroadleafCommerce	15	38	337371	1364	18
Bukkit/Bukkit	4	5	66404	1970	16
caelum/vraptor4	15	19	46543	341	4
cbeust/testng-eclipse	2	2	25257	161	2
cdapio/tephra	2	6	57530	157	5
cglib/cglib	3	4	22682	3516	3
Comcast/jrugged	3	7	16626	236	4
coverity/coverity-security-library	1	1	2322	169	4
damianszczepanik/cucumber-reporting	15	27	13368	396	2
dankurka/mgwt	9	15	40433	92	2
davidB/scala-maven-plugin	8	12	5673	448	5
dcm4che/dcm4che	11	15	97843	621	4
eclipse-vertx/vert.x	11	20	175125	11088	5
EngineHub/CommandHelper	7	17	223046	88	5
FasterXML/jackson-datatype-guava	3	4	5807	69	3
flyingsaucerproject/flyingsaucer	8	13	100305	1473	4
forcedotcom/phoenix	15	30	145717	563	8
forcedotcom/wsc	9	20	26801	218	20
geosolutions-it/geoserver-manager	2	2	25527	149	9
google/closure-compiler	15	18	575621	5528	40
greenmail-mail-test/greenmail	2	6	24374	377	2
hector-client/hector	15	24	46212	651	8
hub4j/github-api	15	37	42139	667	12
HubSpot/dropwizard-guice	1	1	1392	264	4
HubSpot/Singularity	15	31	130299	776	9
iipc/openwayback	15	38	82152	340	4
internetarchive/heritrix3	11	13	90265	1741	6
jalkanen/speed4j	1	1	3323	95	1
javafx-maven-plugin/javafx-maven-plugin	9	9	4962	714	4
jcabi/jcabi-github	15	38	68509	232	17
jeluard/semantic-versioning	2	2	6696	135	2
jenkinsci/email-ext-plugin	15	24	15585	277	17
jenkinsci/jclouds-plugin	9	12	9073	67	6
jenkinsci/jira-plugin	12	27	15075	142	17
jenkinsci/performance-plugin	4	4	16443	167	11
jenkinsci/promoted-builds-plugin	12	17	10049	94	9

MS = Merge scenarios, FM = File merges, CLOC = commented lines of Java code, Stars = GitHub stars, CC = core contributors. CLOC, Stars and CC are calculated as of the 7th of June 2020

Project	MS	FM	CLOC	Stars	CC
jitsi/jitsi-videobridge	4	12	29517	1861	8
jitsi/libjitsi	8	10	215121	478	6
jmeter-maven-plugin/jmeter-maven-plugin	2	3	6670	332	2
jmock-developers/jmock-library	2	2	15454	119	4
jmxtrans/jmxtrans	7	10	30681	1598	27
joel-costigliola/assertj-core	15	122	353593	1759	13
jsevellec/cassandra-unit	4	8	2837	404	7
kairosdb/kairosdb	5	5	63425	1540	2
kohsuke/args4j	8	13	8537	679	4
komoot/photon	10	17	5582	975	6
larsga/Duke	7	12	25230	579	1
libgdx/box2dlights	2	2	4533	206	8
linkedin/camus	14	14	13832	850	12
liquibase/liquibase-hibernate	3	3	5398	182	8
lucidworks/spark-solr	5	6	6824	387	6
Mach5/supersonic	1	2	57885	220	3
magefree/mage	15	23	1782052	1060	15
marceloverdijk/lesscss-maven-plugin	1	1	1524	176	5
maven-nar/nar-maven-plugin	15	67	53937	211	10
mboudreau/Alternator	10	18	42612	77	3
mgodave/barge	2	14	19997	86	2
mongojack/mongojack	2	10	16990	231	7
mvel/mvel	12	16	82004	693	3
mybatis/mybatis-3	15	28	99846	13625	6
NGDATA/hbase-indexer	3	11	29419	334	4
ning/compress	1	2	9218	218	2
nishio-dens/bitbucket-pullrequest-builder-plugin	10	16	3694	122	12
notnoop/java-apns	8	14	10779	1748	5
OAuth-Apis/apis	4	4	12277	593	3
ontop/ontop	15	55	207180	350	7
opendatalab-de/geojson-jackson	3	20	1841	178	4
OpenHFT/Chronicle-Map	10	14	52715	1956	3
OpenHFT/Java-Thread-Affinity	2	3	6671	1036	6
openshift/openshift-java-client	4	19	30864	64	5
orientechnologies/spring-data-orientdb	1	1	7230	125	7
OWASP/SecurityShepherd	1	15	64642	820	7
pazone/ashot	1	2	3287	465	7

MS = Merge scenarios, FM = File merges, CLOC = commented lines of Java code, Stars = GitHub stars, CC = core contributors. CLOC, Stars and CC are calculated as of the 7th of June 2020

Project	MS	FM	CLOC	Stars	CC
perf4j/perf4j	3	4	14256	262	3
Progether/JAdventure	10	19	5976	248	6
protostuff/protostuff	6	9	102640	1418	4
QSFT/Doradus	4	4	97383	202	4
quickfix-j/quickfixj	3	5	57647	573	7
rackerlabs/atom-hopper	15	18	18779	58	11
searchbox-io/Jest	15	19	29473	1947	19
siddhi-io/siddhi	15	34	205790	1045	24
simpligility/android-maven-plugin	15	18	86709	1028	13
skyscreamer/JSONassert	8	13	4840	717	2
spotify/docker-client	15	21	35068	1327	8
spotify/docker-maven-plugin	8	16	3788	2385	3
spotify/hdfs2cass	1	1	3995	75	8
spotify/helios	15	16	71553	2035	6
Spoutcraft/LegacyLauncher	15	18	13830	94	5
stanfordnlp/CoreNLP	1	1	788082	7242	9
stateless4j/stateless4j	6	19	4126	558	6
stormpath/stormpath-sdk-java	15	26	133248	220	8
Stratio/crossdata	15	29	3942	167	12
Stratio/Decision	7	8	22973	305	13
stratosphere/stratosphere	4	8	280025	189	9
taycaldwell/riot-api-java	2	3	17128	174	2
threerings/getdown	7	11	11609	387	6
tlrx/elasticsearch-test	1	2	5286	153	2
TooTallNate/Java-WebSocket	15	23	24306	6883	6
tumblr/jumblr	2	5	4693	278	4
versly/wsdoc	3	4	5089	59	3
WiQuery/wiquery	2	2	49363	62	4
xebia/Xebium	7	7	4408	75	3
yandex-qatools/htmlelements	3	3	7253	251	5

MS = Merge scenarios, FM = File merges, CLOC = commented lines of Java code, Stars = GitHub stars, CC = core contributors. CLOC, Stars and CC are calculated as of the 7th of June 2020

TRITA -EECS-EX-2020:648