Data Flow and Control Flow Analysis of Problematic Commits

Ward Muylaert

Dissertation submitted in fulfilment of the
requirement for the degree of Doctor of Sciences

April 2024

Promotor:
Prof. Dr. Coen De Roover, Vrije Universiteit Brussel, Belgium

Jury:
Prof. Dr. Viviane Jonckers, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Lynn Houthuys, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Jean-Rémy Falleri, Université de Bordeaux, France
Prof. Dr. Josep Silva, Universidad Politècnica de València, Spain
Prof. Dr. Kris Steenhaut, Vrije Universiteit Brussel, Belgium

Software Languages Lab
Department of Computer Science
Faculty of Sciences and Bioengineering Sciences
Vrije Universiteit Brussel

Deze uitgave is vrijgegeven onder Creative Commons Naamsvermelding 4.0 Internationaal (CC-BY 4.0). Je bent vrij om deze uitgave te delen en te bewerken onder de voorwaarde dat correcte naamsvermelding wordt toegepast.

This publication is released under Creative Commons Attribution 4.0 International (CC-BY 4.0). You are free to share and adapt this publication as long as proper attribution is given.

©2024 Ward Muylaert

Printed by
Crazy Copy Center Productions
Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel
Tel: +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

ISBN: 978-94-6494-822-6
NUR: 980
THEMA: UMZT

Abstract

When creating and maintaining programs, software developers make use of version control software to store different versions of the source code. Version control software uses commits as building blocks. Commits play a dual role: they represent a version of the software program at a certain point in time; they also represent the changes compared to the preceding version of the program.

In this dissertation, we investigate two types of commits that are harder for developers to understand. First, we consider *composite commits*. Composite commits group many unrelated changes. The changes target different tasks, such as fixing a bug or introducing a new feature. Besides being harder to understand, composite commits also prove harder for software developers to revert or integrate and for empirical researchers to analyse. Second, we consider *merge commits*. Versions of the program start to diverge when different developers work on different features or bug fixes. A merge commit recombines divergent versions. An overlap in source code or an interaction in behaviour demands a resolution from the software developer, requiring them to understand the source code of the different versions.

We propose an algorithmic foundation for tool support for these two types of problematic commits. Our first algorithm untangles composite commits using a *data flow driven* approach. Our algorithm considers a program dependence graph and fine-grained changes to the abstract syntax tree. The algorithm groups fine-grained changes according to the program dependence graph slices they belong to. Our second algorithm analyses merge commits using a *control flow driven* approach. It uses symbolic execution to gather path conditions of the different versions of the program. We define the program semantics in function of these path conditions. The path conditions are checked against rules that indicate presence of a semantic merge conflict.

We evaluate both algorithms. By analysing, refining, and using an established dataset of composite commits, we find our untangling algorithm can determine whether a commit is composite. The groups of fine-grained changes tend to be smaller than the commit's tasks, but stay within their boundaries. We evaluate our semantic merge conflict detection algorithm in two ways. First, we evaluate its correctness through mutation testing. Second, we evaluate it empirically by applying it to real-world merges. We discuss challenges in the empirical evaluation of semantic merge conflicts. Our evaluation shows that in specific cases our approach is a promising extension to existing mechanisms in semantic merge conflict detection.

Samenvatting

Bij het schrijven en onderhouden van programma's gebruiken softwareontwikkelaars versie controle software om verschillende versies van de broncode op te slaan. Versie controle software gebruikt commits als bouwstenen. Commits hebben een dubbele rol: ze representeren een versie van het programma op een zeker tijdspunt; ze representeren ook de veranderingen ten opzichte van de vorige versie van het programma.

In deze dissertatie onderzoeken we twee types commits die moeilijker verstaanbaar zijn voor ontwikkelaars. Eerst kijken we naar *samengestelde commits*. Samengestelde commits groeperen verschillende ongerelateerde veranderingen. De veranderingen behandelen verschillende taken, zoals het herstellen van een bug of het toevoegen van een eigenschap. Behalve moeilijker verstaanbaar te zijn, zijn samengestelde commits ook moeilijker voor ontwikkelaars om ongedaan te maken of te integreren en voor onderzoekers om te analyseren. Ten tweede behandelen we *samenvoegcommits*. Versies van het programma divergeren wanneer verschillende ontwikkelaars aan verschillende eigenschappen werken of bugs herstellen. Een samenvoegcommit voegt de gedivergeerde versies weer samen. Indien de broncode overlapt of het gedrag mekaar beïnvloedt, dan moet de ontwikkelaar een oplossing voorzien na eerst de broncode van de verschillende versies te begrijpen.

Wij stellen een algoritmische onderbouwing voor hulpmiddelondersteuning voor beide types van problematische commits voor. Ons eerste algoritme ontwart samengestelde commits aan de hand van een *dataverloop gebaseerde* aanpak. Ons algoritme gebruikt een programma afhankelijkheidsgraaf en fijnmazige veranderingen aan de abstracte syntaxboom. Het algoritme groepeert de fijnmazige veranderingen op basis van de stukken uit de programma afhankelijkheidsgraaf waar ze toe behoren. Ons tweede algoritme analyseert samenvoegcommits aan de hand van een *controleverloop gebaseerde* aanpak. Het gebruikt symbolische executie om padcondities van de verschillende versies van het programma te verzamelen. We definiëren de programmasemantiek in functie van deze padcondities. De padcondities worden vergeleken met regels die de aanwezigheid van een semantisch samenvoegsconflict aanduiden.

We evalueren beide algoritmes. Door het analyseren, verfijnen en gebruiken van een erkende dataset van samengestelde commits besluiten we dat ons ontwaralgoritme kan vaststellen of een commit samengevoegd is. De groepen fijnmazige veranderingen zijn over het algemeen kleiner dan de taken in de commit, maar blijven erdoor begrensd. We evalueren ons semantisch samenvoegsconflictalgoritme op twee manieren. Eerst evalueren we de correctheid aan de hand van mutatietesten. Ten tweede evalueren we het empirisch door het toe te passen op echte samenvoegcommits. We beschrijven ook uitdagingen bij het empirisch evalueren van semantische samenvoegsconflicten. Onze evaluatie laat zien dat onze aanpak in specifieke gevallen een veelbelovende uitbreiding is op bestaande mechanismen.

Acknowledgements

This work would not have been possible without the support of many people, be it academical, technical, or mental support.

Thanks to my promotor Prof. Dr. Coen De Roover. It drained blood, sweat, and tears out of the both of us, perhaps out of you more so than out of me, but we got there in the end.

Thanks to my jury for a fruitful discussion during the private defence and the valuable feedback that improved this text greatly: Profs. Drs. Jean-Rémy Falleri, Lynn Houthuys, Viviane Jonckers, Josep Silva, and Kris Steenhout. Reading a long scientific text is a time investment that cannot be understated.

Thanks to the people at the Software Languages Lab for office, lunch, and other discussions. Shout-out to Mathijs, for the many running discussions, Sam, our Christmas lunches and summer BBQs were clearly the best SOFT has ever seen, and Noah, with whom I shared an office and the highs and lows of our PhDs for many years. Special shout-out to my old office mate Reinout for the continuing enjoyable runs to a real gem and beyond, as well as for indulging a silly photo idea I had.

Ad fundum to the student groups I have been a part of, WK and BK. They have made me meet many people over the years and their events formed a welcome distraction when I needed it.

Ik kan “de mannen” van Mabo niet vergeten: Max, Kont, Berthe en Kosti. Sorry, maar onze nieuwe groepsnaam ga ik niet in deze tekst zetten.

Dank aan mama, voor de onvoorwaardelijke steun, en aan mijn broertjes Jeroen en Tim.

And above all, thanks to my wife, Tina. I could endlessly write here about how her support has helped me, but my emotions and prose do not mix well. Instead, I will apply a habit of mine that she has oft bemoaned: haphazardly make up a song about her set to an existing tune. It only seemed logical to finally put one in writing and — through this medium — publish it.

An Meine Freude

Tune: Chanson de Bicêtre, as sung in student folklore at the VUB and ULB.

Somewhere near Philly, I found my silly
Though now my wife, I can't tame her, no still she
Is in control of the things that we do
All for the best or boredom would ensue!
Yes I must say: what a life we are living
Crossing the ocean, it leads to thanksgiving

Oh Tina, what do you do?
Making me smile, when I don't want to
Oh Tina, what do you do?
No escape now, it's me and you

Hailing from China, parents left the nation
Brought to the US, much to my elation
對不起, 但我不会说中文[1]
Même en français ce serait comme une peine
Nee geef me dan toch maar Nederlands praten
Dat het nu Engels is, kan ik niet haten

Refrain

We have new adventure, right on the horizon
Your land of corn syrup, handegg, bison bison
Road trips abound, national parks, see you there
Roses in your hand and jasmine in your hair
Lacking all that we will still have each other
Living to the fullest with my sweetheart, my lover

Refrain

---

[1]Duìbùqǐ, dàn wǒ bù huì shuō zhōngwén.

Contents

Abstract iii

Samenvatting v

Acknowledgements vii

Contents xi

1 Introduction 1
1.1 Problem Statement 2
1.2 Overview of the Approach 3
1.3 Contributions 3
1.4 Supporting Publications 4
1.5 Dissertation Outline 6

2 Background 9
2.1 Why Version Control Your Software? 10
2.2 Version Control Software History 11
2.2.1 Ad Hoc Solutions 11
2.2.2 Local File-Based 11
2.2.3 Client-Server Repository-Based 13
2.2.4 Distributed Repository-Based 14
2.3 Git Model and Terminology 15
2.3.1 Objects 16
2.3.2 References 18
2.3.3 Forking and Merging 20
2.3.4 Future 24
2.4 Program Dependence Graph and Program Slicing 27
2.4.1 Program Dependence Graph 27
2.4.2 Program Slicing 29
2.5 Symbolic Execution 30
2.5.1 Overview 30
2.5.2 Dynamic Symbolic Execution 33
2.5.3 Limitations 34
2.6 Conclusion 35

3 Problematic Commits 37
3.1 Composite Commits . . . 38
3.1.1 Negative Effects . . . 38
3.1.2 Prevalence . . . 41
3.2 Merge Conflicts . . . 42
3.2.1 Textual Merge Conflicts . . . 43
3.2.2 Syntactic Merge Conflicts . . . 44
3.2.3 Semantic Merge Conflicts . . . 47
3.2.4 Build Merge Conflicts . . . 47
3.2.5 Test Merge Conflicts . . . 48
3.2.6 Summary . . . 49
3.3 Semantic Merge Conflicts . . . 50
3.3.1 Conflict Difficulty . . . 51
3.3.2 Conflict Resolution . . . 52
3.3.3 Solution Validation . . . 52
3.3.4 Behavioural Change . . . 53
3.3.5 Tool Support . . . 53
3.4 Conclusion . . . 54

4 Untangling Composite Commits Using Program Slicing 57
4.1 Proposed Solution . . . 58
4.2 Related Work . . . 59
4.2.1 Semantics-Based Commit Untangling . . . 59
4.2.2 Syntactic- or Textual-Based Commit Untangling . . . 61
4.3 Overview of the Approach . . . 62
4.3.1 Fine-Grained Change Distilling . . . 64
4.3.2 System Dependence Graph . . . 66
4.3.3 Program Slicing . . . 67
4.3.4 Change Grouping . . . 70
4.3.5 Prototype Output . . . 72
4.4 Dataset . . . 73
4.5 Evaluation . . . 76
4.5.1 Research Method . . . 77
4.5.2 Results . . . 79
4.5.3 Threats to Validity . . . 86
4.6 Conclusion . . . 87
4.7 Future Work . . . 87

5 Prevalence of Merge Conflicts 89
5.1 Research Questions . . . 90
5.2 Related Work . . . 90

5.3 Dataset 91
5.3.1 Origin of the Dataset 91
5.3.2 Refining the Dataset 94
5.3.3 Describing the Dataset 94
5.3.4 Terminology 96
5.4 Research Method 97
5.4.1 RQ1: How Often Do Code Integrations Lead To Syntactic And Semantic Conflicts? 97
5.4.2 RQ2: How Much Effort Is Needed to Fix Conflicts After Code Integration? 97
5.4.3 RQ3: What Type of Files Is the Effort to Fix Conflicts Concentrated In? 98
5.5 Results 99
5.5.1 RQ1: Frequency of Conflicts 99
5.5.2 RQ2: Effort to Fix Conflicts 101
5.5.3 RQ3: Source vs Test 103
5.6 Conclusion 104

6 Symbolic Execution to Detect Semantic Merge Conflicts 107
6.1 Proposed Solution 108
6.1.1 Context 108
6.1.2 Approach 109
6.1.3 Situating Our Approach 109
6.1.4 Evaluation 111
6.1.5 Contributions 111
6.2 Related Work 112
6.2.1 Program Analysis Approaches 112
6.2.2 Data-driven Approaches 114
6.3 Detecting Merge Conflicts by Symbolic Execution 115
6.3.1 Property P 115
6.3.2 Merge Conflict of Property P 115
6.3.3 Semantic Merge Conflict 117
6.3.4 Decompose the Conflict 118
6.4 Technical Details 121
6.4.1 Aligning Variables in Path Conditions 121
6.4.2 Symbolic Execution Engine 122
6.4.3 Constraint Solver 123
6.4.4 Prototype 124
6.4.5 Future Automation 127
6.5 Evaluation 129
6.5.1 Technical Validation 129
6.5.2 Empirical Validation 131

6.6 Conclusion 137

7 Conclusion 139

7.1 Revisiting the Contributions 140

7.2 Limitations and Future Work 141

7.3 Closing Remarks 142

Bibliography 145

List of Tables

2.1 Solutions by the Z3 constraint solver for the path conditions resulting from symbolic execution of Listing 2.2. 33

4.1 Descriptive statistics for the original dataset by Herzig and Zeller [66, 67], the dataset after automated filtering, and the dataset after our manual commit verification. In case of aggregated numbers (i.e., those per commit or per file), the median is given. 74
4.2 Identification of composite commits on a per project basis. The precision, recall, and F-score in the "Total" row are calculated using the totals of the columns to their left, not a combination of the per project precision, recall, and F-score. 80
4.3 The precision, recall, and F-score for the identification of single-task commits. These are calculated using the absolute numbers present in Table 4.2. The total row is calculated using the totals across the projects, not by a combination of the numbers in this table. 81
4.4 Identifying the number of single tasks within a composite commit. Only commits for which the number of partitions is present in the dataset are considered. 82
4.5 Time in seconds taken to analyse a commit with our untangling approach. A zero indicates analysis took under one second. The 25%, 50%, and 75% percentiles are listed. Mean and standard deviation are rounded to the nearest integer. 84
4.6 The twelve commits for which the analysis took over ten minutes. All times are listed in seconds. For every commit, the file that took the longest to analyse is listed as "Slowest File". The time taken for this file is listed in the second-to-last column. The percentage compared to the total time for the commit is listed in parentheses. The time taken to create the SDG and obtain a list of fine-grained changes for that file is listed in the last column. After that point, the analysis loops over the fine-grained changes to slice and link them together. In parentheses is the percentage of that time compared to the total time for the file. 85

5.1 A summary of the 348 projects with 50 or more builds of merge commits and parents as well as a sufficient number of successful builds across the project. 94
5.2 Definition of the *BREAK*% metrics for RQ1. 97
5.3 A summary of the *BREAK*% for all 348 projects. 100
5.4 An overview of the $\overline{TTF}$ metric. It shows 67% of projects usually fix a breaking build of a merge commit within a day. 102

6.1 Overview of the cases studied in RQ2. The first five cases are extracted from real-world projects by means of a manual examination following our retroactive semantic merge conflict identification. Case six is a synthetic adjustment of one such real-world case. Case seven is a synthetic case added to indicate a limitation in our approach. 133

List of Figures

2.1 Depiction of the chain of deltas used in SCCS [137]. A new delta for release 2, delta `2.3`, would be added at the very end of the chain. A new delta for release 1, delta `1.5`, would be added after delta `1.4`. Delta `1.5` would not be used when determining the source code for release 2. Reproduction of a figure in the original paper. 12

2.2 Overview of Git objects and how they relate to one another. Blobs represent file contents. Trees are lists of references to blobs and other trees (as visualised in the bottom two trees), akin to directories. A commit represents one version of the program under version control. A commit has a tree and refers to its parent commit(s). Every arrow indicates a reference from one object to another. . . . 16

2.3 Overview of references in Git and a line of commits (depicted as circles) they refer to. Main, cool-feat, and bugfix are branches (rectangles). HEAD is the special reference HEAD (cloud), here pointing at the main branch. `v0.4.1` is a tag (hexagon) pointing at a specific commit. Note that the direction of the arrows indicates the direction of the internal pointers. A commit thus happened after the commit it points to. 18

2.4 A commit graph with two merges. Merge commit 7 has commits 5 and 6 as parents. Their common ancestor is commit 2, from where the history forked. Merge commit 9 has commits 7 and 8 as parents. Their common ancestor is commit 6, another fork in the history. 21

2.5 Situation in which Git will default to a fast-forward when merging. The situation after fast-forwarding is depicted in Figure 2.6. If instead a merge commit is forced, the situation looks as depicted in Figure 2.7. 23

2.6 Situation after a merge that was instead fast-forwarded. The situation before fast-forwarding is depicted in Figure 2.5. 23

2.7 Situation when forcing Git to create a merge commit M instead of fast-forwarding. The situation before the merge is depicted in Figure 2.5. 24

2.8 The commit history before rebasing `bugfix` on `main`. Figure 2.9 shows the situation afterwards. 24

2.9 The rewritten commit history after the `bugfix` branch has been rebased on the `main` branch. Figure 2.8 shows the situation prior to rebasing. Note that commits 3, 6, and 7 are slightly modified as changing the parent changes the SHA-1 hash used to identify the commit. 25

2.10 A program dependence graph of Listing 2.1. Solid lines represent a control dependency. Consequent and alternative are marked by `true` and `false`, respectively. Dashed lines represent a data dependency. They are labelled with the used variable. This graph was generated by TinyPDG [68, 69, 70] and manually cleaned up for readability. The `int b` and `int c` declarations are hidden. Line number information was removed from the nodes. 28

2.11 A program dependence graph of Listing 2.1 after slicing forwards and backwards around node `c = 2 * b`. 30

2.12 The symbolic execution tree for an analysis of Listing 2.2. The diamonds depict a control flow decision. The equations outside the polygons are path conditions. 32

3.1 The four relevant parts when talking about merging and merge conflicts: the merge commit M, its direct parent commits A and B, and their common ancestor commit O. As in Chapter 2, the arrows indicate a child-parent relation. 43

3.2 Common categories when describing merge conflicts. Textual, syntactic, and semantic focus on the cause, while build and test focus on the symptom. As such, build and test can overlap depending on the context. 50

4.1 Overview of our approach. The input is a commit, the output the untangled single-task commits that make up the commit. The four main parts of our approach are each described in detail in Section 4.3. 63

4.2 Relevant parts of the system dependence graph for the running example of Section 4.3. Solid lines with the label `true` are control dependencies. Dashed lines are data dependencies. The label indicates the identifier set by the source node and used by the destination node. 68

4.3 Visual explanation of how slicing around a fine-grained change operation c works. The abstract syntax tree node affected by change operation c is marked as o_c. o_c corresponds to or is part of a node n_c in the system dependence graph. $S(c)$ is the resulting slice backwards from n_c. 69
4.4 Diff view produced by our prototype tool when applied to the running example from Listing 4.1. Key here is the leftmost column, which indicates the clusters present. 73
4.5 Results of a three-person survey. Each person analyses 31 commits and the result of our change clustering. They need to assess whether the change clustering makes sense. 83

5.1 Visualisation of the refinement of the TravisTorrent dataset. In the end we are left with 348 projects. 93
5.2 Number of merge commits plotted against, from top to bottom: number of commits, maximum team size, and number of branches. Each graph on the left hand side depicts all 348 selected projects. Each graph on the right hand side zooms in on a section closer to the origin. Every dot represents one project. 95
5.3 A comparison over all projects of the *BREAK*% metrics. (a) splits up breaking commits by regular commits and merge commits. (b) splits up the breaking merge commits by pull request. 100
5.4 $\overline{LINES}$ for every project. Despite the long tail, for 75% projects $\overline{LINES}$ is less than 36. The inset zooms in on the lower end of the graph. The inset still shows 88% of projects. 102
5.5 *SRC*, *TEST*, *BOTH*, and *NONE* metrics. Breaking merge commits in the majority of projects are usually repaired by changes to the source code. 104

6.1 The four relevant parts when talking about merging and merge conflicts: the merge commit M, its direct parent commits A and B, and their common ancestor commit O. The arrows indicate a child-parent relation. Reproduction of Figure 3.1. 108
6.2 The pipeline of steps to realise a merge commit that is free of conflicts. The process eventually leads to an accepted merge. 110

Listings

2.1 An example function for which a program dependence graph is shown in Figure 2.10. 27
2.2 A method where two guesses have to be provided. The first guess, `x`, needs to be larger than 50. The second guess, `y`, needs to be exactly 27182. On success, 0 is returned. On failure to guess correctly, numbers 2 and 1, respectively, are returned. 31
2.3 Block of code executing a random number generator ten times. For every positive random number, `sum` is incremented. Otherwise, `sum` is decremented. The number of paths in this program is $2^{10} = 1024$. 34

3.1 Version O of `welcome.txt` in a textual merge conflict. 44
3.2 `welcome.txt` in version A in a textual merge conflict. 44
3.3 `welcome.txt` in version B in a textual merge conflict. 44
3.4 When attempting to merge versions A and B of `welcome.txt` (see Listings 3.2 and 3.3), Git will not know which version to prioritise. Instead, this error message is shown and the user is expected to choose a resolution themselves. 44
3.5 Version O in a real-world syntactic merge conflict. Code extracted from the Spring Cloud Config project [148], merge 81585fe09e-5ffb70708e4c6b2767bb2af73ecc5c. 45
3.6 Version A in a syntactic merge conflict. A newline is removed and an `else` is added. 45
3.7 Version B in a syntactic merge conflict. The same `else` is added as in version A, but with a newline after an opening parenthesis. . . . 45
3.8 Version M in a syntactic merge conflict. Git does *not* consider this a textual merge conflict and completes the merge without problems. The resulting piece of code, however, has two `else` branches. . . . 45
3.9 Version O in a semantic merge conflict. `myAdd` sums two integers. . 48
3.10 Version A in a semantic merge conflict. `myAdd` sums two integers and adds one. 48
3.11 Version B in a semantic merge conflict. `myAdd` sums two integers and adds one. 48

3.12 Version M in a semantic merge conflict. `myAdd` sums two integers and adds one, then adds one again. 48
3.13 Version O in a build conflict. `inc` increases the argument by one. . 49
3.14 Version A in a build conflict. The parameter `x` is renamed to `n`. . . 49
3.15 Version B in a build conflict. The value of `x` is logged. 49
3.16 Version M in a build conflict. The parameter `x` is renamed to `n`. The old parameter `x` is still logged. 49

4.1 Difference view for commit 5e2cdc06 of ArgoUML. Used as running example for Section 4.3. The variables `_bigPort` and `_cover` are renamed to `bigPort` and `cover`. 65
4.2 The distilled changes for commit 5e2cdc06 of ArgoUML, the running example of Section 4.3. Each line is one fine-grained change. Note that the representation here is simplified for viewing. The actual distilled changes also indicate any new content as well as where in the abstract syntax tree they are supposed to be inserted, updated, moved, or deleted. 66
4.3 An example of a partial definition of an object. TinyPDG does not realise line three should have a data dependency on line two. . . . 67
4.4 Data produced by our prototype tool when applied to the running example from Listing 4.1. One line is produced per file analysed in a commit. 72

6.1 Version O in a semantic merge conflict. `myAdd` sums two integers. . 118
6.2 Version A in a semantic merge conflict. `myAdd` sums two integers and adds one. 118
6.3 Version B in a semantic merge conflict. `myAdd` sums two integers and adds one. 118
6.4 Version M in a semantic merge conflict. `myAdd` sums two integers and adds one, then adds one again. 118
6.5 The original version O has redundant safety checks, i.e., it checks for a division by zero twice: first on line 2 and then again on line 5. Branches A and B remove one check. However, they do not agree on which check is removed. In M, both checks are missing, so the safety check semantics are missing. 120
6.6 Branch A removes lines 2–4 from O. 120
6.7 Branch B removes lines 5, 7, and 8 from O. 120
6.8 In version M, both checks are missing, so the safety check semantics are missing. 120
6.9 Output of our prototype when run on the `myAdd` example in Listings 6.1 to 6.4 in Section 6.3.3. 124

6.10 Slightly modified output of our prototype when run on the `div` example in Listings 6.5 to 6.8 in Section 6.3.4. 128
6.11 Small simulation showing the probabilities of correctly labelling ten merges. 130

List of Acronyms

AST Abstract syntax tree

CF Conflict freedom

CVS Concurrent versions system

JDT Java development tools

PC Path condition

PDG Program dependence graph

RCS Revision control system

RQ Research question

SCCS Source code control system

SDG System dependence graph

SPF Symbolic PathFinder

SVN Subversion

VCS Version control system

Chapter 1

Introduction

When creating and maintaining programs, software developers need to store and manage many different versions of the source code. As one developer works on a new feature, another may be fixing a bug; each developer is working on their own version. Released versions of the program are also kept, in case a bug fix needs to be backported. A developer may also just want to keep the code around before making a bigger change, in case the change does not pan out.

To handle these different versions, version control software was first developed through ad hoc in-house solutions at least as far back as the 1960s, e.g., with IBM's CLEAR [22]. In the 1970s, Bell Labs developed the Source Code Control System (SCCS) [137]. The Revision Control System (RCS) was introduced in 1982, building on SCCS's ideas [156, 157, 158] and introducing the concepts of branching and merging. In 1986, the Concurrent Versions System (CVS) was built on top of RCS [62]. Contrary to SCCS and RCS, CVS manages the entire software project and its history as one entity: the repository. CVS works in a client-server setup, with one server managing the repository and users connecting to it to request a certain version. CVS dominated the version control software landscape throughout the 1990s and early 2000s. Subversion (SVN) was released in 2004 to solve many of the bugs and pitfalls CVS had to deal with [31]. SVN soon took over the top spot from CVS. As this was happening, a distributed approach to version control software was gaining prominence in the early 2000s, pioneered by BitKeeper [102, 103, 104] and Arch [97, 136]. In 2005, BitKeeper removed their community edition aimed at open source developers, which at the time was used by the Linux kernel. Linus Torvalds, author of the Linux kernel, decided to develop his own version control software: Git [159]. Git grew in popularity and is now by far the most popular version control software in use [79, 123].

Git uses the commit as a building block. A commit plays a dual role: it represents a version of the program at a certain point in time; it also represents the changes compared to the previous version of the program. In combination with its distributed nature, Git enables developers to work in parallel, to revert changes, to apply changes across different branches, and to do all of this in a structured manner.

1.1 Problem Statement

The commit is the building block in modern version control software. This in turn makes it a necessary focus of practitioners. We provide some examples. Developers share commits with one another to publish their changes, deal with commits when reverting a change, can pick certain commits out of the history to reapply them in other situations, and can perform various other commit-based version control software features. Reviewers evaluate changes per commit or set of commits. Researchers analyse the history of a project by looking at the timeline of commits [77], look for patterns in commits [150], or draw other conclusions from commits.

In that light, any perceived flaw or shorting of a commit has wide-reaching consequences. Consider, for example, the following two types of problematic commits. First, consider a *composite commit*, a commit that groups many unrelated changes. The changes target multiple tasks, such as fixing a bug, introducing a new feature, or performing a refactoring of the code. This makes a composite commit harder to understand than a commit performing just one task: a reviewer or other developer needs to distinguish which parts of the code belong together themselves [18, 61, 89, 154]. Composite commits also pose problems when wanting to revert or integrate the changes of just one of the tasks contained within. Composite commits are also harder for researchers to analyse [64, 66, 88, 108, 109, 119]. Second, consider *merge commits*. As developers work on different features or bug fixes, the different versions of the programs start to diverge. The different versions are recombined in a merge commit. The merge commit is thus the point where two different pieces of code first interact. This can lead to a textual overlap in the changes, a syntactic flaw in the combined code, or semantic conflicts as the behaviour creates unintended interactions [105].

Some of the issues related to problematic commits can be avoided by having developers adhere to proper practices. One can spend extra time working with their Git client to split up changes into different commits rather than committing everything at once. One can maintain an extensive test suite and use a continuous integration pipeline to detect issues with merge commits [114]. The upfront time and effort that need to be put into these proper practices are clear to the developer, but their long term gains may not be obvious. As such, developers do not necessarily follow proper practices, leading to problems down the line.

To mitigate these issues, we thus require automated techniques for problematic commits that enable us to reap the benefits of proper practices without requiring the same level of developer time and effort. Because code has semantic meaning, such techniques inherently require having a certain understanding or model of that semantic meaning in order to correctly create or validate commits.

The research in this dissertation is guided by the following statement:

We hypothesise that many of the issues caused by problematic commits can be avoided by means of tool-supported identification and resolution of these problematic commits. These techniques must be able to reduce some of the time and effort developers need to invest applying proper practices beforehand or resolving issues afterwards.

1.2 Overview of the Approach

We approach two types of problematic commits from two different angles. The types of problematic commits we consider are the two types discussed in the previous section: composite commits and merge commits. We investigate a data flow driven approach for one type and a control flow driven approach for the other.

For composite commits, we propose an untangling algorithm using a data flow driven approach. The algorithm uses fine-grained changes and slicing in program dependence graphs. Based on the program dependence graph slices they belong to, the changes are grouped together. We posit that these groups of changes can be used to identify the different tasks present in the composite commit, thus enabling the application of this technique prior to creating commits or post hoc to untangle existing commits.

In merge commits, we specifically target those where semantic merge conflicts occur: conflicts where the merge seems to have completed successfully, but an interaction in behaviour of the different branches leads to bugs that are not discovered until triggered by the test suite or, worse, a user. We propose a detection algorithm for semantic merge conflicts using a control flow driven approach. The algorithm uses symbolic execution to obtain path conditions from the different versions of the program. We define program semantics in terms of these path conditions. The algorithm checks the path conditions against rules that indicate whether a semantic merge conflict is present. The developer is warned when this is the case and provided the offending path conditions in order to locate the potential conflict.

1.3 Contributions

We make the following main contributions:

1. Design an automated approach to detect and untangle composite commits. The approach is based around the idea of creating a slice around the fine-grained changes made to an abstract syntax tree. The approach first considers which parts of the abstract syntax tree are affected by a change. The

approach then slices around the corresponding nodes in a program dependency graph. Overlap in the slices is used to decide whether the changes belong together.

2. The refinement of an existing dataset of composite commits from five Java projects. We perform an automated cleaning step and a manual verification. We use this dataset to evaluate a prototype implementation of our commit untangling technique. The dataset can be used by other researchers working with composite commits.

3. A large-scale study into the prevalence of syntactic and semantic merge conflicts. We combine data from Github and Travis CI to attain build status information of merge commits and their parents. We report on the prevalence of syntactic and semantic merge conflicts, the effort and time involved in fixing them, and the parts of a project the effort is situated in. The list of used projects and the derived list of fixing commits was made available to researchers desiring further study.

4. An automated approach to detecting semantic merge conflicts by means of symbolic execution. Symbolic execution is used to analyse every version in a merge. The path conditions produced by the symbolic execution are combined based on equivalency across versions. The combinations are validated against rules that recognise the presence of a semantic merge conflict.

5. An alternative approach to detecting semantic merge conflicts in the full revision history of a project. This approach centres around locating bug fixing commits that can be linked to the merge commit. The semantic merge conflicts found this way are used in the evaluation of the previous point.

6. Prototype implementations of contributions 1 and 4, available via [113]. Both prototypes are written in Java and target Java code. The prototypes were used to evaluate the approaches they implement.

1.4 Supporting Publications

- Ward Muylaert and Coen De Roover. 'Untangling Source Code Changes Using Program Slicing'. In: *BElgian-NEtherlands software eVOLution symposium (BENEVOL)*. ed. by Serge Demeyer, Ali Parsai, Gulsher Laghari and Brent van Bladel. Vol. 2047. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 36–38. URL: `https://ceur-ws.org/Vol-2047/BENEVOL_2017_paper_10.pdf`

In this publication, we describe a first tentative idea towards untangling composite commits. This idea was further worked out and presented in the publication that follows.

- Ward Muylaert and Coen De Roover. 'Untangling Composite Commits Using Program Slicing'. In: *18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, Sept. 2018, pp. 193–202. DOI: 10.1109/SCAM.2018.00030

In this publication, we analyse the problem posed by composite commits. The publication proposes an automated approach to untangling composite commits into clusters of related changes. The approach uses fine-grained changes between two versions of a program. These fine-grained changes are combined with information from a program dependence graph. The concept of slicing in a program dependence graph is adapted to slice around the fine-grained changes. The fine-grained changes are then grouped together based on the produced slices. This grouping finally determines how to untangle the composite commits. We will describe the problems caused by composite commits in Section 3.1. We will describe our untangling approach and its evaluation in Chapter 4.

- Ward Muylaert and Coen De Roover. 'Prevalence of Botched Code Integrations'. In: *14th International Conference on Mining Software Repositories (MSR)*. ed. by Jesús M. González-Barahona, Abram Hindle and Lin Tan. IEEE Computer Society, May 2017, pp. 503–506. DOI: 10.1109/MSR.2017.40

In this publication, we look into the prevalence of syntactic and semantic merge conflicts on a large scale. By combining data from Github, a software repository host, and Travis CI, a continuous integration service, we obtain information on the build status of merge commits and their parents across a wide range of projects. Based on this information, we analyse the frequency of syntactic and semantic merge conflicts, the effort required to fix such conflicts, and the parts of the code the effort focuses on. We will describe this analysis and its results in Chapter 5.

- Ward Muylaert, Johannes Härtel and Coen De Roover. 'Symbolic Execution to Detect Semantic Merge Conflicts'. In: *23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. ed. by Leon Moonen, Christian D. Newman and Alessandra Gorla. IEEE, Oct. 2023, pp. 186–197. DOI: 10.1109/SCAM59687.2023.00028

In this publication, we consider the problem of semantic merge conflicts. We first define merge conflicts in function of a property of each of the four parts of a merge. Using symbolic execution, we instantiate this property as a description of the behaviour of a program. With the merge conflict definition and this semantic description, we then propose an approach to detect semantic merge conflicts. We will describe the problems caused by semantic merge conflicts in Section 3.3. We will describe the approach and its evaluation in detail in Chapter 6.

1.5 Dissertation Outline

Chapter 2: Background

In this chapter, we start by motivating the use of version control software. We then provide a walk through the history of version control software and how it evolved from its beginnings to the current most-used one: Git. We follow this up by a deep dive into Git, its concepts, and terminology. We finish the chapter by describing three existing techniques that we will apply in later chapters: first, we look at program dependence graphs, which we use in Chapter 4; next, we look at slicing on those program dependence graphs, also used in Chapter 4; and finally, we describe symbolic execution, which we apply in Chapter 6.

Chapter 3: Problematic Commits

We describe two types of commits that have negative effects on developers, researchers, and other practitioners. First, we discuss *composite commits*, commits that combine many unrelated changes. We describe in what manner composite commits negatively affect practitioners and discuss research analysing their prevalence. This sets up Chapter 4, where we propose an approach to automatically untangle composite commits. Second, we discuss different types of merge conflict and then focus on *semantic merge conflicts* in particular. In a semantic merge conflict, merging code together from different branches may appear to have succeeded, but the interaction of behaviour leads to bugs that are not caught until the code is executed. We describe how these conflicts affect developers. We perform a study into the prevalence of merge conflicts in Chapter 5. We propose an automated approach to detecting semantic merge conflicts in Chapter 6.

Chapter 4: Untangling Composite Commits Using Program Slicing

In this chapter, we propose an algorithm to untangle composite commits using a data flow driven approach. Our approach combines two main concepts: (1) distilling fine-grained changes between the abstract syntax trees

of two versions of a program and (2) slicing in a program dependence graph. We define slicing around a fine-grained change. Our approach applies this definition to group the fine-grained changes together based on the produced slices. We develop a prototype implementing our approach. To evaluate our technique, we analyse and refine an established dataset of Java commits. We find that our algorithm can determine whether or not a commit is composite. When untangling the commit, we find that the groups of fine-grained changes tend to be smaller than the different tasks making up a composite commit. Groups do, however, stay within their respective tasks.

Chapter 5: Prevalence of Merge Commits
Following the discussion of different types of merge conflicts in Chapter 3, we look into the prevalence of syntactic and semantic merge conflicts on a large scale. By combining data from Github, a software repository host, and Travis CI, a continuous integration service, we obtain information on the build status of merge commits and their parents across a wide range of projects. We find merge commits lead to failure less often than regular commits. Repairing the code usually happens the same day, in fewer than ten lines, and primarily involves editing source code as opposed to test code. Applying proper practices, e.g., maintaining an extensive test suite, may take time and effort, but our results indicate they mitigate many issues associated with code integration.

Chapter 6: Symbolic Execution to Detect Semantic Merge Conflicts
In this chapter, we present an automated approach to detecting semantic merge conflicts using a control flow driven approach. We define program semantics in terms of the path conditions produced by a symbolic execution engine. Our approach checks whether the path conditions satisfy established rules that indicate a merge conflict. We develop a prototype that warns developers in the case of semantic merge conflicts. To evaluate our approach, we first perform a retroactive study to classify semantic merge conflicts using heuristics. The results of the retroactive study are used to evaluate our proactive detection of semantic merge conflicts. The evaluation shows our approach to be a promising complement to existing techniques for detecting semantic merge conflicts.

Chapter 2
Background

In this chapter, we provide some general background and context to the topics that are discussed in the rest of this dissertation: we detail version control software, program dependence graphs, and symbolic execution.

First, we briefly discuss a situation motivating why developers use version control software, such as Git. Second, we detail different approaches to version control software through its history. This is done to increase understanding of design decisions in Git. Third, we go deeper into how Git works, what terminology is common in its usage, as well as discussing some of the developments currently being explored to improve the common approach to version control software. Git is currently the most used version control software [79, 123] and the work in this dissertation is evaluated on projects managed in Git. Fourth, we detail program dependence graphs, which see heavy use in Chapter 4. Finally, we introduce symbolic execution, which is used in Chapter 6.

2.1 Why Version Control Your Software?

When developing and publishing software alone or as part of a larger team, it is easy to end up with many largely identical versions of the same application. This quickly leads to the lack of an overview and the inability to keep track of these versions.

Consider a situation where a developer has released version `1.0.0` of an application to the public. The developer continues working on the application to add a new feature for a future version `1.1.0` release. A user discovers a bug in the released version. The developer fixes the bug in the source code of the first released version so a new version, `1.0.1`, can be released. This version does not include the in-progress work on the feature as it is not yet ready for release. Instead, the developer also ports the bug fix to their in-progress `1.1.0` version. Some time later, the developer decides they need to refactor the work on the new feature. Afraid something might go wrong, the developer keeps a copy of the source code before starting the refactoring. After the refactoring and some further improvements, the feature is ready and the developer releases version `1.1.0`. The application is a success and the developer hires a second developer to speed up adding new features and fixing bugs. Both developers set to work on different new features for the application. As they do not want their work interfering with one another before it is finished, they keep their versions separate. When both features have been completed, the developers determine which parts of the source code need to be combined in order to release the next version which will contain both features.

As shown, even when working alone a developer may find themself needing to manage and support different stable released versions, an unreleased development version, a version currently being tested, or other versions of the software they are working on. When developers work together on the same software, the problem of managing different versions is exacerbated. Each developer may have their own local versions as they work on different features, fix bugs, or perform other software development related tasks.

To help developers with the management of different versions, version control software was created. Not only does version control software help developers store different versions, version control software also keeps track of connections between different versions, metadata describing different versions, moving code across different versions, and several other features.

2.2 Version Control Software History

2.2.1 Ad Hoc Solutions

The first documented references we could find concerning the creation of version control software describe ad hoc in-house solutions in the 1960s [22]. One such example is CLEAR and its extended version CLEAR-CASTER, developed at IBM [22]. CLEAR made use of the concept of "deltas", changes that were kept separate from the original source text. CLEAR was, however, never publicly released nor was it described in detail. Indeed, even further details of the exact representation of the deltas are unclear. The authors described CLEAR-CASTER as "requiring extra machine time, which has to be balanced against the benefits that might accrue in the long run to a complete development project. We haven't proved yet, one way or the other, whether it will significantly reduce the time taken to complete a project." [22].

2.2.2 Local File-Based

The first published research of an approach to version control software was in 1975 with the Source Code Control System (SCCS) [137]. SCCS first saw development and usage in 1972 at Bell Labs. SCCS, like CLEAR, uses the concept of deltas applied to a previous version of a file. Note that SCCS has no concept of an entire project, the version control is on a file basis. The authors argued using deltas, instead of the entire source code for every point in time, was necessary to manage disk space while attempting to keep a record of every version of every file. A delta in SCCS consists of the lines inserted and deleted by that change. A move of lines of code is described in terms of insertions and deletions as well. A textual reason for the delta also has to be added by the developer creating the delta.[1] The deltas are sorted through a version identifier of the form `x.y`. The `x` represents the release number, the `y` a level within the release. All the deltas form a list, sorted by their release number, then by their level, both from lowest to highest. This is depicted in Figure 2.1. Internally, the deltas are not stored separately, but instead more on a line-per-line basis for a certain file. Thus a file internally starts with just the exact source in version `1.1`. As new revisions are added, the file gradually grows with sections stating things such as "this line added in version `1.5`" or "this line deleted in version `2.3`". When a developer needs to get the source code for a certain release (or level within a release), SCCS builds the correct source code by going through the file that is requested and

[1] The authors note that "The quality of the reason (like the quality of the change itself) depends on the conscientiousness of the programmer. Reasons like 'Trouble Report 5576: change SUM header' are what one likes to see. Sometimes, unfortunately, one sees instead things like 'Another bug' or 'Tried again'.". This will still be familiar to anyone working with version control software.

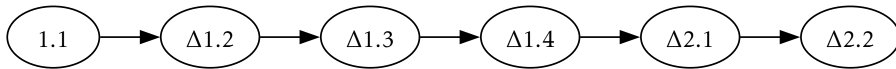

Figure 2.1: Depiction of the chain of deltas used in SCCS [137]. A new delta for release 2, delta `2.3`, would be added at the very end of the chain. A new delta for release 1, delta `1.5`, would be added after delta `1.4`. Delta `1.5` would not be used when determining the source code for release 2. Reproduction of a figure in the original paper.

apply the deltas as needed. This approach makes the complexity to get a file at a certain revision independent of when that revision was made. The approach does, however, make the complexity grow as the number of revisions increases. Conceptually, new deltas can only be added to the end of the release they belong to. In case this release is not the most recent one, they are entered within the chain of deltas, prior to the deltas of subsequent releases. SCCS then knows not to apply that particular delta in case the source for a subsequent release is asked for by a developer. After all, a freshly inserted delta may invalidate those that follow it. SCCS also has what it calls optional deltas. Optional deltas only get applied when explicitly asked for by the user. The idea behind this was to enable temporary fixes for a specific customer. Optional deltas and the ability to add a delta to the previous release thus enable a primitive form of explicitly splitting up the history of the source code.

The Revision Control System (RCS) was introduced in 1982 and builds on the approach employed in SCCS [156, 157, 158]. Just like in SCCS, editing a file in RCS requires checking the file out of the version control system, making your edits, and then checking it back into the system. To prevent issues due to two people editing the same file, checking out a file also means taking a lock on said file.[2] This in turn makes it clumsy for a team to work on a project. One might have to wait a while before being allowed to start editing a file. To get around this limitation, RCS introduced *branching* and *merging*. In the RCS model, a line of revisions is called a branch. There is one main line, referred to as the *trunk*. Different branches may sprout ("fork") from the trunk or from other branches. Thus a tree of revisions is created. Two branches A and B can be merged together again by the developer. Due to the tree structure, they have a common point from which both branches ultimately sprouted. To merge A and B together, RCS considers changes since that common point. The changes that happened since

[2]While not explicitly mentioned in [137], we believe SCCS worked in the same manner.

that common point to branch B are applied to branch A. If there is an overlap between the changes in both branches, RCS cannot decide what is desired. Instead, the developer needs to resolve the issue by manually editing the result.

Internally, RCS's approach to storing deltas is different from the one employed by SCCS. Tichy [156] refers to SCCS's approach as "merged deltas", whereas RCS uses "separate deltas". In the separate deltas approach, one initial revision is kept as-is in the version control software. Every other revision then has a delta describing the changes to the revision one step closer to the initial revision. Thus to get the file at a certain revision, the version control software needs to start from the initial revision and then apply every delta between the initial revision and the desired revision. There are two ways to implement the concept of separate deltas: (1) forward direction, where the oldest revision is kept as is and newer versions are made with deltas building on it and (2) backwards direction, where the reverse is true. Grune [62] use the terms positive and negative deltas, respectively, instead. RCS uses a backwards direction on the trunk, reasoning that the most recent version is the one that will be accessed most often. Getting the most recent revision does not require any processing since it is kept as-is. Getting older revisions however does increase in complexity the further back in the history the revision is. For branches other than the trunk, RCS uses forward direction from the point where the branch separates from the trunk. Thus to get the most recent revision on a branch, RCS needs to start at the most recent version from trunk, apply backward deltas to get back to the point of separation, then apply forward deltas to reach the correct revision. All this juggling with deltas is done to avoid having to store too many full versions of a file on disk, which was expensive at the time.

2.2.3 Client-Server Repository-Based

In 1986, Dick Grune built a first version of the Concurrent Versions System (CVS) on top of RCS [62]. Rather than working per file as RCS did, CVS considers the entire software project and its history as one entity. The entity is called a *repository*. The repository keeps track of what files are in it, though CVS does not track file metadata or renaming of files. A developer, interested in making changes, gets a revision from the repository, which results in them obtaining a local version of all the tracked files at that revision. Just like RCS, CVS supports branches separating from a main branch called trunk. As with RCS, the assumption is that most of the work happens on the latest revision of the trunk. As mentioned, CVS is built on top of RCS, using it to handle the versioning of individual files in the repository. Grune posits this is an implementation detail and not inherent to the CVS approach.

CVS changes the approach to locking files. In RCS, a developer obtained a lock on a file when getting a revision of the file. Thus, nobody else could attempt to

create a new revision for that file until the lock was released by that developer. In CVS, this is no longer the case. Two developers can both both get the same revision, version O, from the repository and make their local changes to it, creating a version A and a version B. When the first developer updates the repository with their version A, the update proceeds as usual. When the second developer attempts to update the repository with their changes, however, CVS does not allow it. Instead, the second developer needs to bring their local copy up to date with the new source of truth, i.e., version A. To do so, CVS employs the same strategy as was used by RCS to merge branches together. CVS considers the lines changed by versions A and B compared to the revision they both started from. If there is no overlap in the changed lines, the local version B is adjusted such that its changes are described in terms of version A. If there is an overlap, a concurrency conflict and its location are reported to the developer who is expected to resolve the conflict. Once any such conflict is resolved, the second developer can update the repository with their version B, which now follows version A.

At first just a collection of shell scripts, CVS was picked up and fleshed out by Brian Berliner. In 1990, Berliner published a version of CVS rewritten from scratch in C. The improved portability gained from this led to CVS quickly being picked up by users [14, 15]. Later improvements saw CVS take on a client-server model in 1993, which enabled the development of software by developers spread over the entire world. In this model, the server hosts the repository. Creating new revisions requires the developer to connect to the server. The introduction of this model was a boon to the burgeoning free software movement and CVS dominated the version control software landscape throughout the 1990s and early 2000s.

CVS was, however, not without its flaws and bugs. To that end, a company called CollabNet kickstarted work on a new version control software by hiring some people intimately familiar with CVS [31]. The new project was called Subversion (SVN), was also open source, and work on it was started in 2000. One of the goals was to keep the interface as similar as possible to the one used by CVS while avoiding any of the bugs and pitfalls CVS had encountered. The first `1.0` release was published in 2004 and SVN soon became the dominant version control software in use.

2.2.4 Distributed Repository-Based

Around the same time that development started on Subversion, a new kind of version control software was being developed. There are no clear research sources for this, but online sources indicate that Arch by Thomas Lord in 2001 was the first open source program of this new breed [97, 136]. This new kind of version control software was notably decentralised. With the client-server model of CVS and SVN abandoned, no special server software is needed. Instead communication happens over other protocols such as HTTP or SSH. This also means a revi-

sion can be finalised locally and simply shared with others afterwards, whereas CVS and SVN required the client to connect to the server in order to create a new revision. Arch shifted the focus to the difference between two revisions: *changesets*. A changeset consists of (1) a structural diff which describes the difference between the file trees, such as renames or permission changes, and of (2) a textual diff which describes the actual textual difference in the files themselves. Lord decided that getting changesets right was important, since these were what was shared with others through patch files sent around to others. They also showed more clearly the intention of the developer. This approach with a focus on changesets is still in use in some modern version control software, see for example Section 2.3.4. To further support a distributed approach in Arch, the computational complexity of branching and merging was made cheaper. This was a necessity when, contrary to CVS and SVN, every developer can create new revisions without first syncing up with others. Rather than having to merge locally before being able to *commit*, i.e., create a new revision, a developer could now commit locally before merging.

BitKeeper was a proprietary distributed version control program first proposed in 1998 and released in 2000 [102, 104]. BitKeeper pioneered at least some of the features later found in Arch [103]. BitKeeper provided a community edition for open source developers. Notably, and at the time controversially, the Linux kernel made use of BitKeeper from 2002 through 2005. The community edition was not available to all contributors of the Linux kernel however. To get around some of the proprietary limitations, some scripts were created by those contributors. In reaction to this, the community edition of BitKeeper was removed by its owners in 2005.

In the wake of this removal, Linus Torvalds, author of the Linux kernel, found the open source alternatives at the time lacking [159]. Instead Torvalds started from scratch and created Git. Git's popularity gradually grew and it is now the most popular version control software in use [79, 123]. We will go into more detail on Git in Section 2.3.

2.3 Git Model and Terminology

Git is currently the most used version control software. In JetBrains' "The State of Developer Ecosystem in 2023" survey, 87% of respondents state using Git regularly [79]. The next highest response for version control software scores but 6%. Of the nearly 1.5 million projects tracked by Open Hub[3] 74% use Git [123]. In this dissertation we analyse projects that make use of Git. That said, the problem we tackle, as well as our approach to do so, is not inherently linked to Git,

[3]Open Hub is a website with the goal of maintaining an index of open source software development.

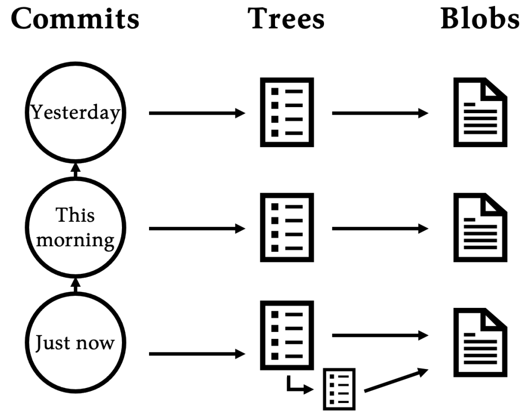

Figure 2.2: Overview of Git objects and how they relate to one another. Blobs represent file contents. Trees are lists of references to blobs and other trees (as visualised in the bottom two trees), akin to directories. A commit represents one version of the program under version control. A commit has a tree and refers to its parent commit(s). Every arrow indicates a reference from one object to another.

but is a more general problem when integrating software from different sources. This section gives a broad overview of Git and the terminology it uses. Sections 2.3.1 and 2.3.2 of this overview are based on the *Pro Git* book by Chacon and Straub [28].

2.3.1 Objects

Internally, Git is a collection of objects and the pointers between them.[4] The three types of objects are: (1) *blobs*, representing files, (2) *trees*, representing a collection of blobs and other trees akin to a directory, and (3) *commits*, representing a version and pointing to a tree and to zero or more parent commits. Figure 2.2 depicts how the different objects relate to one another.

On disk, the objects are stored in compressed files which are named after the SHA-1 hash[5] of a combination of a Git header to identify the type of object and the content of the object itself. By default, objects are stored in full, meaning a new object is saved in its entirety in a different file. This in stark contrast to the delta approach taken by the version control software described in Section 2.2. Git does optimise the storage periodically by returning to that delta approach. Once some configured threshold is hit, Git's garbage collection will pack certain objects together in what they call "packfiles". Within a packfile, the different versions of

[4] *Pro Git* describes Git's internal database as a "content-addressable filesystem", a key-value store.
[5] At the time of writing this is SHA-1. There are plans to move to SHA-256 [120].

an object may be saved as deltas compared to their most recent version present in the packfile.

Blobs

At the basis of Git's internal system lies the *blob* object. The blob object represents a file's content. As noted in the previous paragraph, a new object means a new file on disk, storing the content in full. This is also true for blob objects, a far cry from older version control software which was forced to minimise disk usage from the start.

Trees

The second object is a *tree*. A tree is a list of references to blobs and to other trees, identified by their SHA-1 hashes. Conceptually this can be seen as a directory. Consider as an example a file at location dir1/dir2/file.txt. A tree representing `dir1` would have a reference to a tree representing `dir2`, which in turn has a reference to a blob representing the contents of `file.txt`. A tree also stores metadata for the objects it references: their file or directory name, their object type, and the permissions. A new tree object is created if its list of objects changes, i.e., when a blob or tree is added, removed, or renamed. Such a new tree object would still point to the same blobs and trees if their contents were unchanged.

Commits

The third object is a *commit*. Conceptually this is a revision, a version, of the source code. Most commits have two pointers. First to a certain tree object, i.e., the root folder of the files at that point. Second to its parent commit, i.e., the previous version of the source code. Not all commits have exactly one parent commit. A root commit will have no parent commits. A merge commit will usually have two parent commits, though Git can be made to use more parent commits. We will provide more detail about merges later on in Section 3.2. Note that one commit can be the parent of many other commits, this is how the history splits up ("forks") as different people work on the code. Besides the pointers to other objects, commits can also carry some metadata such as a commit message describing the changes, a date, the author, or the committer. Since commits are also objects, they are also identified by means of their SHA-1 hash. These are the hashes a Git user will be most familiar with, every `git commit` issued creates a new one in a project's history.

It is through these commits that a project gets its history. Starting from a commit, one can walk back following the commit's parents to find previous versions of the project. Creating a new commit through `git commit` is akin to saving a snapshot of the project at that point. Doing so has the new commit pointing to

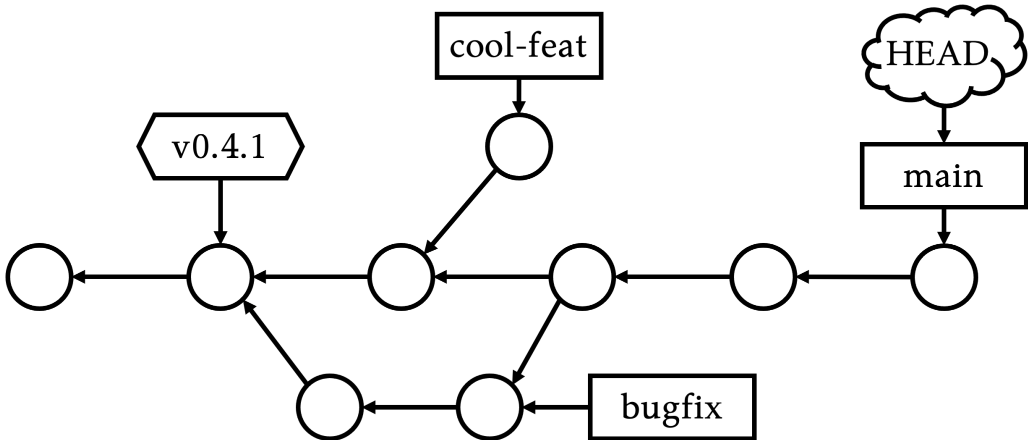

Figure 2.3: Overview of references in Git and a line of commits (depicted as circles) they refer to. Main, cool-feat, and bugfix are branches (rectangles). HEAD is the special reference HEAD (cloud), here pointing at the main branch. `v0.4.1` is a tag (hexagon) pointing at a specific commit. Note that the direction of the arrows indicates the direction of the internal pointers. A commit thus happened after the commit it points to.

the currently active commit as its parent commit. The new commit then becomes the currently active commit.

Commits are an overloaded term; in common usage they carry a dual meaning. On the one hand, commits are the version of the source code at that recorded point. On the other hand, commits are the difference between that version and the version prior to it. Conceptually, one can see the commit graph as consisting of either of the two meanings [96]. Indeed, switching between the two representations is possible as long as two tools exist: one to create a patch given two versions (such as `diff`) and one to apply a patch to one version (such as `patch`).

2.3.2 References

Working with just commit SHA-1 hashes quickly becomes confusing. To remedy this, Git uses four different types of "references". One can think of these as human-readable names for the commit hashes. An overview depicting three of the different reference types and a commit graph is shown in Figure 2.3.

Branches

The first type of reference is a *branch*. In Figure 2.3 these are visualised by rectangles. A branch is a named pointer to one certain commit, internally it is just a

file with the given name and as content the SHA-1 hash of the commit in question. Multiple branches can point to the same commit. Every project will start out with one branch, commonly called ‘master’ or ‘main’. A new branch can be created with `git branch`. Branches are the main way that a Git user will use to reach (collections of) commits. Since Git’s commits only keep track of their direct parents, branches serve as the starting points to find such a chain of commits. When referring to a branch, one may mean either the exact commit being pointed at or that commit and its entire history.

HEAD

The second type of reference is unique. It is called *HEAD*. In Figure 2.3 HEAD is visualised with a cloud. Git uses the HEAD reference to decide what the currently active commit is. To some extent, HEAD can be thought of as corresponding to the current local version of the code that a developer sees on their file system.

Generally, HEAD points to a branch, with the branch then pointing to a certain commit. Switching from one branch to another (also called “checking out” a branch, achievable through `git checkout`) changes the HEAD pointer to point to the other branch. Switching what HEAD points to also directly affects the code the developer sees in their project directory. The content changes to match the content of the tree object belonging to the commit object pointed to by the branch reference that the HEAD reference now points to. When creating a new commit, Git will also check what branch HEAD currently points to. Git will then modify the branch reference, making the branch point to the newly created commit. In this scenario, the HEAD itself is not changed directly, though following the references does make it point to the newly created commit.

When HEAD does not point at a branch, however, it is referred to as being “detached”. This can be achieved by checking out a commit directly, a tag (see below), or a remote branch (see below). Git warns the user when this situation occurs. If one were to create a new commit while the HEAD is detached, then there is no branch pointer to move to the newly created commit. Nothing references the new commit, meaning it will not be listed in any of the default ways one would interact with Git. Those default ways rely on references to find starting points in the chain of commits. While the user can refer to the new commit by its SHA-1 hash, the lack of references puts the commit at risk of being garbage collected by Git at a later point in time.

Tags

A third type of reference is a *tag*. In Figure 2.3 tags are visualised with a hexagon. A tag can point to any type of object, but is most commonly used to reference a commit. Specifically, a common approach is to tag a specific release of the soft-

ware under version control, e.g., `v0.4.1`. A tag can be "lightweight" or "annotated". A lightweight tag is nothing more than the name of the tag and a reference to an object by means of the object's SHA-1 hash. An annotated tag holds extra metadata such as the tag author, date, and a description. Due to the extra information that needs to be stored, an annotated tag is actually stored as an object internally. This is, however, an implementation detail.

Remote Branches

A final reference type are *remote branches*. These are read-only branches showing the last known state of a remote version of the Git repository. That last known state may or may not be integrated in the local versions of the Git repository. Often when checking out a remote branch locally, Git will automatically create a local branch of the same name, pointing to the same commit. If this did not happen, then HEAD points to a commit directly instead of to a branch. As mentioned before, this creates a detached HEAD state and puts the user at risk of losing commits that are made from that point onwards.

2.3.3 Forking and Merging

A local repository communicates with the world by fetching changes from or pushing changes to a remote repository.[6] Git is distributed version control software, so it imposes no hierarchy onto the different repositories. That said, designating one repository as the one true repository is a common approach taken by developer teams. Doing so enables every authorised developer to fetch from and push to one specific repository rather than keeping track of which repositories need to contacted. Git also offers a `pull` command. This is the same as fetching a remote branch reference and then merging (see below) it into the local branch.

When fetching or pushing, Git uses the references in the repository to decide which information gets shared. Pushing by default only considers the currently active branch, fetching defaults to all remote branches and tags. When the references get shared, Git recursively decides which commits and other objects need to be shared between the two repositories such that the references make sense. After all, if a reference points to a commit, then one would expect not just that commit to be present, but also the commit's tree, the commit's parent commit(s), and any other objects referenced by those two. Fetching does not change the local branches, only the remote branch references are updated.

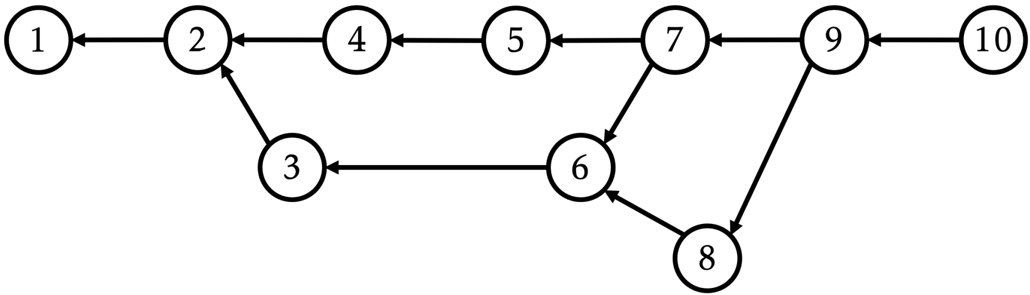

Figure 2.4: A commit graph with two merges. Merge commit 7 has commits 5 and 6 as parents. Their common ancestor is commit 2, from where the history forked. Merge commit 9 has commits 7 and 8 as parents. Their common ancestor is commit 6, another fork in the history.

Fork

As explained in Section 2.3.2, Git keeps track of the currently active commit. Creating a new commit means making it the currently active commit. The previously active commit is used as the new commit's parent.

A commit c_2 that already serves as the parent of a commit c_4, can be checked out again, either through a branch, directly, or through another reference. When the new commit c_3 is then created, the existing child c_4 of the current commit c_2 is not removed. Instead, the history *forks* so that the current commit c_2 is also the parent of the new commit c_3. The result of these actions is visualised in Figure 2.4 in the commits marked 2, 3, and 4. Since a Git user mainly works through branch references in Git, this usually means there is a branch reference pointing at c_4 (or one of its descendants) and another branch reference at c_3. Thus the terms often intertwine: one forks off a new branch, one branches off, and other combinations.

Due to Git's distributed nature, every repository is standalone. Git nor the developer are aware of what happens in remote repositories unless the information is explicitly fetched. Thus, developers on different machines can and do create commits using the same parent commit without even realising so. In other words, the Git commit history is constantly forking into different branches even when a developer is not consciously trying to do so. Note that this does not cause any issues in the Git history. After all, fetching merely updates the remote branch references.

[6]A remote repository can also reside on the same machine, but this does not change the way repositories interact.

Merge

Forking is an essential part of working with Git. However, in many situations one may want to do the opposite and recombine forked off versions. Consider for example a branch in which a bug was fixed, while regular development continued in another branch. Ideally, the other branch would also incorporate the bug fix. While one can manually reapply the bug fix to the other branch, the more common way in Git is to perform a *merge*.

When merging, two commits serve as the parent commit for one commit referred to as a *merge commit*. In this dissertation, we will usually refer to the two parent commits as A and B. We will use M to refer to the merge commit. While technically possible to merge more than two commits at a time, this is rarely done. Figure 2.4 depicts a commit graph with two merge commits. Git uses a three-way merge algorithm to perform this merge. These kind of algorithms were introduced with `diff3` in the late 1970s [43, 84]. Git's three-way merge algorithm uses not only the two commits being merged, but also goes through the project's history to find their common ancestor.[7] We will refer to the common ancestor as O. The extra information from the common ancestor enables checking what has changed on either side compared to that common ancestor. This in turn is used to decide what to keep in the final merged version. If one of the two parent commits did not change a line of a file compared to the common ancestor while the other parent commit did, then Git will use the changed line for the merged version. If both parent commits changed the same line in different ways, then Git does not know which version is preferred. Instead, a *merge conflict* is raised to the developer. The developer is then expected to choose which side, if any, they prefer to keep in the merged version.[8] We go into greater detail in merge conflicts in Section 3.2.

Performing a `git merge` does not always result in a new (merge) commit. Consider a situation such as in Figure 2.5 where the `bugfix` branch will be merged into the `main` branch. In this case, no commits were made on the `main` branch after the `bugfix` branch was created. The commit history did not fork; `main` is

[7]In normal Git usage, there will always be a common ancestor. It is, however, possible to create two unrelated trees of commits. A developer can do so manually with `git checkout --orphan new_branch_name` which will create an entirely new branch. A user of Github Pages may also have noticed such an unrelated tree in their project repository under the `gh-pages` branch. Trying to merge two such unrelated trees with `git merge` will result in an error: `fatal: refusing to merge unrelated histories`. The developer can still force Git to merge the two unrelated trees by passing the `--allow-unrelated-histories` flag. In case of any overlap, Git will, due to the lack of any common ancestor, more quickly raise a merge conflict for the developer to solve, requiring them to decide which lines to keep in the final version.

[8]Perhaps both changes need to be in the merged version. Perhaps neither needs to remain and the developer instead chooses a third ad hoc solution. Indeed, the developer may choose to add or remove completely unrelated lines to add to the version of the file that will be part of the merge commit.

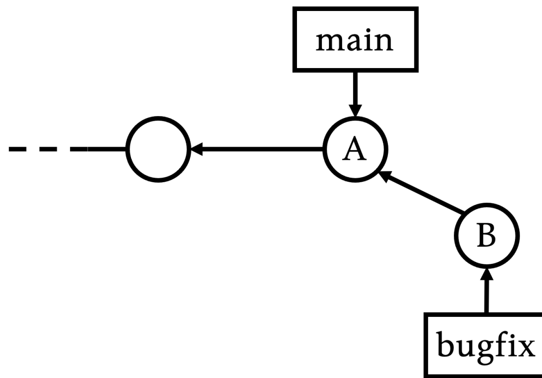

Figure 2.5: Situation in which Git will default to a fast-forward when merging. The situation after fast-forwarding is depicted in Figure 2.6. If instead a merge commit is forced, the situation looks as depicted in Figure 2.7.

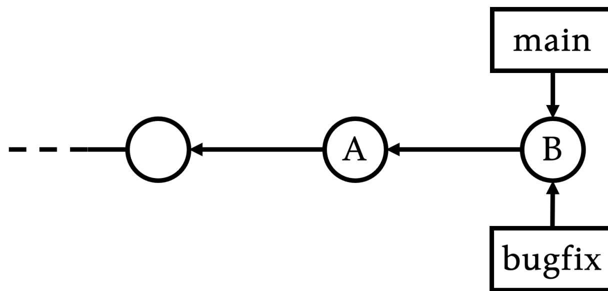

Figure 2.6: Situation after a merge that was instead fast-forwarded. The situation before fast-forwarding is depicted in Figure 2.5.

a part of the history of `bugfix`. Rather than create a merge commit, Git will instead *fast-forward* `main` to catch up with `bugfix`. Git does this by moving the `main` pointer to point to the same commit as the one pointed at by `bugfix`. This is depicted in Figure 2.6. The fast-forwarding behaviour can be overridden by using `--no-ff` when merging, in which case a "trivial" merge commit is created instead. This is depicted in Figure 2.7.

Rebase

Besides merging, Git offers "rebasing" as another way of reconciling histories. One can think of rebasing as performing a merge, but rewriting history so that it appears as if the history never forked to begin with. Figures 2.8 and 2.9 depict a rebase. When rebasing the `bugfix` branch on the `main` branch, Git still considers the common ancestor of both branches. Git takes every commit in `bugfix` since

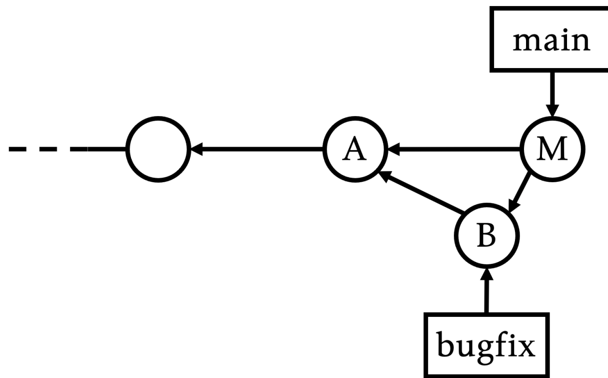

Figure 2.7: Situation when forcing Git to create a merge commit M instead of fast-forwarding. The situation before the merge is depicted in Figure 2.5.

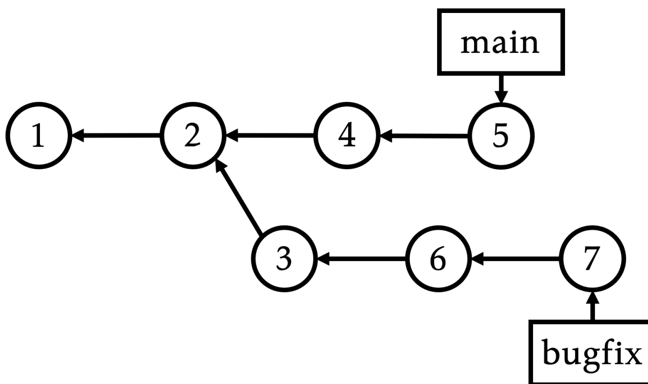

Figure 2.8: The commit history before rebasing `bugfix` on `main`. Figure 2.9 shows the situation afterwards.

that common ancestor and "replays" the changes of each commit in turn on `main`. In this situation it replays commits 3, 6, and 7. New commits (3′, 6′, and 7′) are created when rebasing. "Merge" conflicts can still occur. If a replayed commit conflicts with changes in the `main` branch, Git warns the developer of a merge conflict. As before, the developer is then expected to decide what to do before Git continues replaying commits. A rebase effectively rewrites the Git history. Afterwards, one cannot tell that the history had forked. Due to this limitation, we do not consider merge conflicts created while rebasing in this dissertation.

2.3.4 Future

Git is ubiquitous, but by no means perfect. Alternative approaches have seen research and development. Whether any programs will replace Git or whether

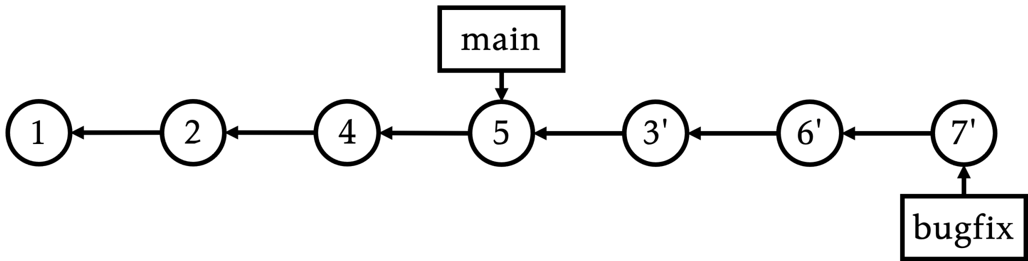

Figure 2.9: The rewritten commit history after the `bugfix` branch has been rebased on the `main` branch. Figure 2.8 shows the situation prior to rebasing. Note that commits 3, 6, and 7 are slightly modified as changing the parent changes the SHA-1 hash used to identify the commit.

any of the major ideas will be included in Git, only time will tell. We detail some different approaches taken that try to improve the user experience of version control software. These approaches range from improving Git with plug-ins to new tools with radically different approaches to version control software.

Improving Line-Based Differencing

Git stores the different versions of a file, not the changes between those versions (see Section 2.3). Only when asked to show the difference between two versions, does the differencing tool create the patch format to be shown to the user.

Rather than relying on the built-in differencing tool, Git users can choose any other differencer. At its most basic level this can be done through other line-based differencers, which visualise the difference in another way, such as delta [38] or vimdiff [166].

Entirely different tools can be used as well. Asenov et al. [8] build on top of Git by parsing the file into its abstract syntax trees (AST) and showing differences between those. One could take a similar approach using the more popular ChangeDistiller [29, 48] which describes changes to an AST in terms of inserting, updating, moving, and deleting nodes. Taking this idea a step further, one may be able to introduce semantic differencing [19, 73, 78, 92, 125, 128, 129] straight into one's Git setup. In semantic differencing, the behaviour of two versions is compared and changes in said behaviour are reported to the user.

Integrated Services

Fossil [72] is distributed version control software that works in a similar way to Git, but uses some different design choices [71]. Fossil was first released in 2006 and saw its first serious adoption by a project in 2009 when SQLite, database software by the same author as Fossil, started using it. The major selling point of

Fossil is that features like a web interface, an issue tracker and a wiki are built-in and version controlled. Thus, there is no need for extra software, such as GitHub, offering those features.

While Fossil is distributed, using it in the distributed manner that one might be used to from Git seems to be discouraged. By design, users are not supposed to work away in private. Instead all branches are synced continuously. Fossil does not allow much in terms of history rewriting either. This leads to failed or abandoned attempts in different branches remaining visible in the history forever. This is in contrast to Git where a branch may remain private indefinitely and its history rewritten entirely before it is shared with the world. Furthermore, Fossil wants users to see the entire graph of changesets at once by default, showing all parallel branches rather than just the changesets in the branch one is working on.

When merging or performing similar actions in Git, developers are immediately expected to create a changeset of the new situation. In Fossil, this changeset creation is postponed. The developer is expected to first test that the merge went as expected, before making a concious decision to create a changeset. The Fossil authors state that this stimulates clearer descriptions by the developer performing the merge, compared to the often used default message when using Git.

Sound Distributed Version Control Software

Pijul advertises itself as a "sound distributed version control system" [106], stating it is inspired by Darcs [37], a version control system with a focus on changesets akin to Arch's approach discussed in Section 2.2.4. Pijul claims it fixes soundness and performance problems found in Darcs.

Pijul describes files by means of a graph: a line of code is a vertex, an edge indicates the order from one line to the next. An edge is associated with a change that introduces or removes it. Updates to the file mean the addition of edges in the case of added lines, or the indication that an edge is now 'dead' in the case of deleted lines. The change to an edge is also linked to the changes it depends on. The Pijul authors say this creates a conflict-free replicated datatype (CRDT).

We will not go into further detail here, but the authors claim the graph approach precludes merge conflicts (see Section 3.2) as they are known in Git. Specifically, the approach precludes *textual* merge conflicts (see Section 3.2.1). As the authors state: "this does not mean that the merge will have the intended semantics" [107].

Version Controlling Conflicts

Jujutsu [174] is version control software that stores extra information not considered by most version control software. For example, Jujutsu constantly stores the current state of the directory, amending an existing commit of the code. In

```
1  int foo(int x, int y, int z) {
2      int b;
3      int c;
4
5      if (x > 10) {
6          b = x + y;
7          c = b * z;
8      } else {
9          b = 5;
10         c = 2 * b;
11     }
12
13     return c;
14 }
```

Listing 2.1: An example function for which a program dependence graph is shown in Figure 2.10.

Git, one can see the changes made since the last commit by using `git diff`, but these changes are not tracked until explicitly added with `git add`. Jujutsu also keeps track of all operations that are executed, such as syncing with remote repositories. This enables undoing any of the operations. Perhaps most radically, Jujutsu commits (textual) merge conflicts (see Section 3.2 and specifically Section 3.2.1), simply indicating in the revision log that there is an unresolved conflict. The user may still continue working as they please, committing more code and only bothering to fix the conflict when they are ready to do so. When the conflict does get resolved, the history that follows the conflict is automatically rebased (see Section 2.3.3).

2.4 Program Dependence Graph and Program Slicing

We provide a primer on program dependence graphs and program slicing, which we use in Chapter 4.

2.4.1 Program Dependence Graph

Ferrante et al. [47] introduced the program dependence graph (PDG) in 1987. A PDG represents both data flow dependencies and control flow dependencies of a program in one graph. The nodes in a PDG are statements of the program. The edges are either data flow edges or control flow edges. A data flow edge connects two nodes when the one node uses data, e.g., a variable, set by the other node. A control flow edge connects a node to its preceding control statement, e.g., to a node representing an `if` statement.

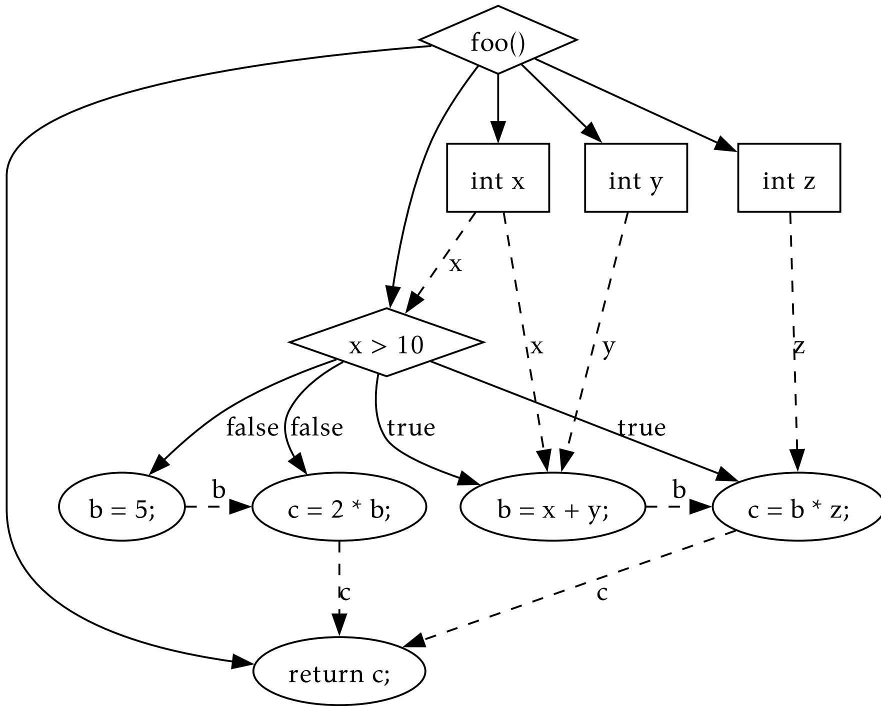

Figure 2.10: A program dependence graph of Listing 2.1. Solid lines represent a control dependency. Consequent and alternative are marked by `true` and `false`, respectively. Dashed lines represent a data dependency. They are labelled with the used variable. This graph was generated by TinyPDG [68, 69, 70] and manually cleaned up for readability. The `int b` and `int c` declarations are hidden. Line number information was removed from the nodes.

An example program is shown in Listing 2.1. The `foo` method has three parameters: `x`, `y`, and `z`. If the value of `x` is larger than 10, then the three parameters are combined to one value that is returned. If `x` is not larger than 10, then a value unrelated to the three parameters is returned.

We use TinyPDG [68, 69, 70] to generate a PDG and clean it up for readability.[9] This PDG is shown in Figure 2.10. Control dependencies are depicted by solid lines. Statements that get executed regardless of the `if` get a control dependency

[9] TinyPDG creates intra-procedural PDGs. In Chapter 4, we extend this functionality to create inter-procedural PDGs.

from the start of the function. Those in the consequent or the alternative of the `if` get marked as `true` or `false` control dependencies, respectively. Data dependencies are depicted by dashed lines. The label next to the dashed line indicates which variable caused the data dependency.

2.4.2 Program Slicing

Program slicing [169, 170] is the process of creating a smaller version of a program such that, for a certain program property or behaviour chosen by the user, the original and the smaller program obtain the same value or behave the same way. By enabling a developer to focus on a smaller section of the code, program slicing helps, for example, when debugging or understanding the code. One way to perform program slicing is by using the program dependence graph described in Section 2.4.1.

When working with a program dependence graph, this slicing is performed by transitively following the edges starting from a certain node of interest. There are some different approaches here depending on the user's preferences. The three main slicing strategies in program dependence graphs are backwards, forwards, and both. Backwards means starting at the node of interest and following the edges backwards till the start of the program. This thus finds nodes that affect the node of interest. Forwards slicing follows the edges, finding nodes that are affected by the node of interest. On its own this forward slicing may not produce any code that can be executed, but knowing which parts of the code are affected by the node of interest can, for example, help when planning a modification to the program. When applying the both strategy, the edges are followed in both directions.

Slicing can also be performed statically or dynamically [143, 151]. Static slicing [170] is the original approach and aims to preserve the behaviour for all the program input. Dynamic slicing [90] instead aims to preserve behaviour for only a subset of the program input. In Chapter 4, we will use static slicing.

We look again at the example function shown in Listing 2.1 and its program dependence graph in Figure 2.10. Say we are interested in the behaviour on line 10 and want to slice the program to preserve its behaviour. In the program dependence graph, this line has its own node. Slicing backwards means following the transitive control and data dependencies backwards, resulting in the nodes that ensure `c` gets the value it ends up with (here 10). Slicing forwards means following the data dependency to the `return` statement's node. The sliced program dependency graph is shown in Figure 2.11. From this program dependence graph, a function can then be reconstructed.

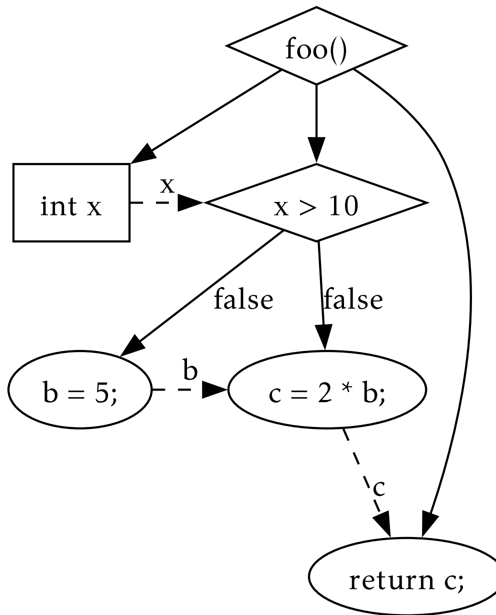

Figure 2.11: A program dependence graph of Listing 2.1 after slicing forwards and backwards around node `c = 2 * b`.

2.5 Symbolic Execution

Symbolic execution is a program analysis technique first described in 1976 by King [86]. The main use case for symbolic execution is the automated exploration of a program and the generation of inputs for test cases to exercise the program [26, 57, 58, 125]. We make use of symbolic execution in Chapter 6.

2.5.1 Overview

When analysing a program, symbolic execution assigns symbolic variables to the program's input. As the code is symbolically executed, constraints are placed on these symbolic variables. When a control structure, such as an `if`, is encountered, the analysis splits up. The analysis thus considers the different execution paths of the program. The repeated splitting up can conceptually be thought of as creating a "symbolic execution tree" where each leaf node represents the end of the execution of that path. Every path thus analysed will have a path condition

```
1  int guess_the_codes(int x, int y) {
2      if (x > 50) {
3          if (y == 27182) {
4              return 0; // Success!
5          } else {
6              return 1; // Close
7          }
8      } else {
9          return 2; // Not even close
10     }
11 }
```

Listing 2.2: A method where two guesses have to be provided. The first guess, x, needs to be larger than 50. The second guess, y, needs to be exactly 27182. On success, 0 is returned. On failure to guess correctly, numbers 2 and 1, respectively, are returned.

associated with it: a collection of the constraints on the symbolic variables encountered along that path.

Consider the code in Listing 2.2. The symbolic execution tree for this program is shown in Figure 2.12. The input for this program are the parameters x and y. Symbolic execution creates two symbolic variables: x and y, respectively. Note that what can be considered input is not necessarily only the parameters of a method. Retrieving information from an external source, a call of `Math.random()`, a prompt for user input, and, depending on the type of symbolic execution, a call to a method that is not analysed by symbolic execution are other examples of potential input.

As symbolic execution starts up there is one path to follow with an empty path condition. There are no constraints on the symbolic variables.

The first `if`, on line 2, is encountered and the analysis forks in two. A constraint is added to the path condition on each side of the fork. On the left side, in Figure 2.12, this makes for $x > 50$. On the right side, this is $x \leq 50$.

Continuing symbolic execution on the left side of the tree leads to the `if` on line 3. Once again, analysis forks in two and a constraint is added to each side. The path conditions are now $(x > 50) \land (y = 27182)$ on the left side and $(x > 50) \land (y \neq 27182)$ on the right.

Once more continuing on the left, symbolic execution encounters a `return` on line 4. In the most basic case, symbolic execution now terminates for this execution path and its path condition is logged:

$$pc_1 = (x > 50) \land (y = 27182).$$

In some symbolic execution engines, the value returned by a program or method under analysis also becomes a part of the final path constraint. Suppose this

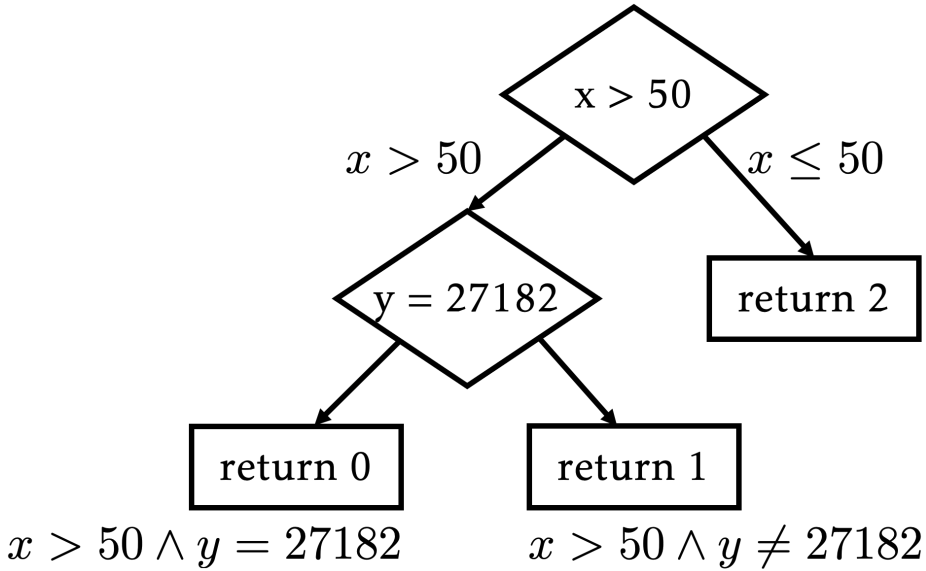

Figure 2.12: The symbolic execution tree for an analysis of Listing 2.2. The diamonds depict a control flow decision. The equations outside the polygons are path conditions.

variable is called *return*. The extended path condition for this first path of the execution then becomes

$$pc_1' = (x > 50) \wedge (y = 27182) \wedge (return = 0).$$

Going back to the previous fork, symbolic execution handles the other side and encounters the `return` on line 6. Once again the path terminates and the path condition is

$$pc_2 = (x > 50) \wedge (y \neq 27182).$$

Finally, handling the right side of the first forking, symbolic execution encounters the `return` on line 9. This path terminates as well and the path condition is

$$pc_3 = (x \leq 50).$$

As mentioned, a common usage of symbolic execution is to obtain inputs for test cases. Going from the generated path conditions, with their constraints on symbolic variables, to the concrete values that lead to the execution of different parts of the program requires solving the path conditions. Constraint solvers, such as Z3 [39], are used to this end. Solving the path conditions pc_1, pc_2, and pc_3 with Z3, for example, results in the values shown in Table 2.1. Note that many different solutions may be possible for one path condition. Calling the program

Path Condition	x	y
pc_1	51	27182
pc_2	51	27183
pc_3	50	0

Table 2.1: Solutions by the Z3 constraint solver for the path conditions resulting from symbolic execution of Listing 2.2.

in Listing 2.2 with the values found for x and y will cause the path corresponding to that path condition to be executed. Thus a test can be added that executes the program using those inputs.

2.5.2 Dynamic Symbolic Execution

Dynamic symbolic execution is a variation on symbolic execution first introduced in 2005 by Godefroid et al. [58]. In dynamic symbolic execution, the program is simultaneously executed concretely and symbolically. While executing it concretely, symbolic variables are still created for input and constraints placed upon them are still collected into a path condition. By having concrete values at its disposal, dynamic symbolic execution can continue analysis in some cases where regular symbolic execution falls short. Another term in the literature is concolic testing [140], a portmanteau of *conc*rete and symb*olic* execution. Concolic testing is an approach to dynamic symbolic execution in which each path is explored in turn from the start of the program under analysis,[10] as opposed to, for example, the forking strategy described in Section 2.5.1. As such, concolic testing is a subcategory of dynamic symbolic execution. We describe this dynamic symbolic execution approach in the following paragraphs.

Consider again Listing 2.2. On a first run, dynamic symbolic execution will start with random concrete values or use user input. Let us assume 0 is chosen as value for both `x` and `y`. As before, symbolic variables x and y are also created. The program is executed concretely and finishes on line 9 with a path condition $x \leq 50$. To continue, one of the constraints is negated and a constraint solver is used to find a solution for this alternative. In this case, there is only one constraint to negate, resulting in $x > 50$. A solution is $x = 51$ and $y = 0$.

The program is executed concretely once more with the new input. Now execution ends up on line 6 with path condition $(x > 50) \wedge (y \neq 27182)$. Dynamic symbolic execution negates the latter constraint and a constraint solver provides new values for the final execution of the program. This execution ends on line 4 and the analysis ends.

[10] We note that some authors make no distinction between dynamic symbolic execution and concolic testing.

```
1  int signs_average() {
2      int i = 10;
3      int sum = 0;
4      while (i > 0) {
5          if (getRandomNumber(-100, 100) > 0) {
6              sum = sum + 1;
7          } else {
8              sum = sum - 1;
9          }
10         i = i - 1;
11     }
12     return sum;
13 }
```

Listing 2.3: Block of code executing a random number generator ten times. For every positive random number, `sum` is incremented. Otherwise, `sum` is decremented. The number of paths in this program is $2^{10} = 1024$.

Now consider the following adjustment to Listing 2.2: we change the `if` on line 3 to have `y == hash(x)` as condition. Here the `hash` function is one that cannot be inversed by the constraint solver. Regular symbolic execution thus ends up with a path condition in which the constraint $y = hash(x)$ cannot be solved, assuming it can even be represented. In dynamic symbolic execution, however, there is a concrete value for `x` and thus the function call can simply be executed. Suppose its result is 345, then the constraint solver can instead find a solution for $y = 345$. This is trivially determined to be 345. Thus the analysis can continue.

2.5.3 Limitations

Symbolic execution has some limitations. We discuss path explosion and insufficient constraint solving theory in this section.

Path Explosion

Since analysis forks when encountering a control flow branching statement such as an `if`, the number of paths to analyse can grow rapidly. Consider the example in Listing 2.3. Every call to `getRandomNumber` will cause the creation of a new symbolic variable. Every time the execution reaches the `if` statement, the number of paths to analyse doubles.

This problem is inherent to a program analysis technique such as symbolic execution where the complexity of the program is not abstracted away, but instead encoded in its entirety through the path constraints. This in contrast to a technique such as, for example, abstract interpretation [32] which simplifies the value

domain in order to analyse programs.

A special case of the path explosion problem is one where infinite paths are created. To understand this, consider again the example in Listing 2.3, but now remove line 2 and instead have `int i` be a parameter of the function. Symbolic execution would then create a symbolic variable i for `i`. This in turn means the `while` on line 4 will now also cause the analysis to split up every time it is executed. As there is no limit on the value of the variable i,[11] the analysis can keep rechecking the `while` indefinitely.

Research continues to address the path explosion problem, for example by reusing an earlier analysis of functions [57, 112] or through merging different program states together [165]. In practice, the problem can also be managed by deciding which parts of the program to prioritise, by putting limits on the number of path to explore, by limiting the number of times one loop can be repeated, by adding loop invariants, i.e., extra constraints that limit the values a variable such as i is allowed to have, or, crudely, by limiting the time symbolic execution is allowed to run for. Note that many of these approaches lead to parts of the program simply not being analysed.

Constraint Solving Theory

The constraint solver is a vital part of symbolic execution, solving path constraints to obtain concrete input. As such, symbolic execution is directly limited by which input domains and operations the constraint solver can represent, what it can solve, and how efficiently it can solve it. When constraints cannot be solved, entire parts of the program may remain unanalysed. For example, a constraint solver that can only solve linear equations would fail to solve $2^x = 16$ for x. Another example would be the $hash(x)$ described in Section 2.5.2. Improving constraint solvers is its own field of research and beyond the scope of this dissertation.

2.6 Conclusion

In this chapter, we provided general background and context to the topics that will be discussed in the following chapters. We motivated the use of version control software to manage a project's history. We described the history of version control software to better understand some of the design choices in Git, the most popular version control software in use [79, 123]. We gave an in-depth description of Git's workings and the terminology used in Git as well as the following parts of this dissertation. Specifically, we described commits which

[11] In practice, `i` is limited in Java by the size of an integer. Other languages may not have this restriction.

will be examined closer in Chapter 3 for two types that require more developer time and effort. We described program dependence graphs which we use extensively in Chapter 4. Finally, we described symbolic execution which we employ in Chapter 6.

Chapter 3

Problematic Commits

In Chapter 2, we described how version control software enables managing a project's source code and its history. Developers record their changes into chunks called commits. The commit is thus the unit in a project's history, both for the team developing the project and those reviewing it. Software evolution researchers analysing projects' histories also commonly consider the history in terms of the existing commits.

Some types of commits are harder for developers, reviewers, or researchers: they demand extra time and effort to understand, categorise, and other aspects we will discuss. In this chapter, we go into detail on two such types. First, we discuss composite commits, which are the result of bad practices by developers during the development of a program. Second, we discuss merge commits, which are an inherent part of the usage of version control software in a team setting. Specifically, we consider the (semantic) conflicts that may arise in a merge commit. The later chapters in this dissertation then focus on finding solutions for these two types of problematic commits.

The chapter is divided in three parts. In Section 3.1 we define composite commits. We discuss their negative effects as well as their prevalence. Later on, in Chapter 4, we will discuss our approach to deal with these composite commits. In Section 3.2 we discuss different types of conflicts that occur when merging branches in version control software. In Section 3.3 we look at semantic merge conflicts in particular and discuss their negative effects. We will look into the prevalence of semantic merge conflicts in Chapter 5. In Chapter 6 we will then discuss our approach to detect semantic merge conflicts.

3.1 Composite Commits

Best practice suggests each commit in a project's history should only contain changes related to one task. Such commits are called single-task or atomic commits [42, 155].

However, developers do not necessarily follow the best practice of creating only single-task commits [64, 66, 88, 108, 119, 155]. For example, a small bug may end up being fixed while work is underway on another feature. The bug fix and the new feature then end up in the same commit. Floss refactoring is another problem: refactoring in order to prepare an implementation for a new feature, after which the refactoring changes and the new features changes both are committed together [111]. These situations result in *composite commits*: larger commits that combine many unrelated changes.

3.1.1 Negative Effects

Composite commits may have several negative effects. We provide some examples.

Negative Effects On Reviewers

A code reviewer will have a harder time dealing with larger commits of unrelated changes. A worse understanding of the code leads to lower quality feedback [9]. In a survey at Microsoft, reviewers specifically called out composite commits as being harder to understand [154]. In a survey at Mozilla, reviewers stated five individual commits are easier to understand and considered them to be of a higher quality than a single composite commit [89]. Google has converged on smaller commits, when compared to similar companies, in order to speed up code review [138]. Commits touching more files lead to fewer useful comments by reviewers [18]. Larger commits lead to more rounds of reviewing [12]. When reviewing pull requests, reviewers value the organization of the commits in the pull request [61], i.e., one commit for every subsystem to be changed by the pull request. Reviewers do not like pull requests that tackle different features and fixes [61]. These are harder to review. In fact, such pull requests may be rejected even if their separate parts would have been useful for the project.

As these studies show, composite commits make it harder to review code. This results in requiring more effort and time, but can also lead to a lower quality review. Addressing this issue requires either developers to be more diligent when creating commits or requires the introduction of tools that can do it for them.

Negative Effects On Research

The history of a project under version control is already split into commits. Thus a researcher analysing the project history will turn to use commits as is, rather than defining their own way to split up the history of a project. Using the commits directly also enables the research to be directly applicable to existing projects, without the need for potential extra preprocessing. However, the research may require just commits making a particular change or fitting a certain category. Composite commits pose a problem for this kind of research, forcing the researchers to decide on the "one" motivation for a commit when that commit comprises various unrelated changes. Equally problematic, a researcher may have to go out of their way to untangle the commits before being able to use them in their analysis. For example, Mills et al. [108, 109] describe an approach to bug localisation, i.e., finding the code relevant to a certain bug. Their technique uses commits as a basis and they had to first manually verify all commits to ensure none are composite.

Herzig et al. [66] find that the presence of composite commits impacts bug counting models, i.e., models determining how often a certain file or set of files has caused bugs in the past. Analysing which files are considered to cause defects, Herzig et al. find a difference of 6% to 50% (with a harmonic mean of 17%) when applying a bug counting model on a dataset with composite commits versus one with the composite commits split into atomic commits. When using models to link source files to bug reports, 16% are linked incorrectly due to the presence of composite commits. When training bug prediction models on datasets of commits, they find a median accuracy improvement of 16% after untangling the commits.

Nguyen et al. [119] look into the effect of composite bug fixing commits. Using a dataset of composite commits and one of atomic commits, they analyse the effect of composite commits on two existing approaches from other authors. BugCache [85] is a model for predicting the presence of bugs in commits. Nguyen et al. find the accuracy of BugCache improves by 1% to 32% after removing composite commits. BugLocator [173] considers a bug report and ranks files it thinks are related to that bug report. Removing composite commits led to Nguyen et al. seeing an improvement of 6% to 21% in BugLocator.

Kochhar et al. [88] look into bug localisation techniques. In their dataset they found 28% of files changed in bug fixing commits were unrelated to the bug the commit was fixing. However, Kochhar et al. found no statistical difference in the results of bug localisation after cleaning out these composite commits.

Herbold et al. [64] consider composite commits on a line-based granularity. They do not consider every unrelated line in a composite commit a problem. Instead, they define "problematic" depending on the type of research the commits are being used for, specifically: program repair, bug localisation, or variations of

defect prediction. They estimate that 3% to 47% of the lines in composite bug fixing commits can be considered problematic for the research the commit is being used for.

The effect of composite commits on research varies depending on the field. Certain models seem to be especially affected directly by it, as they rely on clean input to be modelled on. This may pose problems with the rise of machine learning approaches. The ability to split composite commits up into atomic commits tackling a single task each would help improve these models.

Negative Effects On Tool Users

Since the commit is the unit in a project's history, tools aimed at helping a user work with the history are built around the concept of a commit. These tools generally do not consider the existence of composite commits.

Visualisation tools show a tree of commits or the changes contained in one commit. For these tools, a composite commit is still just one commit. An overview of commits will often show just the first line describing each commit. For brevity, the common approach is to limit this first line to about 50 characters. In doing so, the different unrelated changes contained in a composite commit may be obscured if the developer only describes what they consider the "most important" change. This lack of information in the commit overview may not be obvious to someone looking at it. In a detailed view of the commit, the changes are commonly shown as is; there is no distinction made between possibly unrelated changes. In some cases, the commit message will indicate the different tasks that a commit tackles, but this cannot be relied on: Nguyen et al. [119] find 3% to 41% of composite bug fixing commits did not mention all included tasks in the commit message.

Individual changes are more difficult to revert if they are a part of a larger commit. In Git, for example, one can directly apply `git reset` or `git revert` to undo changes at the commit level. Undoing parts of a commit instead requires extra steps for the user.

Furthermore, changes are more difficult to integrate if they are part of a composite commit with other unrelated changes. Again in Git, one can `cherry-pick` individual commits and apply them to other branches of the code. No such mechanism exists for parts of commits.

In conclusion, several tools do not take into account changes on a more fine-grained level than a commit, which can be problematic in the light of composite commits. If a tool can detect composite commits and warn developers about them before they get committed, this alleviates some of the problems described here.

Negative Effects On Merging

The properties that make composite commits problematic also surface when developers merge branches.

A mining study of 66 projects finds that smaller commits facilitate the resolution of merge conflicts [164]. This study also finds that having more separate chunks in a merge reduces the time to merge.

In a survey, by McKee et al. [101], developers state that they break up changed code into smaller, standalone pieces to manage the complexity when resolving merges. In effect, the developers are untangling the merge commit to make it easier to understand. In another survey, by Vale et al. [164], one of the surveyed developers specifically mentions large branches with many features as problematic due to their potential interaction in behaviour. When merging, all the changes from that branch just show up as one parent commit the developer has to deal with, akin to one big composite commit.

In these cases, a tool to untangle either the commits prior to merging or the merge commit itself after the merge has happened would help the developers to understand the different changes involved in the merge.

Summary

These studies show that composite commits result in negative effects for various practitioners. Developers cannot apply their usual tools to work with a single task within a composite commit. Reviewers have a more difficult time to understand composite commits. Researchers may come to incorrect conclusions.

Developers should be encouraged to apply proper practices and avoid creating composite commits. Tools and techniques should be developed to help developers in this.

3.1.2 Prevalence

Composite commits occur on a regular basis. While the different studies we will cite here do not entirely agree on how prevalent composite commits are, the studies do all indicate the real presence of composite commits.

All Commits

Tao and Kim [155] perform a manual review of 453 commits across four Java projects. They find that between 17% and 29% of the investigated commits are composite.

A study by Herzig et al. classifies 7000 commits across five Java open source projects [66]. The study finds that of those commits referencing a bug report, 6% to 12% are composite.

Kawrykow and Robillard [83] look at changes on a method level. They identify refactorings such as renaming, extraction of local variables, removal of a redundant `this`, and others. They find 79% of method updates have refactorings interleaved with changes with a semantic effect.

Bug Fixing Commits

Mills et al. analyse 837 bug fixing commits and the 1344 files changed by those commits [108]. They find just 496 files to be relevant to fixing, though also note another 262 are test files. They find three common causes for an unrelated file to be included in a composite commit: 72 files had only comment changes, 395 files added new code, and 74 files were refactorings.

Kochhar et al. take a similar approach and analyse 498 changed files in 100 bug fixing commits [88]. They find 28% of the changed files were not related to the bug fix.

The study by Herzig et al. [66] mentioned above also considers bug fixing commits separately. In bug fixing commits, the prevalence of composite commits ranges from 7% to 20%.

Nguyen et al. analyse eight open source projects whose commits were previously used in other research. They find that 11% to 39% of all the bug fixing commits were composite [119]. Of those composite bug fixing commits, 3% to 41% had commit messages that did not indicate the presence of multiple tasks being handled.

Finally, Herbold et al. look at bug fixing commits with a line-based granularity [64]. To do this, they crowd source a manual validation of 3498 commits fixing 2328 bugs across 28 projects. This leads to a total of 289904 modified lines getting classified. Herbold et al. conclude just 22% to 38% of changed lines contribute to the bug fix. When ignoring documentation and test changes, this range improves: 69% to 93% of lines contribute to the bug fix.

3.2 Merge Conflicts

In Section 2.3.3 we briefly touched on conflicts that can occur when merging two branches together in Git. Such a merge conflict occurs when different branches are merged during a three-way merge and the textual changes in each branch overlap. Git cannot decide which changes to prioritise and leaves the choice to the developer. Software developers are likely familiar with these Git merge conflicts. The error message of such a conflict is shown in Listing 3.4. Such a merge conflict is, however, not the only type of merge conflict that may occur. The literature also discerns additional conflict types [105, 142, 152, 160]. We specifically detail textual, syntactic, semantic, build, and test merge conflicts in this section. In the

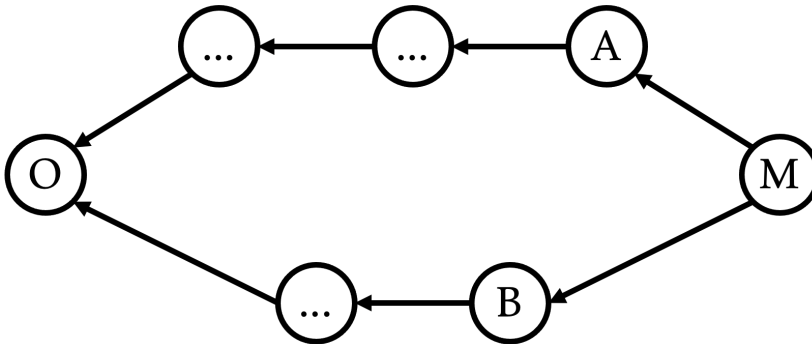

Figure 3.1: The four relevant parts when talking about merging and merge conflicts: the merge commit M, its direct parent commits A and B, and their common ancestor commit O. As in Chapter 2, the arrows indicate a child-parent relation.

rest of this dissertation we use the textual, syntactic, and semantic categorisation of merge conflicts, unless when related work warrants the use of another term.

Textual merge conflicts occur when two or more branches change a line of code in different ways.

Syntactical merge conflicts occur when the textual merge has succeeded, but a resulting merged file is no longer syntactically correct.

Semantic merge conflicts occur when the syntax of the merged files is correct, but the behaviour does not conform to the developers' intentions.

Build and test merge conflicts are characterised by the way they are detected: the former is caught when building the code, the latter when running its tests.

When discussing merges and merge conflicts in the remainder of this dissertation, we will generally consider the merge to have four parts: O, A, B, and M. These four parts are visualised in Figure 3.1. First comes O, the common ancestor commit from which two branches have diverged. We use A and B to refer to the two final commits of both branches before they are merged together. Here we focus on the version of the source code as it is after the commit, not so much the changes making up that final commit. The merge commit M is then where the branches are actually merged together. Merge commit M has A and B as its direct parents. Following the parents back far enough will eventually lead back to O.

3.2.1 Textual Merge Conflicts

```
1 Hello world
```

Listing 3.1: Version O of `welcome.txt` in a textual merge conflict.

```
1 Hello world!
```

Listing 3.2: `welcome.txt` in version A in a textual merge conflict.

```
1 Hello there world
```

Listing 3.3: `welcome.txt` in version B in a textual merge conflict.

```
1 Auto-merging welcome.txt
2 CONFLICT (content): Merge conflict in welcome.txt
3 Automatic merge failed; fix conflicts and then commit the result.
```

Listing 3.4: When attempting to merge versions A and B of `welcome.txt` (see Listings 3.2 and 3.3), Git will not know which version to prioritise. Instead, this error message is shown and the user is expected to choose a resolution themselves.

Textual merge conflicts occur when merging branches that changed the same line in different ways. Consider a file `welcome.txt` as shown in Listings 3.1 to 3.3. Originally, `welcome.txt` contains just one line: `Hello world`. Commit A changes the one line to instead say `Hello world!`. Commit B on the other hand changes the line to say `Hello there world`. When merging both versions together with Git, or with any similar version control software, it will compare the three versions to decide on a merge candidate of the file that incorporates all changes from both the branches. As the same line is changed in different ways, however, Git cannot deduce whether the final file should contain the line `Hello world!`, the line `Hello there world`, both lines, or some combination of the two lines. Rather than trying to guess at a resolution, Git stops trying to merge the file, puts both choices in the file, prints an error such as in Listing 3.4, and lets the developer make the decision instead. Once a resolution is decided on, i.e., the developer edited the merge candidate file themselves so that it looks as desired, the developer manually finishes the merging process.

Textual merge conflicts remain a problem to this day [55, 93, 101, 164] as well as an object of research [100]. Recent efforts include using machine learning techniques to either predict textual merge conflicts [20, 124] or to have the model create the merge candidate [44, 126, 153].

3.2.2 Syntactic Merge Conflicts

One can consider a syntactic merge conflict in two ways. On the one hand, a

```
1 if (pcfProperties.getInstanceCertificate() != null) {
2   builder.instanceCertificate(new ResourceCredentialSupplier(
3     pcfProperties.getInstanceCertificate()));
4 }
```

Listing 3.5: Version O in a real-world syntactic merge conflict. Code extracted from the Spring Cloud Config project [148], merge 81585fe09e5ffb7-0708e4c6b2767bb2af73ecc5c.

```
1 if (pcfProperties.getInstanceCertificate() != null) {
2   builder.instanceCertificate(new ResourceCredentialSupplier(pcfProperties.
      getInstanceCertificate()));
3 }
4 else {
5   builder.instanceCertificate(new ResourceCredentialSupplier(resolveEnvVariable
      ("CF_INSTANCE_CERT")));
6 }
```

Listing 3.6: Version A in a syntactic merge conflict. A newline is removed and an `else` is added.

```
1 if (pcfProperties.getInstanceCertificate() != null) {
2   builder.instanceCertificate(new ResourceCredentialSupplier(
3     pcfProperties.getInstanceCertificate()));
4 }
5 else {
6   builder.instanceCertificate(new ResourceCredentialSupplier(
7     resolveEnvVariable("CF_INSTANCE_CERT")));
8 }
```

Listing 3.7: Version B in a syntactic merge conflict. The same `else` is added as in version A, but with a newline after an opening parenthesis.

```
 1 if (pcfProperties.getInstanceCertificate() != null) {
 2   builder.instanceCertificate(new ResourceCredentialSupplier(pcfProperties.
       getInstanceCertificate()));
 3 }
 4 else {
 5   builder.instanceCertificate(new ResourceCredentialSupplier(resolveEnvVariable
       ("CF_INSTANCE_CERT")));
 6 }
 7 else {
 8   builder.instanceCertificate(new ResourceCredentialSupplier(
 9     resolveEnvVariable("CF_INSTANCE_CERT")));
10 }
```

Listing 3.8: Version M in a syntactic merge conflict. Git does *not* consider this a textual merge conflict and completes the merge without problems. The resulting piece of code, however, has two `else` branches.

textual merge can complete successfully, but the resulting code in M is not syntactically correct. An example of this is given in Listings 3.5 to 3.8. This code is extracted from merge 81585fe09e[1] from the Spring Cloud Config project [148]. In this example, version A slightly adjusts the consequent of an `if` statement and also adds an `else`. Version B does not change the consequent, but adds a slightly different `else`. When merging, both `else`s appear in the resulting merge M. On the other hand, merging can be approached from a structured point of view rather than purely textual: the syntax of the code in question is taken into consideration to create the merge candidate M. A failure to perform such a structured merge would then be a syntactic merge conflict.

Structured merging [25, 171] can sidestep some textual merge conflicts. Consider, as an example of structured merging, a function call `foo(1, 2)` in version O of a three-way merge. Version A changes the call to `foo(3, 2)`. Version B changes the call to `foo(1, 4)`. The commonly used textual merge algorithms would not try to merge this and instead report a textual merge conflict. A tool aware of the syntax of the language in question, however, can conclude that the two arguments are distinct parts and create a merge where the line reads `foo(3, 4)`. As another example, consider a Java class where versions A and B each add a different interface to the class. A syntax-aware tool could conclude that both interfaces can be added to the class in the merge.

This structured approach to merging software code has been around since the 1990s [25, 171]. While the research field has remained active [1, 4, 5, 8, 76, 99, 121, 139, 141], the adoption of structural merging tools in the industry seems to lag behind. We hypothesise that this is due to the need of requiring a merging specification that covers the entirety of the languages one would want to use in a project. Apel et al. [5] have tried to mitigate this issue by combining structured, completely syntax-based, merging with textual merging. The idea in what Apel et al. call “semi-structured merging” is to have a merging specification that starts at the top level of the languages, e.g., classes, their methods. The parts of the language not covered by this merging specification remain represented as textual nodes in the larger syntax tree. Thus, for example, a class and its methods could be explicitly represented in the syntax tree, but the method body remains just a blob of text. Changes within a method’s body would then be merged using a textual merging algorithm. The approach taken by Asenov et al. [8] works as a “plugin” into Git, potentially making adoption easier. They provide a generic tree merging algorithm, which can, assuming the presence of a parser for the language, also handle abstract syntax trees.

[1] Full SHA-1 hash is 81585fe09e5ffb70708e4c6b2767bb2af73ecc5c.

3.2.3 Semantic Merge Conflicts

In semantic merge conflicts the merge encountered no issues on a textual nor on a syntactical level. Instead, the behaviour of the merge is no longer in line with what versions A or B expected.

Consider the case of Listing 3.9, showing version O of a semantic merge conflict. The displayed method adds `x` and `y` together and returns the result. One developer decides that there is an off-by-one error and changes line 2 to `int z = x + y + 1` in version A, see Listing 3.10. Another developer meanwhile also had the same idea and changes line 3 to `return z + 1` in version B, see Listing 3.11. Listing 3.12 shows version M. In version M the changes in A and B result in both lines 2 and 3 adding an extra `+ 1`. Versions A and B have the same semantic change and behave in the same manner, i.e., both versions add one to the sum of both arguments. In version M that behaviour has changed in a way that is no longer as intended. Now two gets added to the sum of the arguments.

As with syntactic merging, one can approach semantic merging from two different angles: creating a valid merge candidate or validating an existing merge candidate.

Creating a valid merge candidate is an approach similar to the structured merging aimed at syntactic merge conflicts mentioned in Section 3.2.2. Berzins [16] achieves this by means of a formal definition of semantic merging in a language-independent manner. If conditions in this definition are met, it describes how to construct a semantically conflict-free merge candidate M. Much like with structured merging, a fallback is required: if the conditions are not met, the approach does not create a merge candidate at all. Another downside is that each program must also be converted to the representation used by Berzins.

On the other hand, one can fall back to simply performing a textual merge, like in Git, or a syntactical merge, like in structured merging. The resulting merged code can then be semantically validated afterwards [35, 117, 145, 152, 160, 172]. In Chapter 6 we will further consider this second approach to semantic merge conflicts, discuss the cited related work, as well as detail our own proposal.

3.2.4 Build Merge Conflicts

Build merge conflicts are conflicts caught by the compiler [36, 160] when building the code. This includes not only the syntactical conflicts described in Section 3.2.2, but also some of the conflicts that we think of as semantic merge conflicts. Consider, for example, the simplified situation depicted in Listings 3.13 to 3.16, representing versions O, A, B, and M of a build conflict. A parameter `x` is renamed in version A to `n`. The usage of `x` is changed accordingly. In version B a developer adds a usage of the variable `x`. In version M, the renaming from version A is propagated, but the use of `x` added in B is also still present. A compiler

```
1 public int myAdd(int x, int y) {
2   int z = x + y;
3   return z;
4 }
```

Listing 3.9: Version O in a semantic merge conflict. `myAdd` sums two integers.

```
1 public int myAdd(int x, int y) {
2   int z = x + y + 1; // Modified
3   return z;
4 }
```

Listing 3.10: Version A in a semantic merge conflict. `myAdd` sums two integers and adds one.

```
1 public int myAdd(int x, int y) {
2   int z = x + y;
3   return z + 1; // Modified
4 }
```

Listing 3.11: Version B in a semantic merge conflict. `myAdd` sums two integers and adds one.

```
1 public int myAdd(int x, int y) {
2   int z = x + y + 1; // Modified in A
3   return z + 1;      // Modified in B
4 }
```

Listing 3.12: Version M in a semantic merge conflict. `myAdd` sums two integers and adds one, then adds one again.

would catch this issue. In a language without a compiler, this may not be caught until the code is run. Some linters[2] can also catch this error in a language without a compiler. As such, a build merge conflict's definition relies on the language and tools used.

Some research focuses specifically on build conflicts. Due to the overlap with syntactic and semantic merge conflicts, we do not discuss this separately here.

3.2.5 Test Merge Conflicts

Test merge conflicts are found when testing the code in question [160]. Specifically, a test failing on the merged code of version M that was not failing before the merge. This also includes failing behaviour for which there was no test and which was thus not caught until encountering it as a bug in the program.

As with build merge conflicts, test merge conflicts also overlap with both syntactic and semantic merge conflicts. Also much like build merge conflicts, they end up being defined in function of the language and tools used. The semantic merge conflict example given in Section 3.2.3 (Listings 3.9 to 3.12) is also a test

[2]The term "linter" is generally used for tools using simple static analysis to catch mistakes in source code.

```
1 public int inc(int x) {
2   return x + 1;
3 }
```

Listing 3.13: Version O in a build conflict. inc increases the argument by one.

```
1 public int inc(int n) {
2   return n + 1;
3 }
```

Listing 3.14: Version A in a build conflict. The parameter x is renamed to n.

```
1 public int inc(int x) {
2   log(x);
3   return x + 1;
4 }
```

Listing 3.15: Version B in a build conflict. The value of x is logged.

```
1 public int inc(int n) {
2   log(x);
3   return n + 1;
4 }
```

Listing 3.16: Version M in a build conflict. The parameter x is renamed to n. The old parameter x is still logged.

merge conflict. The example given in Section 3.2.4 (Listings 3.13 to 3.16) is a build merge conflict in Java. In a language like Python or JavaScript, there is no compiler. Certain code linters might catch the issue, but if this is not the case, then this error would not be detected until tested for it. It can be debated that a test conflict should be considered a build conflict depending on which tools are used at what point in the software development process. Similarly, incorrect syntax may not actually be caught until the incorrect file is actually loaded into the interpreter in a language without a compiler. This in turn can thus change a build conflict into a test conflict.

The murkiness of the definitions is further evidenced by Shen et al. [142] who classify conflicts caught when building the tests as test conflicts. Those conflicts are still errors caught by the compiler, but it was the tests that were not getting built during the build phase in Shen et al.'s setup.

3.2.6 Summary

Some merge conflict definitions overlap, as depicted in Figure 3.2. A build conflict could be either a syntactic or a semantic merge conflict. Similarly a test conflict too could be either a syntactic or a semantic merge conflict. Both build and test conflict definitions also rely on the languages and the tools involved. These categories are based on the symptom, e.g., a compiler error, a test failing, rather

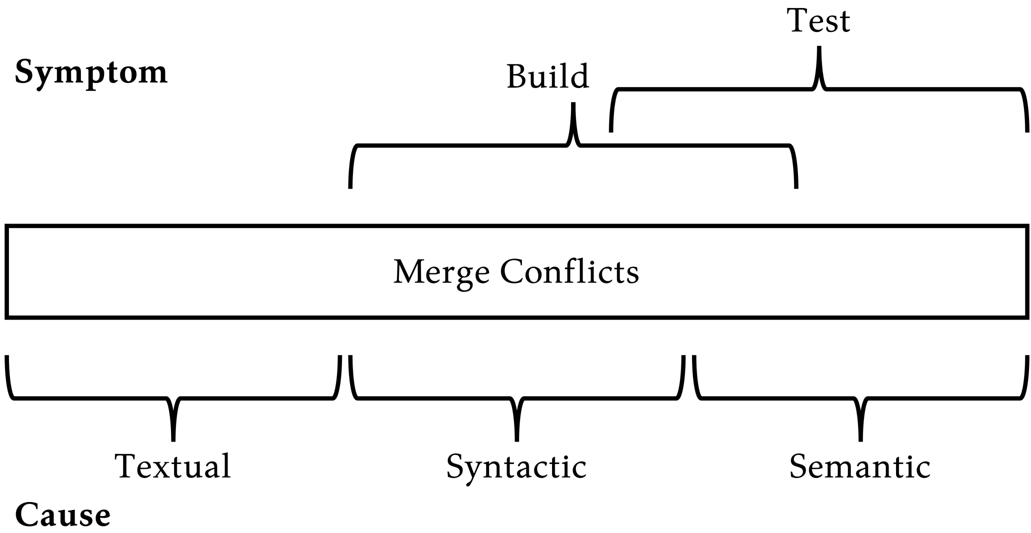

Figure 3.2: Common categories when describing merge conflicts. Textual, syntactic, and semantic focus on the cause, while build and test focus on the symptom. As such, build and test can overlap depending on the context.

than on the cause.

We prefer to focus on the cause, not the symptom. As such, we avoid the build and test merge conflict categories and instead stick to textual, syntactic, and semantic merge conflicts.

3.3 Semantic Merge Conflicts

In general, the version control software or the compiler will warn a developer about any textual and syntactic merge conflicts. Detecting these does not require extra effort for the developer, even if resolving the conflict may do so. For semantic merge conflicts, there is no such widespread ready-made solution. A good and complete test suite can mitigate the issue, but this too requires developer time and effort. As such, we will focus on semantic merge conflicts in more depth in this section as well as proposing our approach to detecting them in Chapter 6.

As detailed in Section 3.2, a semantic merge conflict occurs due to an unexpected interaction of the behaviour in both sides of a merge. Semantic merge conflicts cost developers time to identify and fix. Even in their absence, developers still have to verify that the merge is conflict-free.

A developer taking part in a survey by Vale et al. [164] mentions:

> The worst problems are when Git does not detect a merge conflict because the change appears to merge cleanly, but then bugs are introduced.

Another developer in the same survey [164] says:

> My harder conflicts are often when integrating two different large feature branches, the tests may at most ensure that specific isolated scenarios keep working, not that the involved features interact well, until newer tests are written for that purpose.

Similarly, a Microsoft developer taking part in a survey by Bird and Zimmermann [17] says:

> [Semantic merge conflicts] tend to be subtle because they often are not noticed for a while when totally bizarre behaviour occurs and it takes a long time to track down what happened.

In this section, we discuss how developers perceive semantic merge conflicts as well as the effects that semantic merge conflicts have on developers. We do not go into the prevalence of semantic merge conflicts here, instead deferring this to Chapter 5 where we describe our own study on the prevalence of syntactic and semantic merge conflicts. We also discuss related studies in Chapter 5.

3.3.1 Conflict Difficulty

In surveys, developers find the difficulty of merges and (semantic) merge conflicts is influenced by a few factors: the perceived complexity of the merge [101], whether or not the developer has expertise on the code being merged [101], the number of changed lines [164], and then number of changed files [164]. For textual merge conflicts, some developers also cite semantic differences between branches to increase the difficulty of finding a resolution [164].

Vale et al. [164] find that if the different blocks of changed code in a textual merge conflict are more semantically related, then more time is taken to resolve the conflict. After all, if the different blocks of conflicting code are semantically related, then any semantic differences will have a larger impact across the textual conflict. They find that semantic conflict is a better indicator than the other metrics that they have analysed.

Semantic merge conflicts are difficult to understand. Even non-semantic merge conflicts are more difficult to understand when there are semantic differences to contend with. Tools that pinpoint the causes of semantic merge conflicts or that warn about semantic differences between branches can help developers in managing this difficulty.

3.3.2 Conflict Resolution

Nelson et al. [118] find that developers approach merge conflicts by means of (in decreasing order of use): (1) examining the merge, i.e., looking at all of the changes in the branches; (2) analysing the code, for example by means of a debugger, or manipulating the code, i.e., making changes and seeing their effect; or (3) examining the code, i.e., looking at the final resulting code. In order to resolve a conflict, McKee et al. [101] find that developers require an understanding of the code, contextual information about the conflict, and tools presenting relevant information.

Brindescu et al. [21] perform an in-depth investigation into how developers fix various merge conflicts, Brindescu et al. find that developers often get stuck on finding the right information or on understanding the information once they come across it.

Unexpected conflicts require additional developers and resources [118]. In a survey of 124 Microsoft developers, Bird and Zimmermann [17] find the average respondent spends 5.45 hours per month integrating changes from branches. The 95th percentile is as high as 15.45 hours, which they explain by teams at Microsoft often having a person dedicated to handling integrations. Bird and Zimmermann find most of this time is spent verifying the correctness of merges and resolving conflicts.

Understanding a semantic merge conflict is key. A tool that describes the semantic merge conflict and points the developer in the right direction thus helps minimise the effort spent on resolving the conflict.

3.3.3 Solution Validation

When a developer believes that they have resolved the conflict, they still need to be able to verify that the conflict really is resolved. Lacking dedicated tools or tests, this can prove difficult.

Brindescu et al. find solution validation is a step developers often get stuck on [21]. Approaches to validating their solution tended to mostly be building the program [17, 118], running the program [17], running the tests [17, 118], and even eyeballing the code to decide whether it looks correct [118].

"Using dedicated tools" was noticeably not a top answer by respondents to the surveys (by Bird and Zimmermann [17] and Nelson et al. [118]) when asked how they validate a merge. Tools that were used included generic static analysis tools, none of which were dedicated to merge conflicts [118].

These findings indicate the lack of a widespread real-world tool that helps developers with semantic merge conflicts.

3.3.4 Behavioural Change

The difficulty and effort of dealing with merges and merge conflicts leads to behavioural changes in the developers as well as structural changes in the way teams work.

Bird and Zimmermann [17] describe anti-patterns that occur due to the difficulty of working with branches.[3] These anti-patterns include, but are not limited to: merge paranoia, avoiding or deferring merging due to fear of consequences; merge mania, spending too much time on merging; and development freeze, stopping development activities because a merge is happening or about to happen.

Nelson et al. [118] investigate reasons for deferring a merge or deferring conflict resolution. Developers often cite the complexity of the code involved or having to fix code in many different locations. Not owning the code at fault was also an important factor. Said one person about deferring resolution: "Untangling takes days instead of minutes when it gets too out of hand."

de Souza et al. [40] analysed the workflow of a team of developers at the National Aeronautics and Space Administration (NASA) in the United States of America. Part of the workflow required a developer to inform the rest of the team by email when code was added to their main branch. The email described the changes and had an assessment of the impact the changes might have on which people and which parts of the project. This enabled others to anticipate integrating the code with their own code. Such an approach was also encountered by Nelson et al. [118] in other projects, be it through sending email or through holding weekly meetings.

Other organisations changed policy to avoid bad integrations near major releases [118]. For example, some organisations require everyone to be available during the two weeks prior to a release [118]. Policy changes at Microsoft and extensive time spent validating merges make conflicts relatively rare according to their developers [17].

Merge conflicts cause enough trouble that organisations consciously try to work around them. Better tools could mitigate this to some extent by minimising the effort required.

3.3.5 Tool Support

Vale et al. [164] identify four overarching challenges for merge conflict resolution:

1. Lack of coordination. This includes aspects such as communication between team members, following best practices when it comes to committing code, and a well-defined development process.

[3] Bird and Zimmermann [17] credit Appleton et al. [6] with the definition of these anti-patterns. We were, however, unable to find this information in the cited document.

2. Lack of tool support. This includes dedicated merge conflict tools, but also making use of a proper issue tracker or ways to diff code.

3. Flaws in the system architecture. This can be due to the code being too tightly coupled or due to project accruing too much technical debt.

4. Lack of testing suite or pipeline for continuous integration.

Two of these four challenges (challenge 2 and 4) involve tools. Usage numbers find many projects are lacking in these aspects. Just one in four developers proactively monitor for merge conflicts [118]. Of those proactively monitoring, one in three uses continuous integration [118]. Again of those proactively monitoring, just one in four use code analysis tools although none of the tools seem tailored specifically to merge conflicts [118].

Despite these low numbers, developers do desire better tools. Developers end up maintaining custom tools and scripts [17]. Developers want tools to show relevant information to resolve and validate merge conflicts [101]. Developers mistrust tools that obscure the steps the tool takes or the rationale the tool has for a certain result [101]. In a survey by Nelson et al. [118], one participant stated an explicit desire for a semantic diffing tool. Another participant would already be satisfied with a well-presented way to diff, not even necessarily semantically, the different versions in the three-way merge to their common ancestor O.

Developers want better tools. Even in situations where tools exist, software projects do not always use them to the fullest. This could be simply due to developers being unaware of the existing tools or due to having deemed the tools to be insufficient or too complicated.

3.4 Conclusion

We discussed two types of problematic commits: composite commits, which contain many changes handling different tasks, and merge commits, which may give rise to different types of merge conflicts, semantic merge conflicts in particular. We described how both types of commits have negative effects on developers and other practitioners due to their increased complexity. We also discussed the prevalence of composite commits. In Chapter 5 we will look into the prevalence of semantic merge conflicts in greater detail.

This chapter motivated the need for tools that help developers with the two discussed types of problematic commits: their complexity leads to more time and effort spent. In Chapter 4, we propose an automated approach to untangling composite commits into smaller single-task commits. This enables developers to split up existing composite commits or be warned about them prior to creating them. In Chapter 6, we propose an automated approach to detecting semantic

merge conflicts. This approach can also be used to validate the resolution of a semantic merge conflict.

Chapter 4

Untangling Composite Commits Using Program Slicing

Version control systems (VCS) are widely used to manage the history of code bases (see Sections 2.2 and 2.3). In a VCS, a developer saves changes into units called commits. Best practice suggests each commit should only contain changes related to one task. Such commits are called single-task or atomic commits [42, 155].

In Section 3.1, we described how developers do not necessarily follow the best practice of creating only single-task commits [64, 66, 88, 108, 119, 155]. For example, a small bug that gets fixed while work is underway on another feature. Bug fix and new feature then end up in the same commit. These situations result in composite commits: larger commits that combine many unrelated changes.

Composite commits may cause several problems for developers, tool users, reviewers, and researchers due to being harder to understand, analyse, or categorise. We described these different situations in Section 3.1.1 and concluded all would benefit from dedicated tools to detect and untangle composite commits. We discussed existing research into the prevalence of composite commits in Section 3.1.2 and concluded that composite commits are not just a hypothetical problem, but do commonly occur in the real world.

In this chapter, we discuss our approach to identifying and untangling composite commits into smaller, single-task commits. Our approach extends program slicing (see Section 2.4.2) over program dependence graphs (see Section 2.4.1) in order to apply it to source code changes. We refine an existing dataset of commits from five Java projects. We evaluate our approach on this refined dataset. The results indicate that our approach is able to categorise commits as single-task or composite. The results also indicate that our untangling approach produces more fine-grained commits than single-task commits.

This chapter is structured as follows. Section 4.1 sketches our approach, evaluation, and enumerates contributions. Section 4.2 discusses related work. We detail the components of our technique in Section 4.3 and the dataset used in its evaluation in Section 4.4. The evaluation method and its results are presented in Section 4.5. We discuss potential future work in Section 4.7.

4.1 Proposed Solution

Despite the problems caused by composite commits, developers continue to create them due to the short term gain in time: creating one big commit takes less time than figuring out which changes belong to which task and committing each task individually. Tool support is required to identify commits as composite and to decompose them into single-task commits. The first type of tool suffices to warn developers that they are about to commit unrelated changes. The second type of tool is also of use to researchers analysing the individual tasks a composite commit is composed of.

We propose program slicing as a foundation for such tool support. In Section 2.4.2, we discussed program slicing: a program analysis that answers questions about the influence of certain program statements on other program statements [143, 169, 170]. We extend this idea in this chapter: our foundation for tool support applies program slicing to changes to the abstract syntax tree. We hypothesise that *related changes affect source code from the same program slice* and thus that *a commit may be decomposed into related changes using the created program slices*. Intuitively, this states that changes that belong together also have control or data dependencies between their changed subjects.

To analyse our approach, we make use of a dataset of commits stemming from five Java projects, gathered by Herzig and Zeller [67]. We first analyse this dataset and refine it further to fit the context of this work.

Our results indicate that slicing on changes to the abstract syntax tree largely meets the stated goals. Our technique is able to categorise commits as single-task or composite. When untangling the composite commits into their individual tasks, our technique at times produces more fine-grained results. That is to say, our technique may untangle into several different tasks that should actually belong together as one task. As an example, consider a composite commit comprising two parts: a and b. Our technique might split this composite commit into three parts instead: a, b_1, and b_2, where b_1 and b_2 together make up task b. This is still better than the alternative in which the technique considers (parts of) different tasks to belong together. For example by untangling that same composite commit into (1) $a \cup b_1$ and (2) b_2. As such, the results of our technique still prove useful for reviewing, reverting, or integrating code.

Specifically, we make the following contributions:

1. A technique to slice around changes made to an abstract syntax tree. The technique considers which parts of the abstract syntax tree are affected by a change. The technique then slices around the corresponding nodes in a program dependency graph. Overlap in the slices is used to decide whether changes belong together.

2. The application of this technique to decide whether a commit handles a

single task or whether the commit is composite.

3. The application of this technique to identify different parts of a commit to untangle composite commits.
4. The refinement of an existing dataset of commits from five Java projects. We perform an automated cleaning step and a manual verification of the commits.
5. An evaluation of our approach using the refined dataset.

4.2 Related Work

We present the related work in two main categories. All related work attempts to create single-task commits. Identifying composite commits is a by-product of this process. The first category of related work uses some form of semantic information derived from the changes or the rest of the source code. These approaches thus attempt to derive dependencies between different pieces of code. Our approach falls in this first category. The second category of related work instead uses purely textual and syntactical information. These can include information such as the distance between changes in a file or looking for string similarities. Some look for patterns where a found pattern is immediately used to group together similar changes or where a pattern is used as input data to inform future decisions.

4.2.1 Semantics-Based Commit Untangling

Barnett et al. [11] use what they call *diff-regions* to untangle commits. Diff-regions are the result of performing a textual difference between two versions and splitting up the result such that each block of textual changes stays within one method or within one type. Barnett et al. work with limited information: only the before and after of changed files are provided in one changeset, i.e., one commit. They do not have access to the entire program. Instead they make use of any method definitions that are present. Two diff-regions are linked (a) if they belong to the same method, (b) if one diff-region uses a method whose definition appears in another diff-region, or (c) if both diff-regions use a method defined in a file present in the changeset. Their technique is evaluated on a single closed-source project.

Luna Freire et al. [98] lift Barnett et al.'s technique from C♯ to Java. Luna Freire et al. evaluate their Java implementation on open source Java projects and confirm Barnett et al.'s results. The tool by Luna Freire et al. is made open source as opposed to the closed-source tool by Barnett et al.

Our work differs from Barnett et al. [11] and Luna Freire et al. [98] in that we make use of more than just the relation between definitions and uses. As we make

use of a program dependence graph, all control and data flow dependencies are mapped, providing a richer picture.

Tao and Kim [155] propose an approach using the ZeroOneCFA points-to analysis built into the T.J. Watson Libraries for Analysis (WALA). Their approach then slices in the result of the points-to analysis. In their approach, a change is split up into line-based changes. Their work uses more than just program slices to determine which changes are related. All formatting changes, for instance, are considered as one separate group of changes. In addition, they also employ a pattern-based approach. String comparisons are used to relate, for example, the addition of a `.clone()` method call in several locations without any static dependencies. While this string comparison adds some extra relations between changes, it does not always prove beneficial: the approach sometimes relates changes that do not belong together. We consider more fine-grained changes on the abstract syntax tree, while Tao and Kim look at line-based changes. We focus specifically on the program slicing in isolation, without the patterns, and analyse how well program slicing behaves on its own. Tao and Kim only look at the results of their combined analysis.

Pârtachi et al. [127] create what they name a multi-version name flow graph (δ-NFG). A δ-NFG combines the program dependence graph of multiple versions of a file into one graph. The graph also indicates the changes that occurred between the different versions. The graph incorporates edges indicating the flow of the names of variables and similar program constructs. Within one commit, every changed statement is at first considered atomic. For every change, the 1-hop neighbourhood is considered. For every change, the graph with the node representing that change and the direct neighbours in the δ-NFG are considered. These 1-hop neighbourhood graphs are clustered based on their similarity, using the Weisfeiler-Lehman graph kernel. Once none of the graphs meet a certain threshold of similarity any more, and thus no more new clusters can be created, the obtained clusters represent the untangled changes.

Chen et al. [30] create two control flow graphs: one for the program before the change and one for the version after the change. Every node contains the actual code statement, the function it is a part of, and whether the statement is changed between versions. They augment each control flow graph with data flow edges, name flow edges (similar to Pârtachi et al. [127]), and subtoken cooccurence edges, i.e., whether two names contain the same subtokens, such as `Box` in `openBox` and `closeBox` methods. Both control flow graphs are then merged together by merging nodes with the same statement within the same function. Finally, Chen et al. use the same bottom-up approach as Pârtachi et al. [127], clustering nodes representing a change based on k-hop neighbours using affinity propagation clustering. Again the clusters represent untangled changes.

Li et al. [94, 95] look into the problem of slicing a project's history for a certain feature. They approach this by considering all tests covering that feature. Their

goal is a minimal version of the history such that the tests still pass. The general idea is that their technique removes parts of the history, then checks whether the tests still pass. They use two techniques to do so. In the first technique [94], this is done by considering which methods are called when the test is run, collecting all changes affecting the methods as well as code referenced by it, and running the tests once to verify. In the second technique [95], the history is instead partitioned and the tests are run on partitions to find changes preserving the test results. This process is repeated to determine a minimal history.

Arima et al. [7] try to achieve the joining of commits rather than untangling them. They consider the issue of one task being spread across multiple commits. To detect the issue, they construct a weighted directed graph across two commits. Nodes are the methods. A method present in both commits will have two different nodes, one for each commit. Edges are based on whether methods are the same, whether one calls the other, or whether they are defined in the same class. Depending on the distance between two methods in the graph, the approach by Arima et al. decides whether the different commits should have been one commit.

4.2.2 Syntactic- or Textual-Based Commit Untangling

Herzig and Zeller [67] combine many different metrics by means of confidence voters. They look at changes on the level of addition and removal of method calls and method definitions, which is more coarse-grained than our approach. Herzig and Zeller combine various metrics to decide similarity. Their approach considers, for example, the distance within a file, the similarity in package names, or the distance in a call graph. Rather than a mix of such ad hoc techniques, we base our work on one fundamental assumption: changes belonging to one task relate by means of an overlap in slices in the program dependence graph.

Kreutzer et al. [91] look at the changes in two different ways: line-based, by making use of the `diff` tool, as well as fine-grained by making use of ChangeDistiller [48]. To match changes together, the approach by Kreutzer et al. considers a string representation of these changes. It then looks for the longest common subsequence to determine similarity in changes. This information is finally used to cluster changes together. We too make use of change distilling, but our methods of defining similarity strongly differ. We rely on the dependencies that show up in the program dependence graph, not on patterns.

Just like Kreutzer et al., Kirinuki et al. [87] also use the longest common subsequence to look for patterns. They analyse the changed program statements between many revisions. For each duo of subsequent revisions, they consider the longest common subsequence of the program's statements. Differences the two revisions have compared to that longest common subsequence are extracted as a pattern template. The pattern template is stored in a database. When a new commit is made to the project for which such a database has been constructed,

the new commit can be compared to the patterns in the database. If a pattern matches a strict subset of the entire commit, then the commit is marked composite. This differs from our work in the same way as the previous paragraph, we do not look for patterns in the changes.

Wang et al. [167] break a change into fine-grained changes with GUMTREE [46], an approach we also follow, albeit with another tool. Wang et al. group changes together by using def-use relations, similarly to Barnett et al. [11]. They improve on Barnett et al.'s method by also defining a similarity metric between fine-grained changes. This metric further groups changes that change code in a similar manner. They reason this handles situations where a similar code change, like a refactor, is applied in many unrelated changes.

A completely different approach is performed by Dias et al. [42]. They make use of a change logger, a program that needs to be installed by the developers prior to their programming. A change logger tracks fine-grained changes as they are made. Dias et al. then link the fine-grained changes together based on attributes such as for example proximity in time, proximity in history, i.e., did many other changes happen in-between, or whether the changes happen within the same class. They employ a form of machine learning to use these metrics to decide whether commits are composite. We purposefully did not opt to use change loggers. Requiring a change logger renders the troves of repositories already out there useless. Even with yet-to-be-developed software it may not be feasible to require its developers to install appropriate change loggers. Privacy reasons could be cited, the developer might not be using the appropriate IDE, or the developer may just find it too cumbersome to set up the change logger.

4.3 Overview of the Approach

We want our technique to take as input a commit that needs to be analysed. We want our technique to output the clusters of related changes that the technique considers the commit to comprise. Each of these clusters is then one untangled single-task commit. To achieve this, our technique consists of four major parts, as depicted in Figure 4.1. First, the commit is distilled into fine-grained changes to the program's abstract syntax tree (AST). Second, an inter-procedural program dependence graph is created for every file in the commit. We will refer to an inter-procedural program dependence graph as a *system dependence graph* (SDG). Third, for every fine-grained change, our technique slices on it in the system dependence graph. Finally, changes are grouped by means of the slices they belong to.

We have implemented our technique for commits to Java programs. We provide a high-level overview of the implementation in Algorithm 1. This too follow the structure of this section.

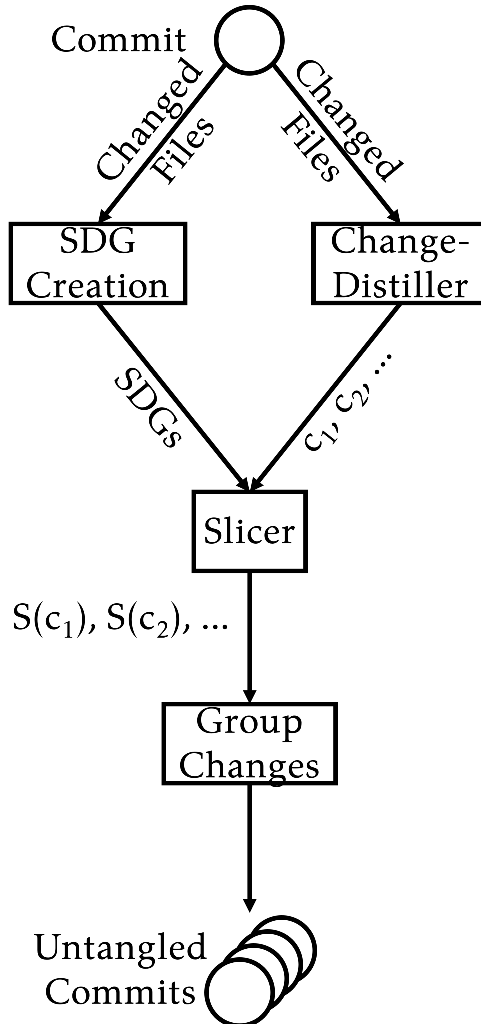

Figure 4.1: Overview of our approach. The input is a commit, the output the untangled single-task commits that make up the commit. The four main parts of our approach are each described in detail in Section 4.3.

```
1  Function untangle(c: Commit): Set[Set[Change]] is
2      files ← changedFilesPreCommit(c);
3      clusters ← ∅;
4      foreach f ∈ files do
5          sdg ← createSDG(f);
6          changes ← changeDistiller(f);
7          slices ← ∅;
8          foreach change ∈ changes do
9              node ← getPreCommitNode(change);
10             slices ← slices ∪ (change, sliceBackwards(node, sdg));
11         end
12         partition ← partitionBy(slices, changeInSliceTransClosure);
13         clusters ← clusters ∪ partition;
14     end
15     return clusters;
16 end
```

Algorithm 1: High level overview of the untangling algorithm. Given a commit, the algorithm untangles the commit into different clusters of related changes.

The rest of this section provides further detail into each of the four steps from Figure 4.1. We will use the example difference of source code shown in Listing 4.1 as a running example. The example in question is part of composite commit 5e2cdc06[1] of the ArgoUML project, trimmed for the sake of this example.

4.3.1 Fine-Grained Change Distilling

In step one, we make use of a change distiller. When working with a project, we have commits at our disposal, i.e., line-based changes. For our approach, we instead want fine-grained changes on an abstract syntax tree. A change distiller can be applied on commits to split them into more fine-grained changes following some algorithm. Another option to obtain fine-grained changes would be by means of a change logger, a program tracking everything a developer does as they work on the code. However, this would require the logger to be installed on developers' machines beforehand. While possible within a company, this approach

[1] Full identifier is 5e2cdc061a0572d4007f4bc84382fff80f29e726. Note that this identifier does not match up with what may be found online at the ArgoUML project. At the time of the creation of the dataset (see Section 4.4), not all projects were managed by Git. Herzig and Zeller converted these other repositories to Git themselves, so this identifier only makes sense within their dataset.

```
1    public FigActionState() {
2 -    _bigPort = new FigRRect(10 + 1, 10 + 1, 90 - 2, 25 - 2, Color.cyan, Color.
       cyan);
3 +    bigPort = new FigRRect(10 + 1, 10 + 1, 90 - 2, 25 - 2, Color.cyan, Color.
       cyan);
4 -    _bigPort.setCornerRadius(_bigPort.getHalfHeight());
5 +    bigPort.setCornerRadius(bigPort.getHalfHeight());
6 -    _cover = new FigRRect(10, 10, 90, 25, Color.black, Color.white);
7 +    cover = new FigRRect(10, 10, 90, 25, Color.black, Color.white);
8 -    _cover.setCornerRadius(_cover.getHalfHeight());
9 +    cover.setCornerRadius(cover.getHalfHeight());
10 -   _bigPort.setLineWidth(0);
11 +   bigPort.setLineWidth(0);
12 -   addFig(_bigPort);
13 +   addFig(bigPort);
14 -   addFig(_cover);
15 +   addFig(cover);
16   }
```

Listing 4.1: Difference view for commit 5e2cdc06 of ArgoUML. Used as running example for Section 4.3. The variables `_bigPort` and `_cover` are renamed to `bigPort` and `cover`.

comes with privacy concerns for the user and is not a feasible approach for many researchers.

To distil the changes from a commit, we make use of ChangeNodes [149]. ChangeNodes is an implementation of the ChangeDistiller algorithm [48]. The ChangeDistiller algorithm in turn is based on work by Chawathe et al. [29]. The ChangeNodes implementation operates on the abstract syntax tree (AST) of a Java program. The AST in question is created using the Eclipse Java Development Tools (JDT). Given two versions of a program, ChangeNodes performs tree differencing on their ASTs and returns a list of *change operations*. A change operation is either Insert, Update, Move, or Delete. One could compare this list of change operations to the changes as produced by the `diff` tool. Applying `diff`'s changes on the first version of the program results in the second version of the program. Similarly, applying ChangeNodes's list of operations on the AST of the first version of the program, results in the AST of the second version of the program. ChangeNodes thus provides fine-grained changes describing the `diff` style changes in the commit. In our scenario the two versions used as input for ChangeNodes are (1) the version of the program with the changes of the commit under analysis not yet applied and (2) the version of the program after the commit under analysis has been applied.

Listing 4.2 depicts the output of ChangeNodes when applied to our running example. ChangeNodes computes nine AST-level change operations that have

```
1 Update: _bigPort SimpleProperty[org.eclipse.jdt.core.dom.SimpleName,identifier]
2 Update: _bigPort SimpleProperty[org.eclipse.jdt.core.dom.SimpleName,identifier]
3 Update: _bigPort SimpleProperty[org.eclipse.jdt.core.dom.SimpleName,identifier]
4 Update: _bigPort SimpleProperty[org.eclipse.jdt.core.dom.SimpleName,identifier]
5 Update: _bigPort SimpleProperty[org.eclipse.jdt.core.dom.SimpleName,identifier]
6 Update: _cover SimpleProperty[org.eclipse.jdt.core.dom.SimpleName,identifier]
7 Update: _cover SimpleProperty[org.eclipse.jdt.core.dom.SimpleName,identifier]
8 Update: _cover SimpleProperty[org.eclipse.jdt.core.dom.SimpleName,identifier]
9 Update: _cover SimpleProperty[org.eclipse.jdt.core.dom.SimpleName,identifier]
```

Listing 4.2: The distilled changes for commit 5e2cdc06 of ArgoUML, the running example of Section 4.3. Each line is one fine-grained change. Note that the representation here is simplified for viewing. The actual distilled changes also indicate any new content as well as where in the abstract syntax tree they are supposed to be inserted, updated, moved, or deleted.

the same effect as the original commit. In our example, they are all of the Update type: the names of the variables are updated. We number the distilled changes from one to nine for future reference.

4.3.2 System Dependence Graph

In the second step, our technique uses program dependence graphs (PDGs). Program dependence graphs contain both control and data flow dependencies as dependence edges (see Section 2.4.1). In this chapter, we also make the following explicit distinction in terminology. A *procedure dependence graph* is the program dependence graph for one method or procedure. Method calls are not resolved. A *system dependence graph* (SDG) is the program dependence graph for a combination of procedure dependence graphs. The method calls are used to link different procedure dependence graphs. Our SDGs are for the entire file in which changes occur.

For the implementation of this second step, we extend the open source TinyPDG tool [68, 69, 70]. TinyPDG creates a procedure dependence graph of a Java method from an AST provided by the Eclipse Java Development Tools (JDT). This was a convincing point in its selection. It enables our implementation to link results from TinyPDG back to ChangeNodes, as both operate on the same AST. TinyPDG does, however, only work on an intra-procedural level and thus creates only procedure dependence graphs. Our implementation therefore renders TinyPDG inter-procedural using the algorithm introduced by Horwitz et al. [75]. Using this algorithm our extended version of TinyPDG, TinySDG, combines the procedure dependence graphs of the different methods into one SDG. Note that our implementation does this on a per file basis, not for the entire project. In other

```
1 A a = new A();
2 a.setX(1);
3 A b = a;
```

Listing 4.3: An example of a partial definition of an object. TinyPDG does not realise line three should have a data dependency on line two.

words, the resulting SDGs connect procedure dependence graphs from within the same file, not from outside that file.

There are some shortcomings in the creation of our SDGs, we describe four. First, consider a situation where a `b++` is used in the right-hand side of an assignment, e.g., `a = b++;`. In this case, TinyPDG does not see this as an assignment to `b`. Thus, TinyPDG does not add the required data dependencies for statements using `b` later in the program. Second, TinyPDG cannot handle `try-catch` statements. Galindo et al. [52] propose exception-sensitive program slicing to handle these statements correctly. Third, TinyPDG does not handle partial object definitions correctly, an example of which is shown in Listing 4.3. In this example, class `A` has a property `x` for which there is a setter function `setX`. In this example, TinyPDG will not create a data dependency between line three and line two. Similar issues arise with constructors. Finally, the SDG algorithm by Horwitz et al. [75] does not consider polymorphism. These issues in turn make the slicing in the next step of our approach (Section 4.3.3) incomplete, i.e., the slices end up containing less code than what is needed. Some of these limitations can be mitigated by use of an expression dependence graph [132, Chapter 5], which uses more fine-grained nodes in the graph and rethinks the edges between the nodes, as well as by using object flow dependencies [53]. Further improvements could also be made in terms of dealing with data structures, by using field-sensitive techniques [50], or in terms of dealing with concurrency [51], which our approach does not consider.

The entire system dependence graph created for the running example is too large to include here as there are other changes in the file that we did not include in Listing 4.1. Instead we show an extract of the relevant parts in Figure 4.2.

4.3.3 Program Slicing

In step three, our technique performs program slicing [169, 170]. We briefly discussed program slicing with an example in Section 2.4.2. The idea behind program slicing is as follows. Given a variable of interest v in a statement s, backwards program slicing on v retrieves the statements that may affect that v in that location. Executing a program reduced to those statements that affect variable v should, in theory, compute the same run-time values for v as if the entire program were executed. We opt for backwards slicing over other slicing

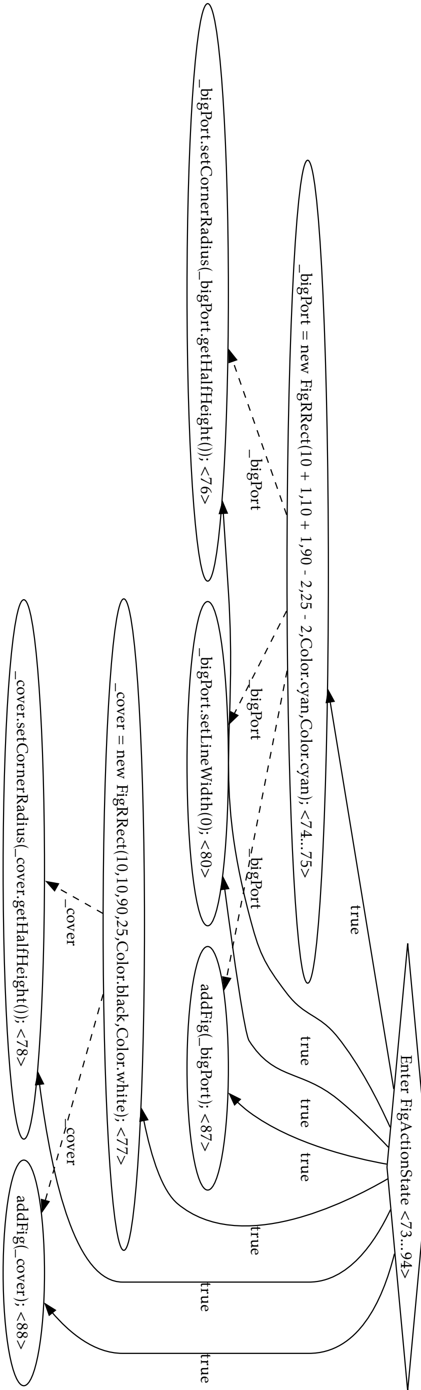

Figure 4.2: Relevant parts of the system dependence graph for the running example of Section 4.3. Solid lines with the label `true` are control dependencies. Dashed lines are data dependencies. The label indicates the identifier set by the source node and used by the destination node.

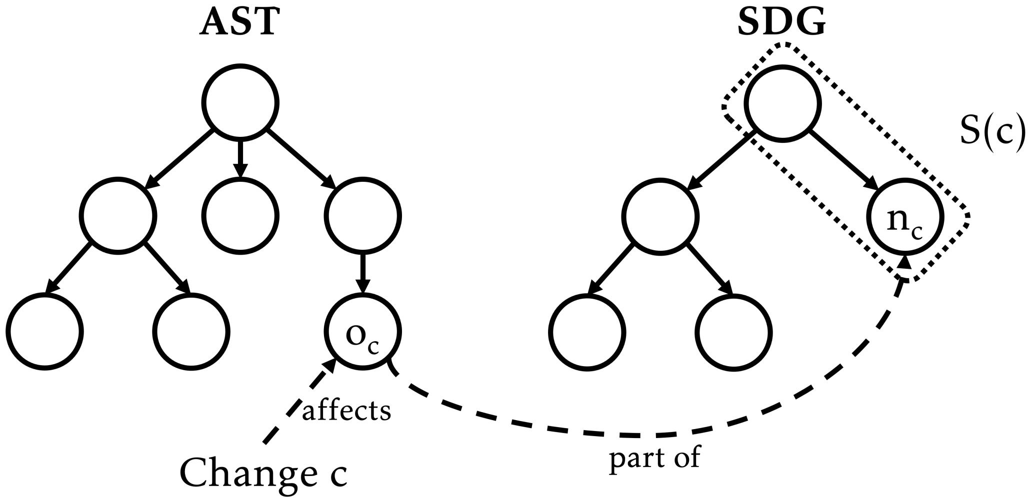

Figure 4.3: Visual explanation of how slicing around a fine-grained change operation c works. The abstract syntax tree node affected by change operation c is marked as o_c. o_c corresponds to or is part of a node n_c in the system dependence graph. $S(c)$ is the resulting slice backwards from n_c.

strategies due to backwards slicing resulting in a small slice, i.e., only what leads directly to a certain program point is included. This minimises which other parts are potentially considered related and also leads to smaller graphs to work with, which should be more efficient.

A common static approach to backwards slicing relies on a program dependence graph. Since the program dependence graph establishes the dependencies for all parts of a program, the information to slice is already present. Specifically, backwards slicing on a node n in a program dependence graph amounts to determining what other nodes n can be reached from.

Our extension to TinyPDG implements the backwards slicing algorithm for system dependence graphs introduced previously by Horwitz et al. [75]. To determine what node to slice on, our technique once more considers the fine-grained change operations provided by ChangeNodes. Figure 4.3 depicts how slicing around a fine-grained change operation c works. For such a fine-grained change operation c, we identify the original location o_c of the abstract syntax tree node affected by the change. This location is used to find the node n_c in the system dependence graph such that o_c is present in n_c. Slicing on node n_c produces the slice $S(c)$. For ease of notation we will just write $c_i \in S(c_j)$ to indicate the situation in which node n_{c_i} belongs to the slice around node n_{c_j}. Such a slice is computed for every fine-grained change operation.

We note that in program slicing, a variable is specified to indicate which data dependencies to follow and which to ignore. In our slicing, we forego this variable and consider the entire node in the SDG. Data dependencies are followed indiscriminately. In certain situations, this will lead to too large slices that include parts of the SDG that are not relevant. Solving this could be done by determining which exact variables are affected by a change to the AST and providing that information to the slicer. Alternatively, an expression dependence graph [132], which is more fine-grained than the SDG we use, can potentially mitigate this issue as well.

For the running example, our technique links the change operations described in Listing 4.2 to the nodes of the system dependence graph in Figure 4.2. We refer to the nodes in the graph by the number between < and > present in the node. Change operations c_1 through c_5 are linked to SDG nodes $74 \dots 75$, 76, 76, 80, and 87, respectively. This leads to slices

$$
\begin{aligned}
S(c_1) &= \{73 \dots 94, 74 \dots 75\} \\
S(c_2) &= \{73 \dots 94, 74 \dots 75, 76\} \\
S(c_3) &= \{73 \dots 94, 74 \dots 75, 76\} \\
S(c_4) &= \{73 \dots 94, 74 \dots 75, 80\} \\
S(c_5) &= \{73 \dots 94, 74 \dots 75, 87\}
\end{aligned}
$$

The calculations for change operations c_6 through c_9 happen analogously.

4.3.4 Change Grouping

Finally, our technique needs to decide which change operations belong together. To do so, we define the equivalence relation $\equiv_S$ between change operations. Using the equivalence relation then enables partitioning the set of change operations into disjunct sets.

To define $\equiv_S$, we first define the helper relation $\equiv'_S$.

$$c_i \equiv'_S c_j \iff c_i \in S(c_j) \vee c_j \in S(c_i)$$

In other words, change operations c_i and c_j are related by $\equiv'_S$ if and only if c_i is in the backwards slice on c_j or vice-versa. Note that the relation $\equiv'_S$ is *not* an equivalence relation. Recall that an equivalence relation is a relation that is reflexive, symmetric, and transitive. By definition of how slicing works, this relation is reflexive. The relation is also trivially symmetric due to its symmetric definition. The relation is however not transitive. We cannot state that if $c_i \equiv'_S c_j$ and $c_j \equiv'_S c_k$, then $c_i \equiv'_S c_k$. Consider for this the following simplified situation. c_j is part of the root node of a program dependence graph with two successors. c_i is part of

one of the successor nodes. c_k is part of the other successor node. Slicing in this situation results in

$$S(c_i) = \left\{c_i, c_j\right\}$$
$$S(c_j) = \left\{c_j\right\}$$
$$S(c_k) = \left\{c_j, c_k\right\}$$

Then $c_i \equiv'_S c_j$ and $c_j \equiv'_S c_k$, but $\neg(c_i \equiv'_S c_k)$. The relation $\equiv'_S$ is thus not an equivalence relation.

We use $\equiv'_S$ to define $\equiv_S$. Specifically, $\equiv_S$ is the transitive closure of $\equiv'_S$:

$$c_i \equiv_S c_j \iff \exists c_{k_1}, \ldots, c_{k_n} : c_i \equiv'_S c_{k_1}, \ldots, c_{k_n} \equiv'_S c_j$$

In other words, there is a relation $\equiv_S$ between two change operations c_i and c_j if and only if there is a chain of change operations that links c_i to c_j. Each change operation in the chain is connected to the next change operation by means of the relation $\equiv'_S$.

Algorithmically, our technique uses $\equiv'_S$ to build the partition for $\equiv_S$. The following steps are followed, given the change operations, their slices, and an empty set to hold the partition.

1. If a change operation is not in relation $\equiv'_S$ with any change operation in any of the existing subsets in the partition, create a new subset with that change operation in it.

2. If a change operation is in a relation with an element (or more elements) of exactly one existing subset in the partition, place the change operation in that subset.

3. If a change operation is in a relation with two (or more) elements of different subsets, join the subsets together and add the change operation to it.

In Section 4.1, we hypothesised that related changes affect source code from the same program slice and thus that a commit may be decomposed into related changes using the created program slices. We rephrase our hypothesis in terms of the partition, i.e., in terms of these subsets of change operations: *a commit is a single-task commit if and only if our technique does not split up the commit into different subsets of change operations*.

Applying this relation to the running example, we see that

$$c_1 \in S(c_2) \land c_1 \in S(c_3) \land c_1 \in S(c_4) \land c_1 \in S(c_5)$$

and thus

$$c_1 \equiv'_S c_2 \land c_1 \equiv'_S c_3 \land c_1 \equiv'_S c_4 \land c_1 \equiv'_S c_5$$

```
1 5e2cdc061a0572d4007f4bc84382fff80f29e726,
    src_new_org_argouml_uml_diagram_activity_ui_FigActionState_java,9,0,0,0,
    false,0,9,2
```

Listing 4.4: Data produced by our prototype tool when applied to the running example from Listing 4.1. One line is produced per file analysed in a commit.

giving finally

$$c_1 \equiv_S c_2 \equiv_S c_3 \equiv_S c_4 \equiv_S c_5.$$

Similarly, we can see that

$$c_6 \equiv_S c_7 \equiv_S c_8 \equiv_S c_9.$$

There is no further connection between change operations, so there are two equivalence classes. Our algorithm considers the running example to consist of two distinct parts: the renaming of `bigPort` and the renaming of `cover`.

4.3.5 Prototype Output

For completeness sake, we describe the output of our tool when applied to the running example in Listing 4.1 that was introduced at the start of this section. In Listing 4.4, we show the output that would be processed by other tools. This output is a CSV dump containing one line per file analysed in a commit. Here, there is only one line. The line lists the commit that was analysed followed by the file that was analysed. Next comes the total number of change operations found, here this is nine. The three next numbers indicate change operations our prototype could not properly analyse, namely those modifying a comment, those producing a certain internal error, and those not within a method. In the running example, there are none of these change operations. Next is a boolean indicating whether something went wrong during analysis, e.g., an error being thrown when trying to create the system dependence graph. In this example, nothing went wrong. The next number indicates for how many change operations there was no corresponding node in the system dependence graph of the program before the commit, again zero in this case. The next number is there as a sanity check and indicates how many change operations are getting clustered. Adding this number up with the previous four numbers, which indicated a change operation could not be handled by the prototype, should result in the very first number. This is correct here: $9 + 0 + 0 + 0 + 0 = 9$. The final number indicates how many clusters were created. In other words, a number higher than one indicates the commit was composite.

We also show the more user-friendly output of our tool in Figure 4.4. This is a diff view where the leftmost column has numbers indicating which cluster of

```
 2     2          public FigActionState() {
#0 3        -             _bigPort = new FigRRect(10 + 1, 10 + 1, 90 - 2, 25 - 2, Color.cyan, Color.cyan);
#0 4        -             _bigPort.setCornerRadius(_bigPort.getHalfHeight());
#1 5        -             _cover = new FigRRect(10, 10, 90, 25, Color.black, Color.white);
#1 6        -             _cover.setCornerRadius(_cover.getHalfHeight());
#0 7        -             _bigPort.setLineWidth(0);
#0 8        -             addFig(_bigPort);
#1 9        -             addFig(_cover);
       3    +             bigPort = new FigRRect(10 + 1, 10 + 1, 90 - 2, 25 - 2, Color.cyan, Color.cyan);
       4    +             bigPort.setCornerRadius(bigPort.getHalfHeight());
       5    +             cover = new FigRRect(10, 10, 90, 25, Color.black, Color.white);
       6    +             cover.setCornerRadius(cover.getHalfHeight());
       7    +             bigPort.setLineWidth(0);
       8    +             addFig(bigPort);
       9    +             addFig(cover);
10    10          }
```

Figure 4.4: Diff view produced by our prototype tool when applied to the running example from Listing 4.1. Key here is the leftmost column, which indicates the clusters present.

changes each line belongs to. Here we can clearly see the two clusters `#0` and `#1`. In the interface, the clusters are coloured differently for easier visual scanning.

4.4 Dataset

We now introduce the well-established dataset of commits that we will use to evaluate our hypothesis. In Section 4.5, we will evaluate our approach on a refinement of this dataset, to which we apply data cleansing through automated filtering and manual commit verification first.

The dataset of commits stems from five Java programs as used by Herzig and Zeller in [66, 67]. We are not aware of version numbers assigned to this dataset. The programs in question are: ArgoUML, GWT, Jaxen, JRuby, and XStream. These projects were chosen by Herzig and Zeller for meeting the following quality criteria: to be under active development (at the time of their analysis), to have at least 48 months of active history, to have more than ten active developers, and to feature a reasonable number of identifiable bug fixes. For each of the projects, Herzig and Zeller manually identified single-task and composite commits using commit and issue information. Using the single-task commits, Herzig and Zeller also created artificial composite commits for each of the five projects. In order to see how our approach deals with real-world situations, we do not consider this set of artificial composite commits for our evaluation. Instead, we limit ourselves to the real-world commits in the dataset. Table 4.1 depicts the number of commits present for each type of commit in each of the projects. Table 4.1 also provides the median number of Java files found per commit.

The prototype implementation of our technique is limited in the types of files it supports. We use this information to perform an automated filtering of the

		Original		After automatic filtering			After manual filtering and verification		
		Commits	Java files per commit	Commits	Java files per commit	Change operations per file	Commits	Java files per commit	Change operations per file
ArgoUML	single-task	125	1	75	1	5	76	1	5
	composite	168	2	112	2	4	63	2	6
GWT	single-task	44	1	19	1	7.5	21	1	6.5
	composite	68	3.5	46	2	7.5	30	2	7.5
Jaxen	single-task	32	1	22	1	5	16	1	5.5
	composite	12	1	6	1	5	3	1	9
JRuby	single-task	200	1	60	1	4	60	1	4
	composite	271	1	112	1	5	81	1	6
XStream	single-task	37	2	26	1	4	21	1	6
	composite	37	3	26	2	7.5	17	2	9
Total	single-task	438		202			194		
	composite	556		302			194		

Table 4.1: Descriptive statistics for the original dataset by Herzig and Zeller [66, 67], the dataset after automated filtering, and the dataset after our manual commit verification. In case of aggregated numbers (i.e., those per commit or per file), the median is given.

commits in the dataset which were known to be affected by these limitations. In terms of file types, our prototype does not support non-Java files that might appear in a commit. Our prototype also fails to construct the system dependence graph for some of the Java files. We mark the corresponding commits in the dataset as causing failures. In the case of two files, moreover, no exception was thrown but graph construction timed out, i.e., it took longer than an hour.

In terms of changes to the files affected by a commit, there are some limitations too. Program or system dependence graphs do not take comments into account. It follows that our approach cannot either. Moreover, in our prototype implementation, the graph construction algorithm only works on code contained within methods. As such, changes to, for example, class or field declarations are ignored. Finally, for some change operations, our implementation failed to identify a matching node in the system dependence graph. This can be the case when something is inserted without a tie to the original code. By this we mean a situation where ChangeNodes categorises the change operation as an Insert rather than a Move or an Update of code that was already in the program. As our system dependence graph considers the original version of the code, this Insert cannot be linked to any meaningful node. Taking these limitations into account, our automated filtering step removes commits for which no files with valid change operations remain. Only the commits that remain are considered for any further evaluation. Table 4.1 depicts the number of each type of commit per project after the automated filtering step. The table also depicts the median number of Java files per commit for these remaining commits. Finally, Table 4.1 also depicts the median number of valid change operations per Java file.

Following the automated filtering described above, we perform a detailed manual verification of the 504 commits that remain from the original dataset. We inspect the code changed by each commit as well their accompanying commit message. We do not further use commits matching one of the following aspects.

- The commit consists of many changes to statements that TinyPDG cannot handle, or is centred around such changes. An example is the `try-catch` statement.[2]
- Our approach considers the project before the changes. If the commit consists entirely of new files, then our approach cannot do anything.
- One of the tasks in a composite commit is the fixing of comments, formatting, or other style issues. The presence of such a task in a composite commit means our technique would not be able to correctly distinguish tasks.

[2]While there is code for this present within TinyPDG, we saw no notice of it in the generated PDGs.

- The commit consists of many repeated changes in otherwise unrelated locations. Consider, for example, commit ba2f8bd2[3] of the JRuby project. In this commit, a `null` check is added to several different methods. While conceptually related, this is not a relation our technique can possibly discern.

In case of uncertainty in our analysis, we left the data of the dataset as is. In this manner, we avoid personally influencing the data. Finally, we also consider for every commit whether the dataset categorised the commit correctly. By this we mean the commit was marked as comprising many tasks while it was actually just a single task, or vice-versa.

Our manual analysis filters out another 116 of the 504 commits. Among the 116, 35 have formatting as one of the tasks, such as whitespace changes. In 13 occasions, TinyPDG would not be able to handle the types of changes. Repeated changes occur 53 times. Finally, 20 of the commits have a majority of new files being added. We point out that these numbers do not add up to 116 because some commits were placed in multiple categories.

We are left with 388 commits for the evaluation of our approach. Of these, 359 commits pass all our scrutiny. We find the other 29 commits to be categorised incorrectly. The main culprit for incorrect categorisations may be a different interpretation of what a composite commit is. Take, for example, commit 26d69d46[4] of the GWT project. This commit refers to two issues in its commit message: "Fix for issues #966 and #867; escapes HTML end tags from string literals in compiler output.". Despite there being two issues mentioned, there is but one fix that happens to fix the both of them. These types of commits were marked as being composite in the dataset, but we consider them to only perform a single task. As such, to perform the evaluation, we switch the categorisation for these commits.

Table 4.1 describes what is left of the dataset. With these remaining commits, we will evaluate our approach. As mentioned before, Table 4.1 depicts the number of commits, the median number of Java files per commit, and the median number of valid change operations per Java file.

Conclusion. After filtering out commits that cannot be used to evaluate our approach, the dataset contains 388 commits which will be used for the evaluation.

4.5 Evaluation

To analyse our hypothesis, we consider two research questions.

[3] The full identifier is ba2f8bd229c62aa68acf176fa5c7578a4e7670e1. As mentioned before, due to the way Herzig and Zeller [67] constructed their dataset, this hash may only have meaning within their dataset.

[4] The full identifier is 26d69d46ad7fbb01ac5a2cd9a03084f73e9cff51.

RQ1 Does our technique correctly identify composite commits?

RQ2 Does our technique correctly identify the single tasks within a commit?

We will use the dataset described in Section 4.4 to answer these research questions. The remainder of this section consists of the following three parts. We describe the research method for each research question, provide the results for each answer, and discuss any threats to validity.

4.5.1 Research Method

In Section 4.1, we hypothesised that *related changes affect source code from the same program slice* and thus that *a commit may be decomposed into related changes using the created program slices*. Our hypothesis thus concerns entire commits. The implementation of our technique, however, works on a per file basis. Its output is the number of sets in the partition of the file, i.e., the number of clustered changes for that file. Commits may contain many changed files and as such we need to reconcile the two. We define the following metrics to do this.

Definition. *The $PARTITION_{file}$ metric is defined as the number of sets in the partition of the change operations for a given file, as attained by our algorithm described in Section 4.3.*

Definition. *The $PARTITION_{commit}$ metric is the maximum of the $PARTITION_{file}$ for the files changed in the commit.*

$$PARTITION_{commit} = \max_{file \in commit} PARTITION_{file}$$

Using that, we define classification by our technique as follows.

Definition. *$COMPOSITE_{commit}$ is a boolean. When true for a certain commit, the commit in question is considered composite.*

$$COMPOSITE_{commit} = (PARTITION_{commit} > 1)$$

In other words, our technique considers a commit composite if at least one file had more than one set of change operations in the file's partition.

Composite Commit Identification

We apply our technique to all commits in the dataset. Commits are classified as either composite or single-task in the dataset. Our technique likewise classifies a commit as either composite or single-task. Thus there are four possible results to consider in the evaluation of a commit.

1. Composite commit correctly identified as composite. A true positive.
2. Single-task commit correctly identified as single-task. A true negative.
3. Single-task commit incorrectly identified as composite. A false positive.
4. Composite commit incorrectly identified as single-task. A false negative.

We will provide the number of times each of these possibilities occurred. We will also provide the precision[5] and recall[6] based on those numbers. Finally, the F-score[7], which combines precision and recall into a single number, is provided. These three metrics provide a number between 0 and 1 where 1 represents a situation in which the algorithm is entirely correct. Results will be reported on a per project basis.

Single-Task Identification

Here too our technique is applied to all commits in the dataset. The results from the previous research question remain relevant here. However, for this research question we will look at the results with a focus on the single-task commits, rather than on the composite commits. If our technique is good at identifying single-task commits, then that is a strong indication it is good at identifying single tasks. However, it is not sufficient: what if the technique just overapproximates single tasks?

To avoid this possibility, we will also look into the number of sets reported by our technique for composite commits (the $\text{PARTITION}_{\text{commit}}$ metric mentioned before). The dataset states for some composite commits how many tasks they comprise. We will compare the numbers reported by our technique to the numbers present in the dataset. If the numbers match up, this is an indication that the sets in the partition identify single tasks correctly within a composite commit.

Note that this would just be an indication of correct partitioning, not a certainty. To be a certainty, we would need to compare the partitioning as performed by our approach to the partitioning in the dataset. We cannot make this comparison here as the dataset lacks the data to compare to. That is, the dataset does not specify which groups of fine-grained changes a composite commit is comprised of.

To further mitigate this issue, we have three computer scientists evaluate the output produced by our tool. At the time of the experiment, two of them had a master's degree in computer science and the third one had a PhD in the same

[5]Commonly defined as precision = $TP/(TP + FP)$.
[6]Commonly defined as recall = $TP/(TP + FN)$.
[7]Commonly defined as F-score = $2 * \text{precision} * \text{recall}/(\text{precision} + \text{recall})$, a harmonic mean between the precision and the recall.

field. None were authors of the experiment or the paper it produced, but all three worked in the same department as the authors. For a random selection of 31 commits, the computer scientists are tasked to rate the output of our tool on a five-level Likert scale: a rating from 1 to 5, a 3 is the middle, anything higher is a positive response. We will provide the median and mode of their replies per person. They are also asked whether the clustered change operations should be further combined, further split up, or neither of the two. To evaluate their answers, we will consider the number of commits for which they thought the clustered change operations should go one way or the other. Even if only one of the reviewers wants to see the clustering done differently, we will still count that commit as needing improvement. This final question is important in knowing whether our clusters span across task boundaries or not.

4.5.2 Results

We present the results and a conclusion for research questions one and two.

Composite Commit Identification

Here we answer RQ1: Does our technique correctly identify composite commits?

An overview of the results when identifying composite commits is given in Table 4.2. As mentioned, this table depicts the total number of commits, the number of true positives, the number of true negatives, the number of false positives, and the number of false negatives. It also gives the calculated precision, recall, and F-score based on those results. All the numbers are provided per project.

In interpreting these results, it is important to keep the number of commits per commit type in mind for each project. This data was provided in Section 4.4, specifically in Table 4.1. All else being equal, a higher or lower number of composite commits versus single-task commits results in higher or lower precision, respectively. If the number of single-task commits in the dataset increases, then the number of false positives increases,[8] which in turn increases the denominator when calculating the precision, thus lowering precision. Similarly a decrease in single-task commits will increase precision. Recall is not affected by this. This is also not a problem when looking at the total numbers across all projects as our refined dataset contains an equal number of composite and single-task commits, as shown in Table 4.1.

In the results, the numbers for the Jaxen project are noticeably lower than for the other results. There were few total commits for this project *and* the majority of those commits were single-task. Only three composite commits were analysed for

[8] We assume here that the ratio of single-task commits identified as single-task commits compared to single-task commits identified as composite commits does not change.

	Commits	True positive	True negative	False positive	False negative	Precision	Recall	F-score
ArgoUML	139	45 32%	51 37%	25 18%	18 13%	0.64	0.71	0.68
GWT	51	18 35%	12 24%	9 18%	12 24%	0.66	0.6	0.63
Jaxen	19	2 11%	13 68%	3 16%	1 5%	0.4	0.67	0.5
JRuby	141	48 34%	46 33%	14 10%	33 23%	0.77	0.59	0.67
XStream	38	12 32%	17 45%	4 11%	5 13%	0.75	0.71	0.73
Total	388	125 32%	139 36%	55 14%	69 18%	0.69	0.64	0.67

Table 4.2: Identification of composite commits on a per project basis. The precision, recall, and F-score in the "Total" row are calculated using the totals of the columns to their left, not a combination of the per project precision, recall, and F-score.

	Precision	Recall	F-score
ArgoUML	0.74	0.67	0.7
GWT	0.5	0.57	0.53
Jaxen	0.93	0.81	0.87
JRuby	0.58	0.77	0.66
XStream	0.77	0.81	0.79
Total	0.67	0.72	0.69

Table 4.3: The precision, recall, and F-score for the identification of single-task commits. These are calculated using the absolute numbers present in Table 4.2. The total row is calculated using the totals across the projects, not by a combination of the numbers in this table.

the Jaxen project. We are thus inclined to attribute this result, at least partially, to these factors. For the other projects, the numbers are more positive. Both precision and recall seem to be around 70%. The F-score too is around that ratio. Here we note that for the GWT and JRuby projects, more composite than single task-commits are present. This affects the precision in a positive manner.

The results are positive, but not overwhelmingly so. The prototype implementation of our technique correctly identifies a large number of commits, but leaves room for improvement.

Conclusion. Our technique correctly identifies composite commits. The F-score of the identification is 67% across all five projects.

Single-Task Identification

Here we answer RQ2: Does our technique correctly identify the single tasks within a commit?

We consider again Table 4.2, but now from the point of view of the single-task commits. To do this, we calculate the relevant precision, recall, and F-scores. The results are depicted in Table 4.3. Besides the GWT project, all resulting F-scores are over 65%. For Jaxen and XStream the results are even more positive. As in the previous section, while the result is positive, it is not overwhelmingly so. We consider the other test results.

The next step is the comparison of the number of tasks our technique identifies for a commit against the number of tasks in that commit according to the dataset. While we perform a validation of our dataset in Section 4.4, we note that this validation does not include validating the number of different tasks attributed to a commit by the original dataset ours is derived from. Table 4.4 summarises

	Composite commits	Correct number of sets
ArgoUML	33	7 (21%)
GWT	28	7 (25%)
Jaxen	0	—
JRuby	65	15 (23%)
XStream	16	4 (25%)
Total	142	33 (23%)

Table 4.4: Identifying the number of single tasks within a composite commit. Only commits for which the number of partitions is present in the dataset are considered.

the results of comparing our technique to the numbers present in the dataset. The dataset does not provide any information regarding the number of tasks in a commit for any of the commits in the Jaxen project. As such, no results are present for the Jaxen project.

The results we encounter here are negative. Identifying the exact number of single-tasks in a composite commit proves difficult for our technique. The dataset's number of tasks making up the commit was found in only a quarter of the cases for the four projects for which data was available.

The results of our survey are more positive. The raw results are visualised in Figure 4.5. As mentioned before, the untangling of 31 commits had to be rated using a five-level Likert scale: a rating from 1 to 5, a 3 is the middle, anything higher is a positive response. The median result for each of the reviewers was 5, 5, and 3. The mode was 5, 5, and 3 with 20 (65%), 18 (58%), and 18 (58%) occurrences, respectively. Two of the reviewers were rather positive about the output, one remained more neutral. In terms of ways to improve the clustered changes, 3 of the 31 cases (10%) were considered to need further splitting up of the created clusters of changes. For 10 (32%) of them the opposite was true: more changes should be combined into one cluster. No changes should be made in the other 18 cases (58%).

The earlier negative result may be explained by our slicing being incomplete (as we described in Section 4.3.2). This incompleteness means the created slices are too small, which causes less clustering to happen. Indeed, the survey result corroborates this and, to some extent, mitigates the earlier negative result. The reviewers believe changes need to be combined further than what our approach currently does, implying a created cluster of changes does not span across multiple tasks in a commit. If the clusters would instead span across tasks, they would no longer be helpful for a stakeholder using our approach. Our clusters of changes

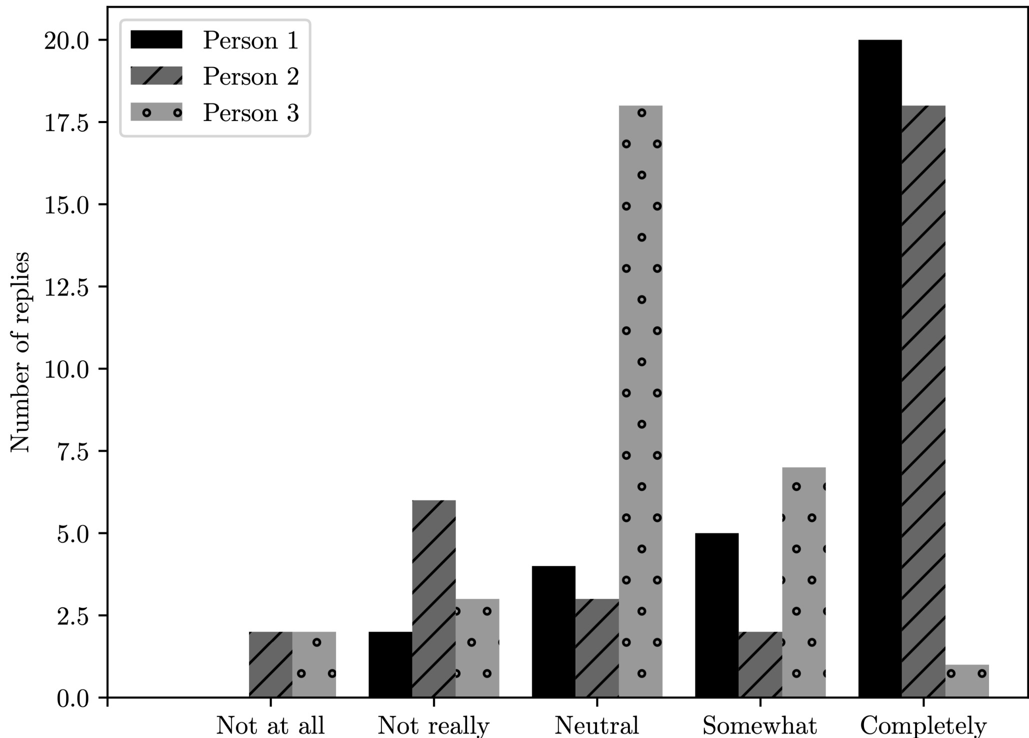

Figure 4.5: Results of a three-person survey. Each person analyses 31 commits and the result of our change clustering. They need to assess whether the change clustering makes sense.

are more fine-grained than reality. The individual clusters stay within their tasks and each manages to identify a part of their task. This ensures the clusters are not rendered useless when analysing or reviewing a commit. A stakeholder may still use the different clusters of changes knowing each cluster contains changes that belong together. The stakeholder can then still further combine the clusters as they deem necessary.

Conclusion. Our approach is able to identify when a commit performs a single task with an F-score approaching 70%. Our approach creates more fine-grained parts than those that are counted in the dataset. A manual review by three computer scientists finds that the clusters of changes are contained within their respective tasks in a commit. As such, the clusters can still be used for code review, integration, and reversion.

		25%	50%	75%	mean	std
ArgoUML	single-task	0	1	4	66	316
	composite	3	11	43.5	199	1079
GWT	single-task	1	2	10	26	72
	composite	3	7	56	270	819
Jaxen	single-task	0	1	2	1	1
	composite	8.5	17	23.5	16	15
JRuby	single-task	1	2	6.75	10	31
	composite	5	19	85	150	617
XStream	single-task	0	1	1	1	1
	composite	1	4	17	10	11
Total		1	3	19	101	585

Table 4.5: Time in seconds taken to analyse a commit with our untangling approach. A zero indicates analysis took under one second. The 25%, 50%, and 75% percentiles are listed. Mean and standard deviation are rounded to the nearest integer.

Time Efficiency

Finally, we have a look into the scalability of our approach. To do so, we consider the time it takes to analyse a commit. These experiments were run on a Macbook Pro from 2013. The times for analysis per commit is depicted in Table 4.5, grouped by project and by commit type (single-task versus composite). Note that time was recorded with a precision of seconds, thus entries stating 0 imply a time under 1 second. The table shows the 25%, 50%, and 75% percentiles as well as the mean and standard deviation. For the majority of commits, the analysis takes but a few seconds. There were outliers, however, with 54 commits (14% of all commits) taking a minute or more and 12 (3%) of those taking over ten minutes.

We list these 12 commits in Table 4.6. We have a deeper look at each of the commits and conclude that in all cases the longer time spent analysing is due to one file. This file is listed in Table 4.6 as “Slowest File”. The time taken on just that file is also listed, as well as the percentage compared to the total time for the commit. Finally, the last column lists the time taken to create the SDG of that file and obtain the list of fine-grained changes made to the file. The percentage indicates that number compared to the total time for that file. The rest of the time attributed to the file consists of the analysis looping through the change operations: creating the slice for each and grouping change operations together using $\equiv_S$. For 11 of the 12 slow files, the slow analysis is to be blamed on this latter step. For commit 87da6cb of the GWT project, however, almost all the

Project	Category	SHA	Time	Slowest File	Time in File	Time Pre-Slice
ArgoUML	composite	66391e6	8532	ParserDisplay.java	8364 (98%)	76 (1%)
JRuby	composite	1dbbe1a	5316	Java.java	4820 (91%)	11 (0%)
GWT	composite	768d390	3702	TypeOracleBuilder.java	3699 (100%)	14 (0%)
GWT	composite	092c418	2554	SerializableTypeOracleBuilder.java	2544 (100%)	10 (0%)
ArgoUML	single-task	3931bf8	2342	Modeller.java	2342 (100%)	5 (0%)
JRuby	composite	327a5a6	1284	Sprintf.java	1125 (88%)	125 (11%)
ArgoUML	single-task	b6c2b2f	1231	ParserDisplay.java	1231 (100%)	45 (4%)
JRuby	composite	510a41b	1187	RubyZLib.java	1170 (99%)	6 (1%)
ArgoUML	composite	9a08725	1096	Facade.java	1095 (100%)	3 (0%)
ArgoUML	single-task	76dda80	879	Facade.java	875 (100%)	2 (0%)
GWT	composite	87da6cb	644	DeveloperGuide.java	435 (68%)	433 (99%)
JRuby	composite	7d9d9f3	641	Java.java	621 (97%)	7 (1%)

Table 4.6: The twelve commits for which the analysis took over ten minutes. All times are listed in seconds. For every commit, the file that took the longest to analyse is listed as “Slowest File”. The time taken for this file is listed in the second-to-last column. The percentage compared to the total time for the commit is listed in parentheses. The time taken to create the SDG and obtain a list of fine-grained changes for that file is listed in the last column. After that point, the analysis loops over the fine-grained changes to slice and link them together. In parentheses is the percentage of that time compared to the total time for the file.

time is spent on creating the SDG and obtaining the change operations. The file in question seems to consist primarily of comments and several nested classes without a body. We hypothesise this esoteric case trips up the SDG creation or the change distiller.

For the 11 slow files where slicing and clustering the fine-grained change operations takes up most of the time, there are always a high number of change operations. A cursory glance at commits that did not take especially long to analyse, however, seems to indicate this does not on its own explain when analysis for a file will be slow. At the time of writing, we lack the structured data to make more conclusive statements, but hypothesise a combination of a large number of change operations and bigger files causes the slicing and clustering to be too inefficient.

4.5.3 Threats to Validity

The evaluation of our approach depends on the correctness of the dataset we use as a ground truth. This is true in its most basic form: stating whether a commit is composite or not. This is also true in the number of single tasks it discerns in a composite commit. As described in Section 4.4, we try to mitigate this to some extent by performing a manual filtering phase. We were conservative in this manual filter phase, so errors may still be present. Mistakes in either could cause an over or under approximation in our evaluation. We mitigated this to some extent by means of our human survey.

Our implementation may have bugs. This in turn affects the results of analysing commits and the evaluation of those results. Also in our implementation, we enable binding resolution, as provided by the Eclipse JDT library, to create the abstract syntax tree. We enable binding resolution in order to resolve method calls and the like to their definition. In this, we are bound by the precision of this static binding resolution. This can further influence the results.

In our implementation, the binding resolution only happens within one file, not across files. In other words, a method call to a method in another file does not get resolved. This means that related changes across different files cannot get linked together when they should be.

TinyPDG is not able to handle some Java statements, like `try-catch`. Our manual filter phase only removes commits where these statements were the *main* part of what was being changed. This results in a possibly subpar analysis. Horwitz et al.'s algorithm [75] is not entirely state of the art. Improvements have been made over the years to enable a better handling of, for example, classes and objects. Not using the state of the art may negatively affect our results. Using more recent work [50, 51, 52, 53, 132] would result in more accurate slices, which in turn affects our change clustering.

In our survey, we did not provide strict guidelines to the participants on how

to grade clusters. Deciding on a number that encompasses all opinions a participant may have on a cluster is not easy. Thus those results may be skewed by the interpretation of the participants.

It is possible unrelated tasks touch overlapping parts of the code in the same way. When committed together, our technique is not able to distinguish the two. We do not see a way around this situation with just our approach.

4.6 Conclusion

We want to help developers, code reviewers, and researchers with tool support for decomposing composite commits according to the tasks they perform. For the foundation of this tool support, we start from the hypothesis that related changes belong to one and the same program slice in a program dependence graph. The corresponding algorithm performs program slicing on the change operations computed for a commit by a change distiller, and clusters the resulting fine-grained change operations according to the slices they belong to. We evaluated our technique on a well-established dataset of commits stemming from five Java projects [67]. We first analysed this dataset and further refined it to fit our context. We found that our technique is able to identify single-task and composite commits. We also found that our technique creates more fine-grained clusters than those counted by the dataset we used. A manual review indicated that in the situations where there is no one-to-one mapping from cluster to task, each cluster of changes still stays within one single task. We conclude that our approach is capable of alerting developers about commits that are composite, prompting action on their part. It also enables identifying the individual tasks of said commit. This way, the commit can be corrected before being pushed to other members of the team or the public at large.

4.7 Future Work

In this chapter we sliced on the program dependence graph of the version of the code before the commit was applied. For larger additions, it may be more interesting to slice in the program dependence graph for the new version of the code. This is because a large addition does not have a clear node in the original AST or SDG to be linked to. A multi-version graph approach such as used by Pârtachi et al. [127] and by Chen et al. [30] may prove beneficial here.

In future work, we can consider different clustering criteria. For example, employing a relation similar to the (c) connection we discussed when describing related work by Barnett et al. [11], which links diff-regions if they used the same method, could be used after our technique has partitioned the changes to combine some of the newly formed groups. Another consideration could be to vary

the importance of data and control dependencies when slicing, e.g., by not following certain control dependencies.

Slicing can be done in two directions. We performed backwards slicing, considering the statements that affect a certain statement. One could also slice forwards, considering the statements affected by a certain statement. Forwards slicing might, however, match larger areas of the program dependence graph, which might influence precision. Alternatively, a best-of-both-worlds approach might combine the two directions to perform some sort of "optimal" slicing. This distinction matters since, as we discussed in Section 4.3.2, TinyPDG does not create a complete slice when dealing with, for example, partial object definitions.

Applying advancements [50, 51, 53, 132] in program dependence graph creation and associated slicing would result in more accurate slices. This in turn would, presumably positively, affect the clustering as performed by our approach.

Our approach is currently applied within files. The alternative of working across files, e.g., for the entire project or on a per package basis, should be investigated. Specifically, since this creates larger graphs and longer lists of fine-grained changes per graph, the trade-offs in time efficiency should be weighed against the gains in accuracy.

What is or is not a composite commit can depend on the situation and on the stakeholder looking at a commit. Our approach uses code dependencies, which may not be applicable in all situations. It would be interesting to explore these situations and experiment with a combination of our approach with related work to perhaps tackle such situations too. Alternatively, one could survey developers on how they would cluster changes and derive what kind of dependencies they would like to see used.

For an industrial setting, our technique needs to be able to analyse a commit reasonably fast. Our analysis only took a few seconds for the majority of analysed commits. However, several commits took over a minute to analyse. Optimisations to our research prototype are likely to be in order.

Chapter 5

Prevalence of Merge Conflicts

In Chapter 3, we introduced the two types of problematic commits that we analyse in this dissertation. In Section 3.1, we discussed composite commits, their effects, and their prevalence. In Section 3.2, we provided definitions of various merge conflicts that can occur in merge commits. The three main categories we described were: (1) textual, where an overlap in changed lines of code leads to the version control software passing the conflict on to the user; (2) syntactic, where the code seems to merge correctly, but the result is not syntactically correct; and (3) semantic, where the conflict appears due to the interaction of behaviour leading to a bug that is not caught until the tests, or worse, a user, trigger it. In Section 3.3, we focused specifically on semantic merge conflicts, but withheld focusing on their prevalence.

In this chapter, we describe the results of our study into the prevalence of syntactic and semantic merge conflicts on a large scale. Specifically, we analyse their prevalence within the context of continuous integration. Continuous integration is an industry practice aiming to detect various run-time errors through an extensive pipeline of successive tests.

We combine data from Travis CI [162], one such continuous integration service, with data from GitHub [56], a host for Git repositories. By combining this data, we can not only determine when a merge conflict occurs, but also walk further up and down the history of the project to determine when the issue got fixed.

We find merge commits lead to failure less often than regular commits. Repairing the code is usually done the same day and takes fewer than ten lines of code. Repairing tends to happen in the source code as opposed to the test code. These results indicate that applying proper practices, such as having a good test suite, mitigates many issues associated with code integration. Writing and maintaining a good test suite requires a lot of development time and effort, which is why we look into an automated way to detect semantic merge conflicts in Chapter 6.

In Section 5.1, we introduce three research questions to analyse the prevalence of merge conflicts. In Section 5.2, we discuss related work. In Section 5.3, we describe the gathering and refining of our dataset. In Sections 5.4 and 5.5, we answer the research questions.

5.1 Research Questions

To study the prevalence of syntactic and semantic merge conflicts in a structured manner, we seek to answer the following three research questions:

RQ1 How often do code integrations lead to syntactic and semantic conflicts?

RQ2 How much effort is needed to fix conflicts after code integration?

RQ3 What type of files is this effort concentrated in?

5.2 Related Work

We discuss related work that looks into the prevalence of merge conflicts. When we performed our study, the existing related work did not look at prevalence of merge conflicts at scale, instead analysing a handful of projects at a time. Due to the novelty of this study at the time, we include the study as a standalone chapter in this dissertation and discuss its related work here rather than in a section in Chapter 3 as we did when describing the prevalence of composite commits.

Brun et al. [23, 24] analysed 3,562 merge commits across nine open source projects. Their study observed that about one in six merge commits leads to a textual conflict. Three of the nine open source projects were investigated for build and test conflicts after a merge had been committed. A build conflicts implies building the projects fails. A test conflict implies one or more of the tests reported an error. In Section 3.2, we described both in more detail and discussed why we avoid the terminology in our own work. Build conflicts were found in 0.1%, 4%, and 10% of merge commits. Test conflicts were found in 4%, 28%, and 3% of merge commits. The study lacks in two aspects. First, the sample size is small. Only three projects were investigated in terms of conflicts beyond the textual. In our study, we will look at commits across 348 projects. Second, the study did not consider what was done to fix these conflicts. We will try to answer this in RQ2 and RQ3 by looking into the effort needed to fix a conflict and the types of files the effort happens in.

Kasi and Sarma [82] analysed four projects in a manner similar to Brun et al. They found that build conflicts were present in 2%, 10%, 15%, and 4% of merge commits. Test conflicts were present in 30%, 15%, 6%, and 35% of merge commits. Kasi and Sarma also looked into the number of days till a conflict was resolved. In the case of build conflicts, the median per project was 0.75, 8, 2, and 0.75 days. Test conflicts took longer to resolve, with medians of 14, 3, 2, and 4 days. We report these results while keeping the same order of projects for every metric. Thus, for example, the first project they looked at, Perl, is the first number in each of those sequences: the reported numbers for Perl were 2%, 30%, 0.75 days, and 14 days. In this study too, only a few projects are analysed.

Perry et al. [133] performed their study in 2001. Both terminology and practices were different then than they are today. Perry et al. investigate the effect of the maximum number of parallel changes PC_{max} on the number of defects in one certain project over several years. In this system, changes are grouped in a "Modification Request" (MR). A MR has to be tested and approved before being accepted into the main branch of the project. Until a MR is in the main branch, it remains open. Perry et al. define PC_{max} for a certain file as the maximum number of open MRs at any point in the lifetime of the file. They define the number of defects for a file as the total number of MRs submitted to fix a bug in the file over its lifetime. Perry et al. control for variables such as the total number of changes or the number of defects prior to the time they analysed the project. Perry et al. find that PC_{max} significantly affects the number of defects.

Da Silva et al. [36] mine 451 open source Java software repositories in order to analyse build conflicts in merges. They performed this study in 2022. Da Silva et al. find fewer build conflicts than they a priori expected. They attribute this to procedures taken by the developers such as ensuring the project builds locally before sharing changes with the world. Most build conflicts were caused by missing symbols. This occurs when, for example, one branch starts using a variable while the other branch renames or removes it. Da Silva et al. also look into actions taken by developers to fix the encountered errors.

Amaral et al. [2] study the effect of textual merge conflicts on bug inducing changes. They locate bug inducing changes using the SZZ algorithm [144]. They find that 9.5% of commits introduce a bug. They find that this remains at 9.5% for merge commits where a textual merge conflict had already occurred. In other words, despite a textual merge conflict getting resolved by a developer, nearly one in ten of those merges still exhibits a semantic (or syntactic) merge conflict afterwards.

The numbers from the different studies provide a big range on the occurrence of semantic merge conflicts. However, the studies do indicate semantic merge conflicts are a common enough occurrence to demand researchers' interest.

5.3 Dataset

5.3.1 Origin of the Dataset

To answer the three research questions, we require a dataset of projects and information on the success of merges. To create this dataset, we combine information from two sources: GitHub [56], a version control repository host hosting primarily projects managed in Git, and Travis CI [162], a continuous integration service. When configured to do so, Travis CI builds a program, runs its test suite, and reports the results back to the developers upon a commit to the repository. We use Travis CI because, at the time that the study was conducted, many pro-

jects hosted on GitHub had been configured such that the Travis CI continuous integration pipeline triggered on every commit pushed to GitHub. In the case of open source projects, Travis CI made these results publicly available. In 2020, Travis CI stopped offering this service for free to open source projects [161] and the year prior Github had started offering a similar service, Github Actions, for free to public projects hosted on their website [49]. The result of these decisions is that Travis CI has become less popular for open source projects and their continuous integration needs.

To use the data from Github and Travis CI, we combine two existing datasets.[1] First, GHTorrent [60] provides GitHub data. GHTorrent brings information from GitHub to researchers in the form of a MySQL database containing over 400 million commits. Second, TravisTorrent [13] provides Travis CI data. Travis CI performs continuous integration on projects: it builds a project and runs its tests after each commit. The TravisTorrent dataset parses the results of the builds and tests from Travis CI and provides the information in a more structured manner. TravisTorrent provides this information for about 1300 Ruby and Java projects. To ensure projects meet a certain standard and to filter out toy projects and inactive projects, the projects in TravisTorrent meet the following criteria defined by Kalliamvakou et al. [81]:

- The project must have received a commit in the last six months. This avoids potentially abandoned projects.
- The project must have public forks on GitHub. In GitHub terminology, a fork of a project by a certain user is a copy of that project from its original namespace to the namespace of that certain user on GitHub. On the Git level, this is similar to the user having the project set up locally with a remote reference (see Section 2.3.2) to the original project. The user can then start creating new commits, forking the Git history (see Section 2.3.3) from the original project.
- The project must have received at least one pull request on GitHub. This explanation of a pull request continues from the example in the previous bullet. After creating new commits in their project, the user can ask the original project to merge in these commits. For example because the user fixed a bug or added a new feature. As the name indicates, a “pull request” requests that the original project pull in commits from the user’s fork of the project. As explained in Section 2.3.3, pulling is the combination of fetching the commits from a remote reference, i.e., the user’s fork, and merging the commits into your own branch, i.e., the original project. A pull request in GitHub enables the original project to explicitly review incoming patches

[1] Snapshots for GHTorrent and TravisTorrent were taken on 2016-05-04 and 2016-12-06, respectively.

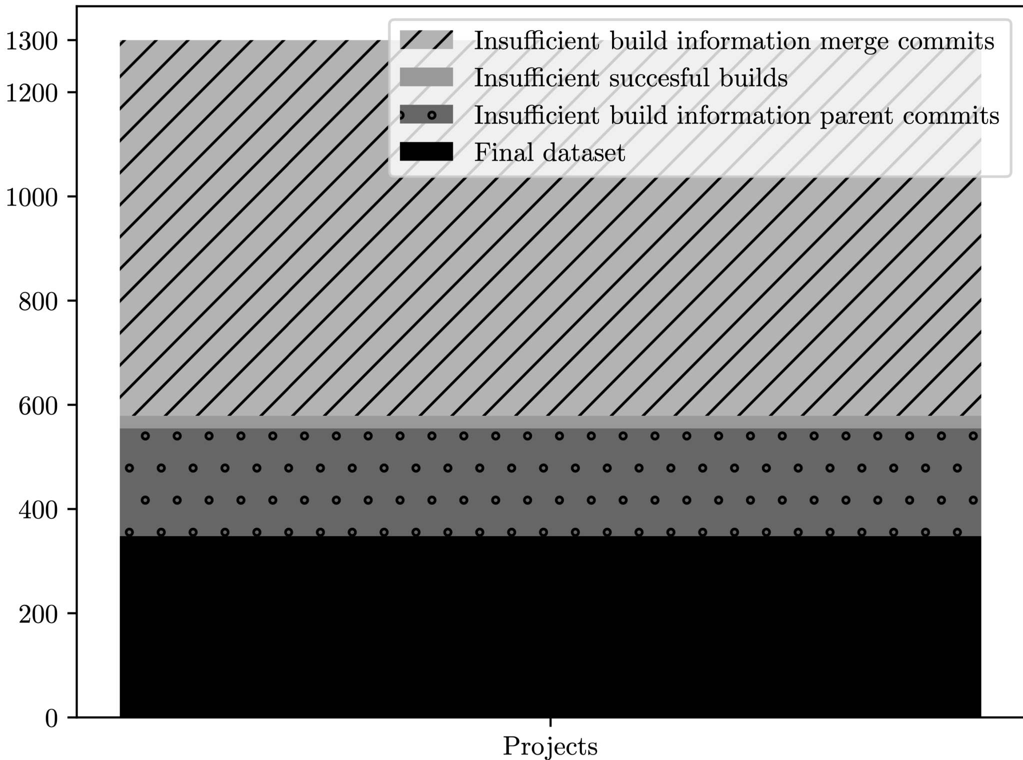

Figure 5.1: Visualisation of the refinement of the TravisTorrent dataset. In the end we are left with 348 projects.

to a repository. Contributors can review the pull request, suggest changes, or comment on it before it is merged into the repository.

- The project must have more than 50 stars on GitHub. Any user can "star" a project on GitHub, an action akin to bookmarking a website in a web browser. If a project has 50 stars, then 50 different users bookmarked the project.

TravisTorrent alone is not sufficient for our study as not enough information is provided about the commit for which the project was built and tested. We still need to identify the merge commits. For this we require GHTorrent, which enables finding information on a commit's parents commits. As explained in Section 2.3.1, each commit in a project is identified by means of an SHA-1 hash. We link commits across both datasets using this SHA-1 hash.

	Commits	Merges	Team size	Branches
Min	138	50	2	1
Q1	360	104.8	9	20.75
Median	566	155	13	53
Q3	1120	288.5	20	117.5
Max	19142	8169	288	1022

Table 5.1: A summary of the 348 projects with 50 or more builds of merge commits and parents as well as a sufficient number of successful builds across the project.

5.3.2 Refining the Dataset

We perform a three-step refinement on the dataset to ensure its projects have sufficient merge commits and adhere to continuous integration practices. A visualisation is provided in Figure 5.1.

First, the refinement eliminates projects with fewer than 50 builds of merge commits. This step leaves 579 projects.

Second, the refinement filters out projects with a build success rate under 34%. We suspect these projects of not adhering to continuous integration practices, which mandate fixing a broken build as soon as possible. The success rate of a project is the ratio of passed builds compared to the total number of builds. A "passed" build is one where nothing went wrong during the building or testing of a commit. In Section 5.4, we will describe the other possible statuses a build in Travis CI can have. Across the 579 projects, the quartiles of the success rate are 0.67, 0.81, and 0.89. The interquartile range *IQR* defines a lower bound l for the success rate:

$$l = Q_1 - 1.5 * IQR = 0.34$$

Of the 579 projects, 555 have a sufficiently high success rate. The other 24 projects have a success rate under this 34% lower bound l and are thus excluded for not adhering to continuous integration practices.

Third, our research method requires information to be present about the build of a merge commit and that of its parents. This third step eliminates projects where build information is present for fewer than 50 merge commits and their parents. This refines the dataset down to 348 projects.

5.3.3 Describing the Dataset

Table 5.1 and Figure 5.2 characterise the 348 projects in the refined dataset by: the number of commits, the number of merge commits, the maximum team size, and the number of branches. Due to the presence of some outliers, we also provide a version of each graph with the outliers removed in Figure 5.2.

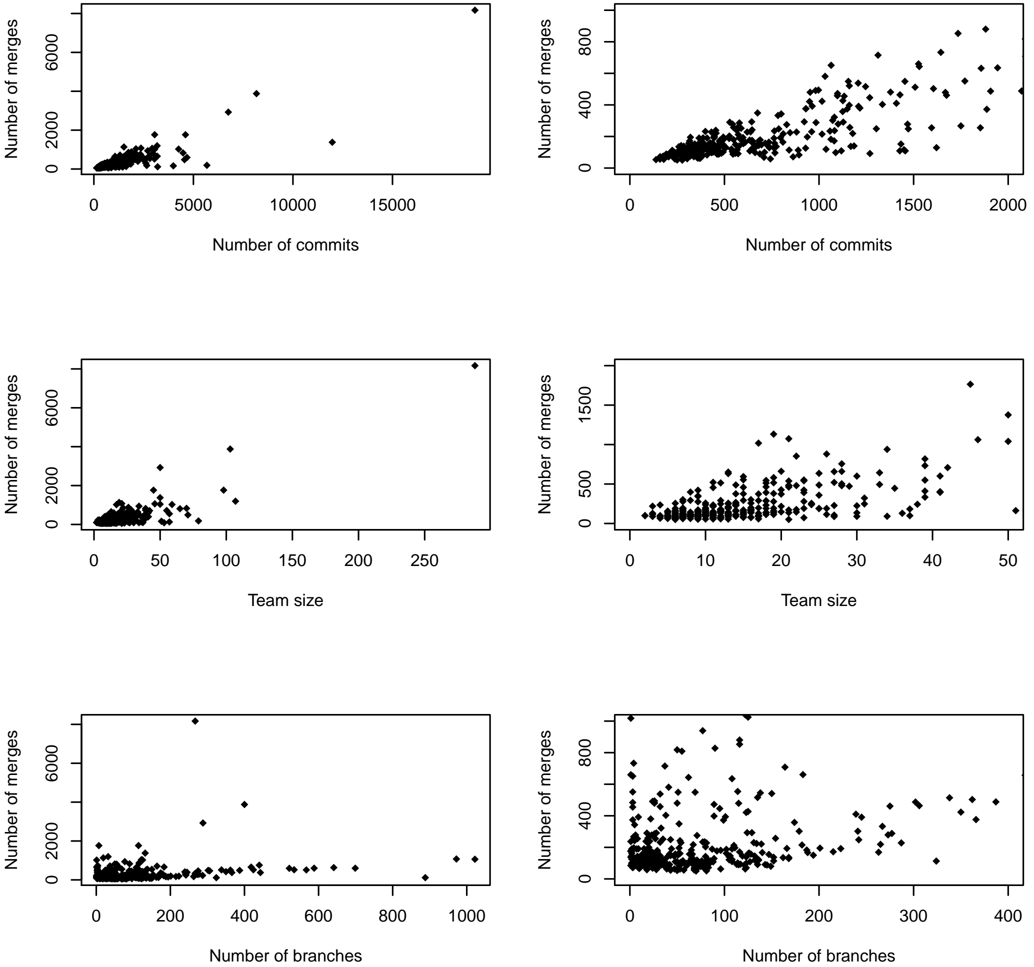

Figure 5.2: Number of merge commits plotted against, from top to bottom: number of commits, maximum team size, and number of branches. Each graph on the left hand side depicts all 348 selected projects. Each graph on the right hand side zooms in on a section closer to the origin. Every dot represents one project.

5.3.4 Terminology

Before explaining the research method for each of the research questions in Section 5.4, we define five concepts in function of the dataset: build status, breaking commit, fixing commit, merge commit, and breaking merge commit.

build status
: The build status is the status assigned to the build of a commit by Travis CI. This information is included in our dataset by way of TravisTorrent. The builds in Travis CI can have a status of "passed", "errored", "failed", "started", or "cancelled". "Started" means the continuous integration pipeline is still running. "Cancelled" is a state triggered by the project's developers if they choose to cancel a run of the pipeline. "Errored" means something went wrong in setting up the project, e.g., a dependency could not be installed. "Passed" means nothing went wrong during building or testing of the project. "Failed" means something went wrong either while building the project or while running the project's tests. The "failed" build status is therefore indicative of a syntactic or semantic bug.

breaking commit
: A breaking commit is a commit of which the build has status "failed" and of which the parent commit(s) have builds with status "passed". Considering the build status irrespective of the build status of the parents would skew results. This is because a build can remain failing for several builds in a row. Explicitly considering breaking commits in this manner enables us to ignore situations where a project's developers are no longer attempting to keep the build successful.

fixing commit
: A fixing commit is the first commit with a build status of "passed" after a breaking commit. We define the succession of builds of commits in terms of the available TravisTorrent information. Each build entry in TravisTorrent has a `tr_prev_build` field which links to the `tr_build_id` of its previous build. We repeatedly follow this link in reverse order until the first build that passes.

merge commit
: As defined in Sections 2.3.1 and 2.3.3, a merge commit is a commit with usually two, and in rare cases more, parent commits. To identify these commits in the TravisTorrent dataset, we look up the commit with the corresponding SHA-1 hash in the GHTorrent dataset. GHTorrent provides information about the parents of a commit through its `commit_parents` table.

BREAK%			The ratio for a project of the number of breaking commits ("failed" after "passed") to the total number of commits after a passing build (anything after "passed"). A lower number is better.
$BREAK\%_R$	$BREAK\%_M$		*BREAK*% for regular and merge commits, respectively.
—	$BREAK\%_{MN}$	$BREAK\%_{MPR}$	Merge commit numbers are further split between merges not resulting from a pull request and those that do.

Table 5.2: Definition of the *BREAK*% metrics for RQ1.

breaking merge commit
: A breaking merge commit is a merge commit that is also a breaking commit. In this case the build status was "passed" for the parents of the merge commit, while the status of the merge commit itself is "failed". In other words, this situation indicates a syntactic or semantic merge conflict.

5.4 Research Method

5.4.1 RQ1: How Often Do Code Integrations Lead To Syntactic And Semantic Conflicts?

Our research method for RQ1 consists in analysing the frequency of breaking commits. For each project in our dataset, we compute *BREAK*%, the ratio of breaking commits to all commits. We do this separately for breaking merge commits, $BREAK\%_M$, and for breaking regular commits, $BREAK\%_R$. Breaking merge commits are then further categorised into those resulting from pull requests, $BREAK\%_{MPR}$, and those not resulting from pull requests, $BREAK\%_{MN}$. In Table 5.2, we provide a hierarchical overview of these *BREAK*% metrics. We identify pull requests through the `gh_is_pr` field present in TravisTorrent.

5.4.2 RQ2: How Much Effort Is Needed to Fix Conflicts After Code Integration?

Our research method for RQ2 involves the measuring of proxies for the effort involved in fixing a build. We use three metrics. The first metric is the number of builds needed to fix a breaking commit. This number is the number of steps

as described in finding the fixing commit in Section 5.3.4. We prefer to look at the number of builds over the number of commits. When several commits are pushed at once, Travis CI will only build the last one. Our reasoning here is that a developer may take a few commits to fix the bug, but will not push their changes until the fix is ready. The second metric is the number of changed lines between the breaking and the fixing commit. The final metric measures the time between breaking and fixing commit. The metric considers the `gh_build_started_at` field provided by TravisTorrent. `gh_build_started_at` has a precision of a day. The measured differences will thus also have a precision of a day.

To summarise:

1. *NBTF*: The number of builds to fix: how many builds it takes before a breaking commit is fixed. A lower number is preferred.
2. *LINES*: The number of lines changed between the breaking and fixing commit. A lower number indicates a possibly lower effort.
3. *TTF*: The time between the breaking commit and its fixing commit. A lower number may indicate the bug was easier to fix.

For every metric M, we also define $\overline{M}$ as its median within a project.

5.4.3 RQ3: What Type of Files Is the Effort to Fix Conflicts Concentrated In?

To answer RQ3, we categorise a fix into one of four categories. To do so, we consider the `git_diff_src_churn` and `git_diff_test_churn` fields of TravisTorrent. These indicate whether there were changes made to the source code and the test code, respectively. We take the sum of all the changes between the breaking merge commit and its fixing commit. The four categories are then defined in function of those numbers:

1. the “source” category contains fixes with only changes to the source code, i.e., `git_diff_src_churn` is greater than zero and `git_diff_test_churn` is zero,
2. the “test” category contains fixes with only changes to the test code, i.e., `git_diff_src_churn` is zero and `git_diff_test_churn` is greater than zero,
3. the “both” category contains fixes with changes to both source and test code, i.e., both `git_diff_src_churn` and `git_diff_test_churn` are greater than zero, and
4. the “none” category contains fixes with changes to neither source nor test code, i.e., both `git_diff_src_churn` and `git_diff_test_churn` are zero.

For each project, we count the number of fixes in each category relative to the project's total amount of fixes. Based on the four categories, we define the following four metrics per project:

1. *SRC*: the ratio of "source" fixes to the total number of fixes.
2. *TEST*: the ratio of "test" fixes to the total number of fixes.
3. *BOTH*: the ratio of "both" fixes to the total number of fixes.
4. *NONE*: the ratio of "none" fixes to the total number of fixes.

5.5 Results

For each of the three research questions posed in Section 5.1, we provide and discuss the results, we provide a brief takeaway message, and we discuss threats to validity.

5.5.1 RQ1: Frequency of Conflicts

For RQ1 we consider the *BREAK%* metrics. The metric uses the previous build for regular commits as defined in TravisTorrent. The metric only considers merge commits with exactly two parents. The dataset resulting from Section 5.3 has exactly one merge commit with more than two parents. This lone merge commit is not used in further analysis.

Table 5.3 and Figure 5.3 (a) depict $BREAK\%_R$ and $BREAK\%_M$, the *BREAK%* for regular commits and merge commits respectively. We notice merge commits break the builds *less* often than regular commits do. Figure 5.3 (b) splits up the merge commits into two categories: non pull requests and pull requests. We believed the pull requests might explain away the good behaviour of the merge commits. However, this does not seem to be the case. Only 35 of the selected breaking merge commits across all the 348 projects are marked as a pull request. Filtering these out does not have a significant impact on the combined numbers.

This result could be explained through our commit selection. We pick *breaking* commits, i.e., commits for which the build not just fails, but the build of the parent commit(s) also passes (see Section 5.3.2). Regular commits are more often something completely new to the source code. The changes in the regular commit are directly to blame for the breaking of the build. Breaking merge commits on the other hand combine two passing versions. The only way for a merge commit to break the build is to have the source code from both branches interact in an unexpected way. These errors can be subtle. While new changes can be made in the merge commit, this is not necessarily the case. Instead, the error then lies in the way either branch interacts with the other. Furthermore, as we discussed in

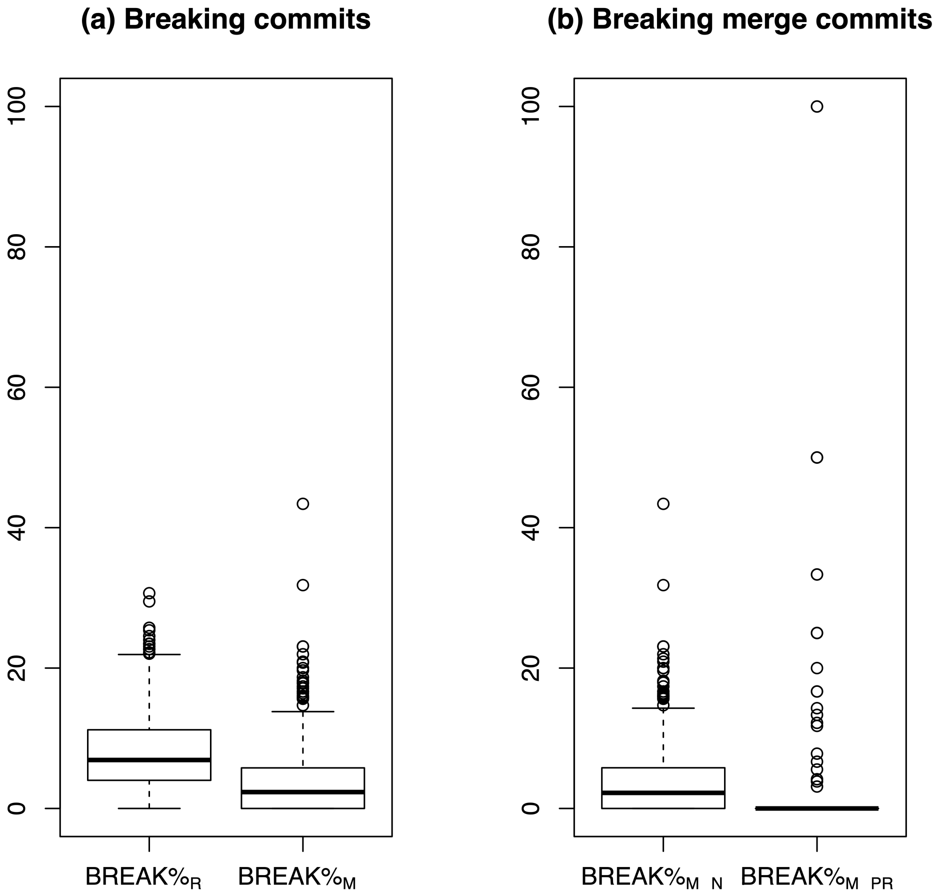

Figure 5.3: A comparison over all projects of the *BREAK*% metrics. (a) splits up breaking commits by regular commits and merge commits. (b) splits up the breaking merge commits by pull request.

	$BREAK\%_R$	$BREAK\%_M$	$BREAK\%_{MN}$	$BREAK\%_{MPR}$
Min	0.00	0.00	0.00	0.00
Q1	4.02	0.00	0.00	0.00
Median	6.90	2.34	2.22	0.00
Q3	11.20	5.78	5.79	0.00
Max	30.67	43.40	43.40	100.00

Table 5.3: A summary of the *BREAK*% for all 348 projects.

Section 3.3.4, the potential presence of merge conflicts affects the behaviour of developers: they coordinate in an attempt to avoid merge conflicts of any kind.

Table 5.3 shows that half of the projects deal with a breaking merge commit at least once every 43 merge commits.[2] For a quarter of the projects this occurs at least once every 17 merge commits.[3]

Summary. A breaking merge commit happens less frequently than a breaking regular commit in projects with a continuous integration pipeline that maintain a 34% or higher success rate.

Threats to Validity. Merge commits are but one form of code integration. The manual application of a patch or a Git rebase would not show up in the Git history [33]. As we described in Section 2.3.3, rebasing rewrites the history of a project to make it seem as if commits were made sequentially rather than in parallel over different branches. This study does not consider these forms of code integration.

TravisTorrent contains projects that adhere to the GitHub workflow. The projects actively use branches, forking, and pull requests. This limits our analysis to this type of project.

5.5.2 RQ2: Effort to Fix Conflicts

We start out with 16413 breaking commits (14430 regular, 1983 merge) from the 348 projects after removal of outlier projects in Section 5.3.2. For 8453 (7664 regular, 789 merge) of the breaking commits a fixing commit is found. The merge commits are spread out over 203 projects.

The $\overline{NBTF}$ metric is 1 for 87% of projects. This indicates a breaking merge commit is usually fixed with the next build. Performing the same analysis on the number of commits rather than the number of builds gives similar results: 71% of projects tend to require just one commit.

Figure 5.4 depicts the $\overline{LINES}$ metric. It has quartiles at 3.25, 9, and 36. The inset zooms in on the left part of the graph. The inset still shows 88% of the projects. Half of the projects repair breaking merge commits usually with up to nine lines.

Table 5.4 summarises the $\overline{TTF}$ metric. $\overline{TTF}$ shows 67% of projects usually fix a breaking merge commit the same day. Within a week, 94% of projects have fixed a breaking merge commit.

[2] $1/43 \approx 2.3\%$, in other words the median for $BREAK\%_M$ in Table 5.3.
[3] $1/17 \approx 5.9\%$, in other words the third quartile for $BREAK\%_M$ in Table 5.3.

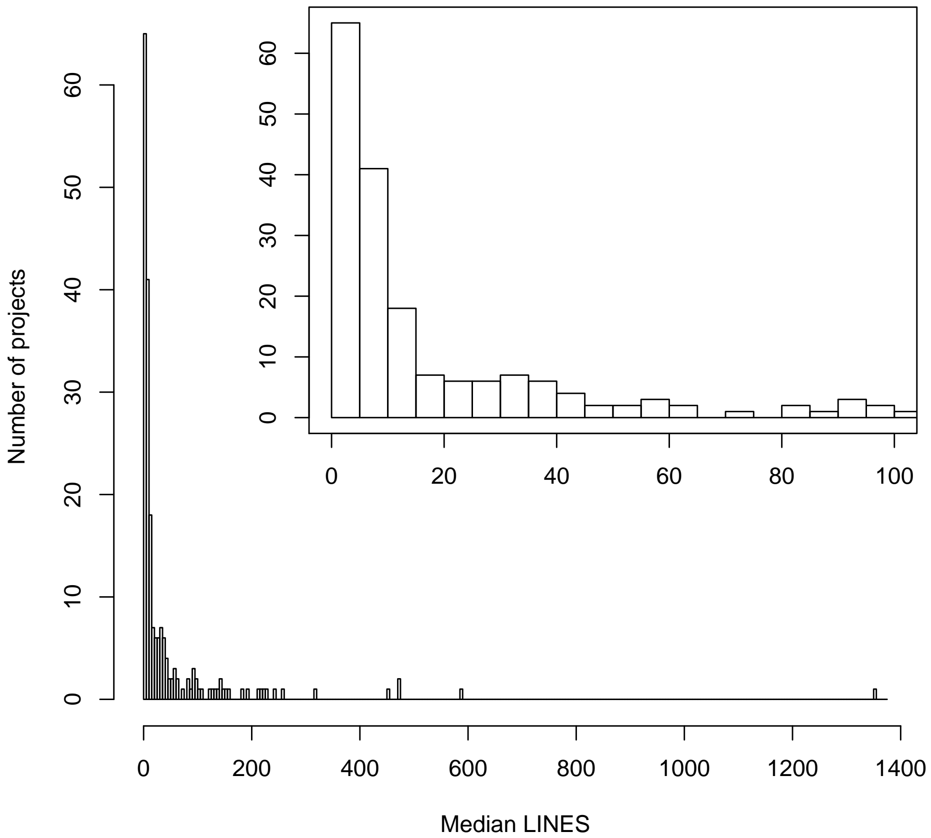

Figure 5.4: $\overline{LINES}$ for every project. Despite the long tail, for 75% projects $\overline{LINES}$ is less than 36. The inset zooms in on the lower end of the graph. The inset still shows 88% of projects.

	Usually fixed
the same day	67%
the next day	14%
the same week	13%
the same month	5%
more than a month	1%

Table 5.4: An overview of the $\overline{TTF}$ metric. It shows 67% of projects usually fix a breaking build of a merge commit within a day.

Summary. In most projects a breaking merge commit is usually fixed with one build on the same day by changing fewer than ten lines of code.

Threats to Validity. Our method for identifying the fixing commit relies on finding the next builds in TravisTorrent. However, TravisTorrent does not provide this information in the case of merge commits. A fixing commit will not be found if a breaking commit is fixed by a merge commit or a merge occurs between the breaking and the fixing commit.

We narrowed down our breaking merge commits to just 789 that have an associated fixing commit. These 789 are spread out over 203 projects. This does not leave a lot of data per project. This may skew the results in favour of what happens in those projects with very few data points.

Clearly all metrics are but a proxy for effort. It may take a lot of effort to track down the exact problem of an issue, while still fixing it with but one line of code in one commit. The $\overline{TTF}$ metric used does not necessarily represent the actual time a developer spent working on fixing the build. The dataset comprises open source projects which are, in general, developed by volunteers on an irregular basis.

5.5.3 RQ3: Source vs Test

Figure 5.5 depicts how the metrics defined in Section 5.4.3 are spread out across all projects. Here we use the same 789 breaking merge commits with associated fixing commit as were identified in Section 5.5.2. Figure 5.5 shows most breaking merge commits are fixed by changes to either exclusively the source code or to both source and test code. From this we conclude that the developer must indeed have had certain expectations about the source code, not the test code, and that those expectations were not met.

Summary. Breaking merge commits are fixed by changes to the source code.

Threats to Validity. This analysis is done for those breaking merge commits for which a fix was found. Only 789 such cases were found. There is not a lot of data per project. This may skew the results in favour of what happens in those projects with very few data points.

This study includes syntactic issues. Syntactic issues in the source code are necessarily fixed by making changes to the source code. Similarly, a missing import or incorrect references to now renamed variables also require source code changes. This skews the result away from fixes to the test code.

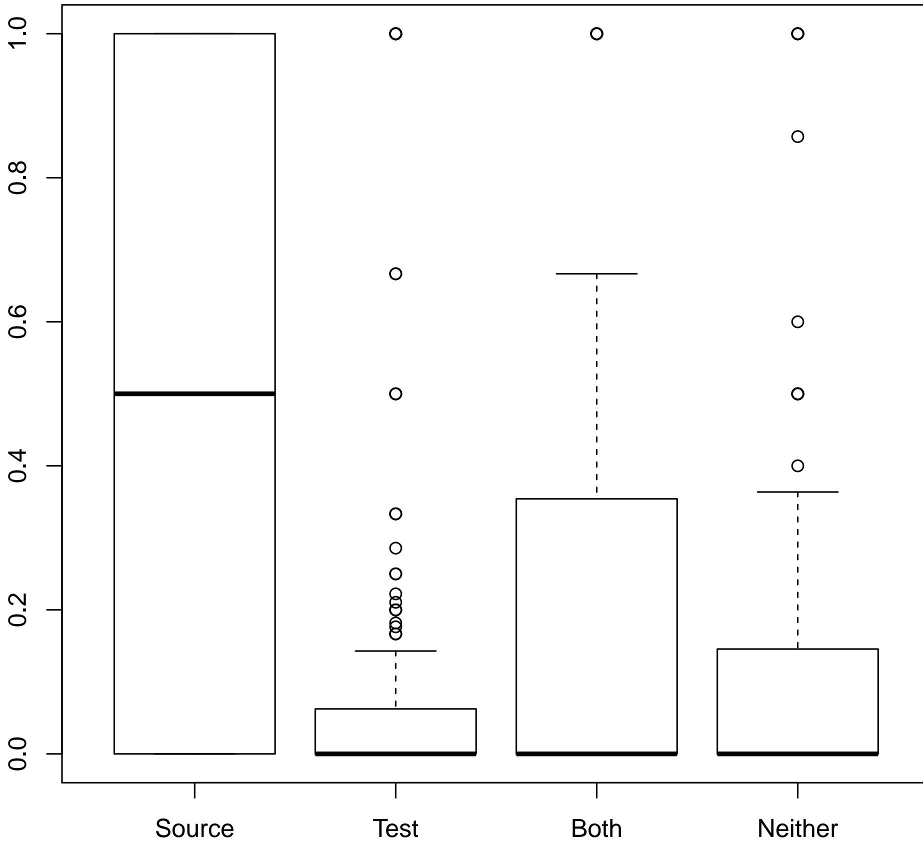

Figure 5.5: *SRC*, *TEST*, *BOTH*, and *NONE* metrics. Breaking merge commits in the majority of projects are usually repaired by changes to the source code.

5.6 Conclusion

We looked into the prevalence of syntactic and semantic merge conflicts on a large scale within the context of continuous integration. Using data from GitHub and Travis CI, we analysed breaking commits: commits for which the build fails and the build of their parent commit(s) passed.

We found that half of analysed projects deal with a breaking merge commit at least once every 43 merge commits. For a quarter of projects, this occurs at least once every 17 merge commits. We found breaking merge commits occur less often than breaking regular commits. Breaking merge commits are repaired with relatively little effort. Repairing is often done the same day and with just one build. Fewer than ten lines of code need to be changed to repair a breaking merge commit. Most of the changes are done in the source code, as opposed to

test code or other places.

Semantic conflicts are more subtle than textual conflicts and may otherwise go undetected until all tests are run or a user encounters its effects. Given their observed prevalence, we recommend further research on tools that warn developers about potential semantic merge conflicts. In Chapter 6, we present such an approach and prototype tool to detect semantic merge conflicts by means of symbolic execution.

Chapter 6

Symbolic Execution to Detect Semantic Merge Conflicts

In Sections 2.2 and 2.3, we described how version control software enables developers to work on projects in parallel. As each developer works on different features or bugs, the project history forks into many different branches. Eventually, the different branches need to be merged together again.

In Section 3.2, we described how merging branches together can introduce conflicts, specifically: textual, syntactic, and semantic conflicts. In Section 3.3, we focused on semantic merge conflicts: merges where there was no problem textually or syntactically, but where the interaction of the branches introduces unwanted behaviour. We also described the negative impact semantic merge conflicts have. In Chapter 5, we studied their prevalence and found them to be rare, but in combination with their problematic nature common enough to warrant tool support.

In this chapter, we present an automated approach to detecting semantic merge conflicts by means of symbolic execution (see Section 2.5). Our approach defines the program semantics as path conditions, produced by the symbolic execution engine, and checks whether the conditions satisfy established rules that indicate a merge conflict. Our usage of symbolic execution to check these rules is novel. We develop a prototype that warns developers in the case of a semantic merge conflict and thereby helps developers avoid them.

To evaluate our approach and prototype, we first perform a retroactive study to detect semantic merge conflicts using heuristics. The results of the retroactive study are used to evaluate our proactive detection of semantic merge conflicts. Our evaluation shows that, in specific cases, our approach using symbolic execution is a promising extension to existing mechanisms to semantic merge conflict detection.

Section 6.1 sketches our proposed solution and the context in which it operates; Section 6.2 discusses the related work; Section 6.3 explains our approach that uses symbolic execution to detect semantic merge conflicts; Section 6.4 describes a prototype implementation of our approach, listing technical details and limitations; Section 6.5 evaluates the prototype; Section 6.6 concludes the chapter.

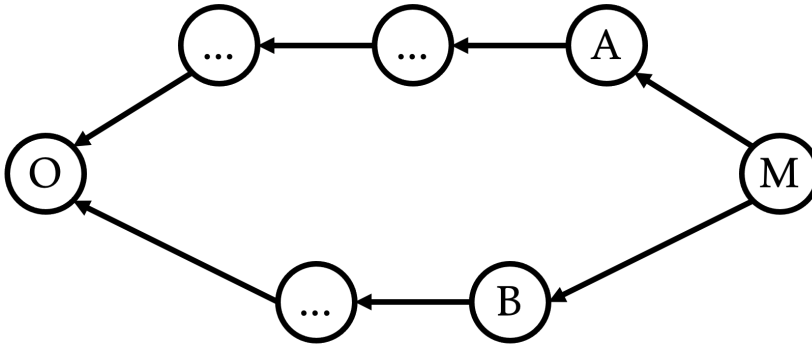

Figure 6.1: The four relevant parts when talking about merging and merge conflicts: the merge commit M, its direct parent commits A and B, and their common ancestor commit O. The arrows indicate a child-parent relation. Reproduction of Figure 3.1.

6.1 Proposed Solution

6.1.1 Context

We reproduce Figure 3.1 here as Figure 6.1 for context. Specifically, we reiterate the naming convention we use when talking about a merge commit.

- The merge commit M, combining the code from two branches.[1]
- The direct parent commits A and B, which represent the state of the two branches prior to being merged. These will be referred to as either branch A or commit A (respectively, B).
- The common ancestor commit O, representing the final point before the histories for A and B diverged. The difference between O and A (respectively, B) represents all the changes made in that branch.

We also require O, A, B, and M to be strictly different commits, thus excluding both fast-forwarded merges, which cannot be identified as a merge in the Git history, as well as trivial merges where no change was made in either the A or B branch. Both of these we described in more detail in Section 2.3.3.

A semantic merge conflict is thus unintended behaviour in M due to an interaction of the behaviour in A and B. In this chapter, we target Java and specifically focus on conflicts that are not already caught by its compiler. Semantic merge conflicts are a problem (see Section 3.3). There are no standardised tools to detect or solve them, thus forcing developers to come up with ad hoc ways to do so.

[1]As in previous chapters, we do not consider the esoteric case where more than two branches are merged together.

Merge conflicts also cause more restrictive behavioural changes by developers to try and avoid situations that could lead to the conflicts. While a comprehensive test suite can help with semantic merge conflicts, it incurs its own development and maintenance burden.

6.1.2 Approach

In this chapter, we present an approach to detecting semantic merge conflicts by means of symbolic execution. As we detailed in Section 2.5, symbolic execution [26, 86] is a program analysis technique which, as a byproduct of its analysis, creates path conditions: combinations of constraints placed on symbolic variables representing input and output. We define the program semantics in terms of the path conditions as produced by a symbolic execution engine. The path conditions are collected for each of the four versions in the merge: O, A, B, and M. Path conditions can be fed to a constraint solver to check satisfiability or to provide concrete values meeting the constraints. Our approach checks equivalency between the path conditions to see whether they satisfy established rules that indicate a merge conflict. Our usage of symbolic execution to check these rules is novel. We also develop a prototype implementing our approach. The prototype warns for semantic merge conflicts and thereby helps developers on avoiding and solving them.

6.1.3 Situating Our Approach

We envision our approach to be used as part of an existing pipeline of tools already present in the developer's toolbox. Figure 6.2 depicts our prototype's place in a chain of the minimal set of tools used in a (compiled) programming language. The goal of our approach is to detect[2] semantic merge conflicts. The approach and prototype will be detailed in Sections 6.3 and 6.4. Other steps may also be present in this pipeline, such as a test suite.

The first step in this pipeline is Git's default three-way merge (see Section 2.3.3) which takes the code in A and B and merges it together to create M. This works on a textual level and, as we described in Section 3.2, can lead to a textual merge conflict: the same line is changed in different ways in A and B, when compared to O, and Git does not know which version to choose. Instead, the developer is asked to resolve the textual merge conflict.

If Git does not encounter a conflict, or if the developer resolved it, then the next step is to compile the code in version M. While this is not specific to merges, it does locate syntactic and certain semantic merge conflicts. Any syntactic merge conflict will have led to incorrect syntax in M. This cannot be detected by Git's

[2] In this chapter, we use the terms detecting and classifying a conflict interchangeably.

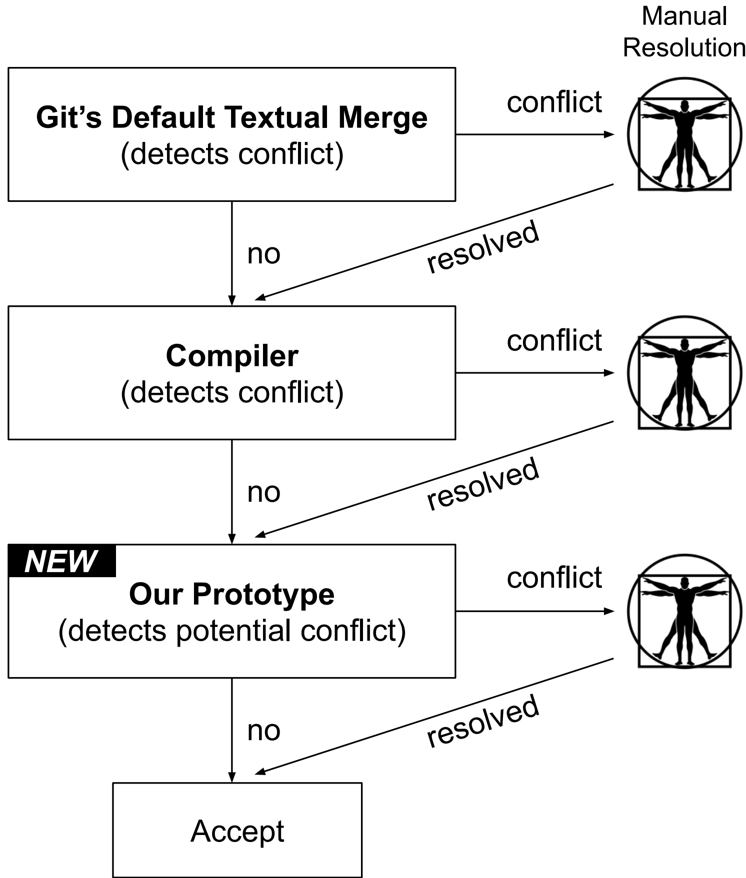

Figure 6.2: The pipeline of steps to realise a merge commit that is free of conflicts. The process eventually leads to an accepted merge.

merge algorithm, but will produce an error by a compiler. As we discuss in Section 3.2, there is also research into syntactic merging, but this is yet to see an uptake by developers. Compiling M will also find some types of semantic merge conflicts, we give two examples: (1) one branch renaming a variable while the other branch adds a use of that same variable, and; (2) one branch removing an included library, e.g., an `import` statement in Java, that the other branch started using. Whichever the issue, the developer is once again expected to resolve the conflict.

Finally, we envision our approach to be part of the next and final step. Our approach aims to detect semantic merge conflicts within parts of the program by using symbolic execution. The classification of a conflict here depends on the behaviour of the program. Semantic conflicts are challenging because the

programs in both branches often differ in behaviour by intention. Our approach warns the developer about some semantic merge conflicts. As in the previous steps, the developer is expected to resolve the conflict themselves.

A merge that reaches the end of this pipeline is considered conflict-free.

6.1.4 Evaluation

Semantic merge conflicts are relatively rare in the wild, as we showed through our study in Chapter 5, and standardised datasets for benchmarking are missing.[3] This complicates the evaluation. To evaluate the prototype implementation, we investigate the following two research questions:

RQ1 Can we classify semantic merge conflicts retroactively by heuristics computed on the full revision history?

RQ2 Can we classify semantic merge conflicts proactively by symbolic execution?

We use the retroactive classification (RQ1) to get a pool of candidates with a realistic chance of being a semantic merge conflict. The pool is then used for the evaluation of our proactive approach by means of symbolic execution (RQ2). Using a random sample of projects and merge commits is unrealistic due to the small number of true semantic merge conflicts.

Our evaluation shows that, in specific cases, our approach using symbolic execution is a promising extension to existing mechanisms to merge conflict detection.

6.1.5 Contributions

1. We present an approach to detect semantic merge conflicts by symbolic execution.

2. We implement a prototype implementation of the approach for detecting Java merge conflicts that is used for the evaluation of our approach on synthetic and on empirical data.

3. We present an alternative heuristics-based approach to retroactively classify commits from the full revision history as semantic merge conflicts. We use this approach to compose a dataset for the evaluation of our prototype.

[3] The results from the study in Chapter 5 cannot be used either. There we studied using Travis CI build status, which includes syntactic and semantic merge conflicts.

6.2 Related Work

We continue with an in-depth discussion of the related approaches. We position this chapter as the first that classifies (or detects) semantic merge conflicts by symbolic execution.

We focus on approaches dedicated to three-way merging. We do not discuss related work that compares two versions of a program, but list some instances here (see [122, 125, 134]).

We introduce the following three dimensions to structure the related work discussion. The first two reflect the structure of this section.

- Program analysis vs. data-driven. We see differences in how approaches classify or resolve conflicts. Approaches can be based on program analysis. This is the case for our approach. Other approaches are 'empirically' motivated using collected data or existing knowledge on merges.
- Classification vs. resolution. This section distinguishes between approaches that (1) classify merge conflicts for a subsequent manual resolution, i.e., detection, and approaches that (2) automatically resolve or prevent conflicts. Approaches for classification and resolution apply related techniques. Our approach provides a classification.
- Semantic vs. non-semantic. This concerns the different types of merge conflicts we discussed in Section 3.2.

We will focus on semantic merge conflicts in this section, but we will mention alternatives, if they are part of relevant related work.

6.2.1 Program Analysis Approaches

We start with a discussion of approaches that do a classification of conflicts based on a program analysis. Our approach falls into this category.

Classification of Conflicts

Sousa et al. [145] define semantic conflicts as the introduction of unwanted behaviour. This is comparable to our definition. They introduce SafeMerge, a tool that considers a merge as four modifications to a common base. They apply a lightweight analysis to the shared parts and verify modifications — in particular the return values of a program. SafeMerge is evaluated on real-world code, but the authors note that a semantic conflict is hard to define objectively. They evaluate against their own definition of a semantic merge conflict, inherent to their approach (Definition 4.4, page 7, in [145]). Our qualitative evaluation instead attempts to mitigate this by judging a conflict through the presence of bug fixing

commits following a merge commit. Our definition of a conflict is closely related to the definition used by SafeMerge. They compare returned variables similarly to our definition, as well will discuss in Section 6.3.2. However, we use symbolic execution to instantiate the definition.

Da Silva et al. [35] consider static analysis tools, such as symbolic execution, as too heavyweight. Instead, Da Silva et al. use test generation. Their approach generates tests using the program versions A and B in order to capture the behaviour. The tests are then executed on versions O and M. When a test fails in O and M, but succeeds in A or B, then there is a possible conflict. We generalise this definition further in Section 6.3.2. Da Silva et al. report that many false negatives are produced by their approach. They state this is due to the generated tests not creating enough assertions to verify all the code, even if the tests do often execute the code. This reflects the complexity of classifying semantic merge conflicts.

Wuensche et al. [172] classify build and test conflicts. Their approach creates a directed call graph of the code and its modifications to detect a conflict if: (1) changes are made on the same call graph node, (2) there is a path from one change to another, or (3) there is a path from an unchanged node to two changed nodes. This is clearly different to symbolic execution and instead operates more on the syntactic level. In an evaluation, Wuensche et al. show that the approach helps to catch build conflicts. However, test conflicts are rare in the sample that Wuensche et al. use for the evaluation. Hence, statements on whether the approach detects test conflicts are not possible. This reflects a challenge for our evaluation too. The results of Wuensche et al. show that semantic conflicts are relatively rare. We resolve this by producing a pool of promising semantic conflict candidates by means of a retroactive search method which operates on the full revision history. We use this sample of candidates in the evaluation of our approach.

Pastore et al. [131] use specification mining [3] to generate behavioural program models for versions O, A, and B. They run a program's existing test suite to collect values for variables, after which pre- and post-conditions are derived from the observed values. If changed conditions differ between A and B, according to a constraint solver, a conflict is reported. In our approach, we generalise this definition further, as will be described in Section 6.3.2. Our approach does not require an existing test suite and also takes into account the behaviour of version M.

Resolution of Conflicts

Another branch of related work focuses on automatically resolving conflicts.

MrgBldBrkFixer [152] is a tool that aims to automatically resolve conflicts, such as inconsistent symbol renaming. When identifying a renamed symbol that causes the build to break, MrgBldBrkFixer filters the commits after the last correct build and uses them to generate patches to resolve the inconsistency.

Horwitz et al. [74] use program dependency graphs and program slicing to create graphs representing the changed code in both branches. If the graphs do not overlap, there is no conflict. Merge conflicts are automatically resolved by merging overlapping graphs, deriving the merged code from the merged graphs. This is a more syntactic resolution of a conflict.

Berzins [16] approaches semantic merge conflicts in a similar manner to the structured merging approach to syntactic merge conflicts. We briefly discussed structured merging in Section 3.2.2. Berzins creates a formal definition of semantic merging in a language-independent manner. If conditions in this definition are met, it describes how to construct a semantically conflict-free merge candidate M. Much like with structured merging, this requires a fallback: if the conditions are not met, the approach does not create a merge candidate at all. Another downside is that each language must also be converted to the representation used by Berzins.

6.2.2 Data-driven Approaches

Other approaches are data-driven instead, using collected data or existing knowledge on merges, to classify or resolve conflicts.

Classification of Conflicts

The Bucond tool, presented in [160], creates a program entity graph for version O, A, and B. The program entity graph contains standard AST information but also def-use relations. A comparison of the different graphs is searched for predefined conflict patterns. Conflict patterns reflect very specific information that we consider as empirically motivated. A merge conflict is reported if a pattern matches. Compared to our approach using symbolic execution, Bucond is limited to predefined patterns. We rely on the compiler to catch the merge conflicts that Bucond detects (as depicted in Figure 6.2 and discussed in Section 6.1.3).

Somewhat related is the classification of pull requests. Models that predict the acceptance of pull requests are trained on empirical data. An example can be found in [163]. However, non-acceptance of pull requests may not necessarily be caused by a merge conflict.

Resolution of Conflicts

DeepMerge [44] uses a machine learning model to suggest resolutions for a textual merge conflict. DeepMerge's merge resolutions for projects written in JavaScript outperform a structured merging tool (see Section 3.2.2). The authors do note that structured approaches tend to have more trouble with a dynamic language such as JavaScript compared to how they perform in statically typed languages such as Java. MergeBERT [153] improves on the DeepMerge approach

and also targets textual merge conflicts. Learning from existing data on resolutions is also employed by, for example, Pan et al. [126] and by Almost Rerere [65].

Our approach does not rely on data. However, we assume the combination of machine learning with symbolic execution to be promising future work.

6.3 Detecting Merge Conflicts by Symbolic Execution

This section describes our approach to detecting merge conflicts by symbolic execution. We structure this section as follows:

- In Section 6.3.1, we describe a merge conflict in terms of a property P. The property is a placeholder. We later define the property such that it reflects the semantics of a program version.
- In Section 6.3.2, we formulate basic rules that describe the merge conflict given a property P. This leads to a classification as merge conflict. The rules are inspired by the related work.
- In Section 6.3.3, we describe instantiating property P using symbolic execution. This results in a classification as a merge conflict based on the semantic notion of symbolic execution.
- In Section 6.3.4, we describe how to decompose a merge conflict that arises from such a definition, into a more fine-grained notion of a conflict.

6.3.1 Property P

Our definition of a conflict is based on a property that we denote as P. The property P is a placeholder. It is specific to the four program versions involved in a merge. This can be described as a 4-tuple: (P_O, P_A, P_B, P_M).

For illustration, we consider a concrete example in which we analyse the presence of a particular line of code. We define P as a boolean function, checking whether the line is present in the origin O, the branches A and B, and the merge M. An example is a program where branch A adds a line which is not present in the origin O nor in branch B. If the line also shows up in the merge, then the tuple is $(False, True, False, True)$.

In Section 6.3.3, we switch to a semantic notion of such a property, created by symbolic execution.

6.3.2 Merge Conflict of Property P

We now define a merge conflict as a function of the 4-tuple for property P. We also report on the intuition why a violation, by our definition, might be a conflict.

In Section 6.5, we evaluate this definition. Our definition comes close to related work which compares other properties in a similar manner: Horwitz et al. [74] and Sousa et al. [145] compare returned variables, Da Silva et al. [35] consider tests, and Pastore et al. [131] compare conditions obtained through specification mining, although without considering merge M.

In our definition, a merge conflict arises if one (or more) of the following three constraints is violated:

CF-A Conflict Freedom A: We say a merge conflict occurs if

$$(P_O \neq P_A) \rightarrow (P_A = P_M)$$

is violated. The intuition is that if the property differs between the common origin O and the branch A, i.e., $P_O \neq P_A$, then there was a conscious change by the developer. As such, we expect the merge property P_M to reflect this by being equal to property P_A. This is $P_A = P_M$.

CF-B Conflict Freedom B: Analogous to CF-A.

$$(P_O \neq P_B) \rightarrow (P_B = P_M)$$

CF-AB Conflict Freedom A and B: We face a merge conflict if

$$(P_A = P_B) \rightarrow (P_A = P_B = P_M)$$

is violated. The intuition is that if the property is the same in both branches, i.e., $P_A = P_B$, then we also expect this property to be present in the merge M, i.e., $P_A = P_B = P_M$. This covers both the situation where P_A and P_B are unchanged compared to P_O as well as the situation where P_A and P_B are changed in the same manner compared to P_O, i.e., a special combination case of CF-A and CF-B.

The definition of equality (=) and inequality (≠) depends on the way we define the property P. The arrow → denotes a logical implication.

Consider again the concrete example from Section 6.3.1, where the property P is defined as a boolean function indicating the presence of a certain line of code. Here the equality (=) is the standard boolean equality operator. One possible reason for a violation of CF-A is that a line is added in branch A, but the line is not present in the merge M. One possible reason for a violation of CF-AB is that a line has been added in both branches, but it is not present in the merge M.

Version control systems, like Git, already apply comparable rules when performing a merge requested by the developer. A merge is constructed in a manner that the above rules are not violated.

In what follows, we switch from the basic example about the presence of a certain line to a semantic property of the program and thereby to a semantic conflict.

6.3.3 Semantic Merge Conflict

We start with a simplified discussion of a program with a single method that is changed independently in two branches A and B. Both versions are eventually merged in merge commit M. In Section 6.3.4, we explain how to extend this idea to make statements about different structural elements of the program and thereby decompose a semantic conflict.

Symbolic execution typically results in multiple path conditions, due to the control structures in a program, such as `if` or `while`. We avoid this detail in this section and consider a program with just one path and thus just one path condition. We then extend this approach to multiple paths in Section 6.3.4.

Our approach uses a symbolic execution engine to define the property P as a path condition. A constraint solver (see Section 2.5) is used to define equality between path conditions. The property P thereby captures the behaviour of the method.

Equality and inequality, previously denoted as $=$ and $\neq$, are now defined using an equivalency check by a constraint solver. To emphasise the difference, we denote the equality proven by the constraint solver as $\Leftrightarrow$ and $\not\Leftrightarrow$. The symbol $\rightarrow$ still denotes a logical implication.

Our rules thereby instantiate as follows:

CF-A $(P_O \not\Leftrightarrow P_A) \rightarrow (P_A \Leftrightarrow P_M)$

CF-B Analogous to CF-A, $(P_O \not\Leftrightarrow P_B) \rightarrow (P_B \Leftrightarrow P_M)$

CF-AB $(P_A \Leftrightarrow P_B) \rightarrow (P_A \Leftrightarrow P_B \Leftrightarrow P_M)$

In essence, these rules state that if we face a semantic change in one of both branches, we expect the new behaviour to be present in the merge too (CF-A or CF-B). If both branches are semantically equivalent, we expect them to be semantically equivalent to the merge (CF-AB).

We consider again the semantic merge conflict example in Listings 3.9 to 3.12 which was previously discussed in Section 3.2.3. The example is reproduced in Listings 6.1 to 6.4. In this example, we have the following path conditions as properties for O, A, B, and M. Here, x, y, and r are symbolic variables for `x`, `y`, and the return value, respectively.

$$
\begin{aligned}
P_O &: r = x + y \\
P_A &: r = x + y + 1 \\
P_B &: r = x + y + 1 \\
P_M &: r = x + y + 2
\end{aligned}
$$

Substitution of the properties into the rules CF-A, CF-B, and CF-AB indicates a semantic merge conflict; all rules are violated.

```
1 public int myAdd(int x, int y) {
2   int z = x + y;
3   return z;
4 }
```

Listing 6.1: Version O in a semantic merge conflict. myAdd sums two integers.

```
1 public int myAdd(int x, int y) {
2   int z = x + y + 1; // Modified
3   return z;
4 }
```

Listing 6.2: Version A in a semantic merge conflict. myAdd sums two integers and adds one.

```
1 public int myAdd(int x, int y) {
2   int z = x + y;
3   return z + 1; // Modified
4 }
```

Listing 6.3: Version B in a semantic merge conflict. myAdd sums two integers and adds one.

```
1 public int myAdd(int x, int y) {
2   int z = x + y + 1; // Modified in A
3   return z + 1;      // Modified in B
4 }
```

Listing 6.4: Version M in a semantic merge conflict. myAdd sums two integers and adds one, then adds one again.

6.3.4 Decompose the Conflict

When applying the previous approach to the program as a whole — which can be imagined as applying the approach to the single main method — we will almost always report a conflict. Such a semantic conflict lies in the nature of branching, since both branches are intended to make semantic changes to the program.

Based on the idea of locality of behaviour, we try to solve this by decomposing the program into its parts. This enables reporting a more meaningful insight into a semantic conflict, by reporting the number and the locations of violations for the parts.

To this end, we change our rules to operate on 4-tuples of sets. The elements in these sets will still use the definition of equality that is plugged into our approach. In the case of path conditions, this is the equivalency used in Section 6.3.3. The rules are adjusted as follows to work on a 4-tuple of sets:

CF-A This rule now comprises two parts. If either part is violated, there is a semantic merge conflict. The first part is

$$P_A \setminus P_O \subseteq P_M$$

which means that all elements added by A, which were not previously present

in O, need to be present in M. The second part is

$$(P_O \setminus P_A) \cap P_M = \emptyset$$

which means that none of the elements in O that were removed by A, may be present in M.

CF-B Analogous to CF-A, $P_B \setminus P_O \subseteq P_M$ and $(P_O \setminus P_B) \cap P_M = \emptyset$ must hold.

CF-AB If either

$$P_A \cap P_B \subseteq P_M$$

or

$$P_M \subseteq P_A \cup P_B$$

is violated, there is a conflict. The first equation means that all elements that are contained in both A and B need to be part of M. The second equation means that any found in M, must also have been in A or B.

For a single element set, the rules correspond to the rules we defined in Section 6.3.2. This set notation can be transferred into statements about which elements are missing or superfluous in the merge, giving their locations, and quantifying violations.

For completeness, we list the set operations with the adjusted equality between path conditions defined by a constraint solver, as we described in Section 6.3.3.

$$A \setminus B := \{a \in A \mid \nexists b \in B : b \Leftrightarrow a\} \quad \text{[Set Difference]}$$

$$A \cap B := \{a \in A \mid \exists b \in B : b \Leftrightarrow a\} \quad \text{[Set Intersection]}$$

$$A \subseteq B \text{ } \textit{iff} \text{ } \forall a \in A \, \exists b \in B : a \Leftrightarrow b \quad \text{[Subset]}$$

For our semantic approach to detecting conflicts, we partition the behaviour of the program. This is done by the partitioning of the input space of the program that is inherent to symbolic execution.

Consider the example in Listings 6.5 to 6.8 with control structures and more than one path condition given by a symbolic execution engine. The code includes two checks for a divide-by-zero error: first on line two and again on line five. Branch A and B do not agree on which check to remove to avoid this redundancy. Hence, the result is a missing check for the divide-by-zero error after merging, i.e., a semantic conflict.

Symbolic execution handles the control structures by branching the analysis and adding new path conditions to, for example, both the consequent and the alternative of an `if`. Thus, symbolic execution partitions the input space of the method under analysis. The 4-tuple of sets of path conditions for the program

```
1 int div(int x, int y) {
2   if (y == 0) // Removed in A
3     return 0; // Removed in A
4
5   if (y != 0) // Removed in B
6     return x / y;
7   else        // Removed in B
8     return 0; // Removed in B
9 }
```

Listing 6.5: The original version O has redundant safety checks, i.e., it checks for a division by zero twice: first on line 2 and then again on line 5. Branches A and B remove one check. However, they do not agree on which check is removed. In M, both checks are missing, so the safety check semantics are missing.

```
1 int div(int x, int y) {
2   if (y != 0)
3     return x / y;
4   else
5     return 0;
6 }
```

Listing 6.6: Branch A removes lines 2–4 from O.

```
1 int div(int x, int y) {
2   if (y == 0)
3     return 0;
4
5   return x / y;
6 }
```

Listing 6.7: Branch B removes lines 5, 7, and 8 from O.

```
1 int div(int x, int y) {
2   return x / y;
3 }
```

Listing 6.8: In version M, both checks are missing, so the safety check semantics are missing.

versions in Listings 6.5 to 6.8 looks as follows. As before, x, y, and r are symbolic variables for `x`, `y`, and the return value, respectively.

$$
\begin{aligned}
P_O &= \{y = 0 \land r = 0,\ y \neq 0 \land r = x/y\} \\
P_A &= \{y = 0 \land r = 0,\ y \neq 0 \land r = x/y\} \\
P_B &= \{y = 0 \land r = 0,\ y \neq 0 \land r = x/y\} \\
P_M &= \{r = x/y\}
\end{aligned}
$$

Applying the conflict freedom rules to these sets results in the following:

- **CF-A**: $P_A \setminus P_O = \varnothing$ and thus trivially $P_A \setminus P_O \subseteq P_M$. Similarly, $P_O \setminus P_A = \varnothing$, thus $(P_O \setminus P_A) \cap P_M = \varnothing$. CF-A is not violated.

- **CF-B**: Analogous to CF-A. CF-B is not violated.
- **CF-AB**: Here, however, $P_A = P_B = (P_A \cap P_B) \not\subseteq P_M$, so CF-AB is violated. Equally, $P_M \not\subseteq P_A \cup P_B$, which also causes a violation.

The lack of the path condition $y = 0 \land r = 0$ reflects that the check for a divide-by-zero error is missing in M. Our approach reports a semantic conflict.

In the following section, we will discuss the technical details of our prototype implementation of this approach to detecting semantic conflicts. In Section 6.5, we will evaluate our definition of a semantic conflict on synthetic and real-world examples. Some examples will be discussed in depth.

6.4 Technical Details

We implement a prototype of the approach that we need for the evaluation in Section 6.5. This section describes the technical details of the prototype and its limitations. We separate such details from the description of our approach, since they can be ignored on a conceptual level. However, they impose technical limitations on the evaluation that follows since that evaluation is based on the prototype.

Symbolic execution is technically challenging and strongly dependent on existing technology. We reuse a combination of Symbolic PathFinder [130], Gumtree [46], and Z3 [39]. We adapt Symbolic PathFinder and Gumtree as described in this section.

6.4.1 Aligning Variables in Path Conditions

The path conditions for each program version result from different runs of a symbolic execution engine. However, there is no clear relationship between symbolic variables created in the different runs. To show an equivalency between the path conditions, we create a mapping between the symbolic variables of path conditions in different program versions.

Consider the example of two path conditions, $x_O < 10$ and $x_A < 10$, resulting from an analysis of O and A. Variables x_O and x_A are symbolic and created during analysis. Without further information on the equivalence of variables, one cannot show the equivalence $x_O < 10 \Leftrightarrow x_A < 10$. An extra conjunct $x_O = x_A$ is needed.

We use a syntactical analysis as a preprocessing step in our approach that resolves this problem. The step finds matches between abstract syntax tree (AST) nodes across different program versions. When using the constraint solver to find an equivalence, our prototype uses this mapping to add equalities between symbolic variables, such as $x_O = x_A$. While $x_O < 10 \Leftrightarrow x_A < 10$ was not provable, $(x_O = x_A) \Rightarrow (x_O < 10 \Leftrightarrow x_A < 10)$ is.

Our prototype uses GumTree [46] for this step. GumTree is commonly used to compute edit scripts between two ASTs: a sequence of add, move, update, and delete operations that transform the first tree into the second tree.[4] As a side effect, GumTree finds a mapping between nodes of the two ASTs it is given. Our prototype runs GumTree on every combination of the ASTs of program version O, A, B, and M. Our prototype uses the resulting mappings to decide which conjuncts to add.

6.4.2 Symbolic Execution Engine

For the symbolic execution engine, we use Symbolic PathFinder (SPF) [130], an extension to the Java PathFinder (JPF) [63]. SPF is still used in recent research, for example in HyDiff [122].

SPF does have some shortcomings. At the time we started using SPF, it supported Java 8. New language features have been added to Java since then. SPF also does not support all Java constructs. It does not model the entire standard library and misses some language features, such as `try-catch`. Where required by SPF, we simplify the code under analysis.

As we described in Section 2.5.3, symbolic execution has limitations, such as path explosion, that lead to some paths not getting analysed. This negatively affects the conclusions drawn by our tool. Improving symbolic execution is outside the scope of this work. We minimise the impact of the limitations by decomposing the program into simpler parts (see Section 6.3.4).

Symbolic Variable for Output

To detect equivalent path conditions (previously denoted as $\Leftrightarrow$), we require constraints on the symbolic input and output variables of a program. SPF does not create constraints on the output, however. We modified SPF to create an extra symbolic variable for the output returned by a method.

Source Code Information

In Section 6.4.1, we motivated the need to add equalities between symbolic variables to show equivalence between path conditions from different program versions. We also discussed the extra syntactical information our prototype gathers by means of GumTree. To link that extra syntactical information to the path conditions produced by SPF, we also modified SPF so that it keeps track of additional source code information on its end.

Specifically, we need to link a symbolic variable back to its corresponding point in the source code. However, Symbolic PathFinder works on Java bytecode which

[4] Similar to ChangeNodes which we used in Chapter 4.

does not maintain such information by default. The analysed code instead needs to be compiled with a debug flag in order to preserve rudimentary source code information in the compiled `.class` files.[5] This flag encodes information regarding variables and at what line they came into scope. The SPF's bytecode parser parses the extra debug information when reading in a `.class` file. However, the information is immediately discarded and not propagated in the analysis, nor is it linked in any way to any of the symbolic variables. We modify SPF to intercept and adapt the creation of symbolic variables. Upon creation of a symbolic variable, our modified Symbolic PathFinder saves any line number and original name information.

6.4.3 Constraint Solver

As we discussed in Section 2.5, symbolic execution uses constraint solvers to decide on the satisfiability of path conditions and to obtain input from path conditions. We use the constraint solver to establish equivalency between path conditions (see Section 6.3.3). In our implementation, we use the Z3 constraint solver [39]. In this subsection, we describe how (1) we extend SPF's built-in translation of path conditions to queries, (2) we rewrite the way SPF and Z3 communicate, and (3) we ensure the Gumtree information is used in this step. After these adjustments, the queries are processable by Z3.

Passing constraints and path conditions to Z3, and to other constraint solvers, is already built into SPF. We extend the existing SPF implementation to fit our needs. We patched support for `implies` and `or`, which was missing, and support for `not`, which was limited.

We rewrite the way SPF communicates with Z3. Originally, SPF posted a constraint to the constraint solver as soon as the constraint was encountered and parsed. Instead, our modified SPF parses the internal constraint representation into Z3 constraints, but delays sending the constraint to the solver until our prototype needs to check satisfiability of the equivalence between two path conditions. This was necessary because immediately posting the constraints to Z3 created unwanted conjuncts between different constraints, e.g., an $\wedge$ when we wanted an $\vee$.

To check for equivalency between path conditions, our tool aligns the links discovered by Gumtree in the AST with the source code information of the symbolic variables in SPF. Note that some discrepancies between line numbers of both can occur due to the Java compilation process. Our tool applies a heuristic looking for lines that are close enough to one another, where trial-and-error led to a threshold of three lines.

[5] In Maven, for example, this is done by passing the `-Dmaven.compiler.debug=true` flag.

```
1  #1: O safe
2  O
3  pc:
4  return: (y_2_SYMINT + x_1_SYMINT)
5
6  ---
7  #2: AB unsafe
8  A
9  pc:
10 return: ((y_4_SYMINT + x_3_SYMINT) + CONST_1)
11
12 B
13 pc:
14 return: ((y_6_SYMINT + x_5_SYMINT) + CONST_1)
15
16 ---
17 #3: M unsafe
18 M
19 pc:
20 return: (((y_8_SYMINT + x_7_SYMINT) + CONST_1) + CONST_1)
```

Listing 6.9: Output of our prototype when run on the `myAdd` example in Listings 6.1 to 6.4 in Section 6.3.3.

6.4.4 Prototype

Our prototype combines the previous technical aspects. It performs a syntactical analysis of O, A, B, and M. It ensures that each version is symbolically executed. It finds equivalent path constraints. Finally, it checks whether any violations of CF-A, CF-B, or CF-AB occur. We list a high-level overview in Algorithms 2 and 3 where the result is a set of path conditions violating our rules.

Results are shown by listing path conditions, their equivalencies, and flagging those that break CF-A, CF-B, or CF-AB. A further improvement would be to link these path conditions more explicitly back to the relevant parts of the code, for example, by using the source code information our prototype already keeps track of.

In Listing 6.9, we provide the output of our prototype when run against the `myAdd` example depicted in Listings 6.1 to 6.4 in Section 6.3.3. Results are shown by grouping paths that are equivalent to one another. In this example, that gives three such groups: P_O, $\{P_A, P_B\}$, and P_M. These are shown in Listing 6.9 at `#1: O Safe`, `#2: AB unsafe`, and `#3: M unsafe`, respectively. For each group, an assessment of that combination of equivalent paths is shown. In this case, the $\{P_A, P_B\}$ and P_M groups are marked unsafe as they violate CF-A, CF-B, and CF-AB. For each group, the path conditions for every version in that group are listed. The path conditions are split up in regular path conditions and an expression indic-

```
1  Function findConflictingPaths(merge: CommitId): Set[(PC, PC, PC, PC)] is
2      O, A, B, M ← fourPartsOfMerge(merge);
3      pcsO ← getPathConditionsFromSymbolicExecution(O);
4      ... // Do the same for A, B, and M
5      OtoA ← findVariableMapping(O, A);
6      ... // Repeat for O-B, O-M, A-B, A-M, B-M
7      crossVersionPaths ← ∅;
8      foreach o ∈ pcsO do
9          // Each of a, b, and m can be null
10         a ← findAndRemoveEquivalentPc(o, pcsA, OtoA);
11         b ← findAndRemoveEquivalentPc(o, pcsB, OtoB);
12         m ← findAndRemoveEquivalentPc(o, pcsM, OtoM);
13         crossVersionPaths ← crossVersionPaths ∪ (o, a, b, m);
14     end
15     foreach a ∈ pcsA do
16         b ← findAndRemoveEquivalentPc(a, pcsB, AtoB);
17         m ← findAndRemoveEquivalentPc(a, pcsM, AtoM);
18         crossVersionPaths ← crossVersionPaths ∪ (null, a, b, m);
19     end
20     foreach b ∈ pcsB do
21         m ← findAndRemoveEquivalentPc(b, pcsM, BtoM);
22         crossVersionPaths ← crossVersionPaths ∪ (null, null, b, m);
23     end
24     foreach m ∈ pcsM do
25         crossVersionPaths ← crossVersionPaths ∪ (null, null, null, m);
26     end
27     return crossVersionPaths.filter(isViolatingCombination);
28 end
```

Algorithm 2: High level overview of the steps taken to get from a merge commit SHA-1 hash to a set of violating path conditions. The definition of *isViolatingCombination* can be found in Algorithm 3.

```
1  Function isViolatingCombination(cvp: (PC, PC, PC, PC)): Boolean is
2      isPresent ← cvp.map(isNotNull);
3      // This function checks every combination
4      switch isPresent do
5          // Path is in every version
6          case (true, true, true, true) do return true;
7          // Path removed in B, still gone in M
8          case (true, true, false, false) do return true;
9          // Path removed in A, still gone in M
10         case (true, false, true, false) do return true;
11         // Path removed in A and B, still gone in M
12         case (true, false, false, false) do return true;
13         // Path added in A and B, present in M
14         case (false, true, true, true) do return true;
15         // Path added in A, present in M
16         case (false, true, false, true) do return true;
17         // Path added in B, present in M
18         case (false, false, true, true) do return true;
19         // ––
20         // Path in O, A, and B, not present in M
21         case (true, true, true, false) do return false;
22         // Path removed in B, returns in M
23         case (true, true, false, true) do return false;
24         // Path removed in A, returns in M
25         case (true, false, true, true) do return false;
26         // Path removed in A and B, returns in M
27         case (true, false, false, true) do return false;
28         // Path added in A and B, disappears in M
29         case (false, true, true, false) do return false;
30         // Path added in A, disappears in M
31         case (false, true, false, false) do return false;
32         // Path added in B, disappears in M
33         case (false, false, true, false) do return false;
34         // Path does not exist in O, A, or B, appears in M
35         case (false, false, false, true) do return false;
36         // Extra case to be exhaustive, should not happen
37         case (false, false, false, false) do return false;
38     end
39 end
```

Algorithm 3: Annotated implementation of *isViolatingCombination* called in Algorithm 2.

ating the special return variable. In this example, there is only an expression for the return variable. These are shown with symbolic variables whose names help a developer determine where a certain value comes from. Here there are eight symbolic variables, each starting with an origin name, a number indicating the order in which the symbolic variable was created, and a specification of the type of the symbolic variable. Because of the mapping between symbolic variables that is performed, A and B were deemed equivalent despite the symbolic variables `y_4_SYMINT` and `x_3_SYMINT` having no symbolic relation to `y_6_SYMINT` and `x_5_SYMINT` prior to the mapping. This mapping was also performed between, e.g., A, `y_4_SYMINT` and `x_3_SYMINT`, on the one hand and B, `y_8_SYMINT` and `x_7_SYMINT`, on the other hand, but this does not lead to equivalent path conditions due to the difference in `CONST_1`s being added.

We also run our prototype against the `div` example depicted in Listings 6.5 to 6.8 in Section 6.3.4. We modify the output slightly for depiction here because SPF has extra built-in checks for its path conditions when it comes to division by zero. This built-in check does not change the assessment of the merge by our prototype, but it does make the output by the prototype more confusing when shown unmodified. The result is shown in Listing 6.10. Recall that while the path condition and return expression are shown separately, they are combined prior to checking equivalence with a path condition of another version. Here, `#1: OAB unsafe` represents the path in O, A, and B when $y = 0$, leading to the program returning 0, `#2: OAB unsafe` represents the path in O, A, and B when $y \neq 0$, leading to the program returning x/y, and finally `#3: M unsafe` is the one path in M which returns x/y without any checks. All three are marked unsafe as they violate CF-AB.

Our prototype requires a user to explicitly state the method and inputs of the program that need to be analysed symbolically. We also need to inject a `main` method into the code that is subject to analysis. This custom `main` guides symbolic execution to the relevant part of the code. This is relevant to the decomposition of the program. We assume that future work can automate and resolve these limitations of our prototype.

We avoid an evaluation of the time our approach needs because it is very sensitive to the technical details listed in this section. Due to our decomposition of the problem (Section 6.3.4), possible simplifications (Section 6.4.2), and addition of a `main` method, the time taken for the actual analysis in our experiments is short (typically under 10 seconds). However, we do not know how this performance generalises to more realistic cases.

6.4.5 Future Automation

We envision our prototype to warn developers of potential problems in merges and pull requests when working with standard version control systems. How-

```
1  #1: OAB unsafe
2  O
3  pc: y_2_SYMINT = CONST_0
4  return: 0
5
6  A
7  pc: y_4_SYMINT = CONST_0
8  return: 0
9
10 B
11 pc: y_6_SYMINT = CONST_0
12 return: 0
13
14 ---
15 #2: OAB unsafe
16 O
17 pc: y_2_SYMINT != CONST_0
18 return: (x_1_SYMINT / y_2_SYMINT)
19
20 A
21 pc: y_4_SYMINT != CONST_0
22 return: (x_3_SYMINT / y_4_SYMINT)
23
24 B
25 pc: y_6_SYMINT != CONST_0
26 return: (x_5_SYMINT / y_6_SYMINT)
27
28 ---
29 #3: M unsafe
30 M
31 pc:
32 return: (x_7_SYMINT / y_8_SYMINT)
```

Listing 6.10: Slightly modified output of our prototype when run on the `div` example in Listings 6.5 to 6.8 in Section 6.3.4.

ever, applying our current prototype still requires high manual effort.

We envision the following more automated sketch of a workflow to remove most manual effort:

- Configuring the build and compilation of the program will remain manual work. It can be set up once for a repository if the build process does not change.
- From this point on, a merge can automatically trigger the following steps.
- We can produce possible decompositions of the merged program. This step may consider dependencies between code and use program slicing [115]. The decomposition needs to be matched over the involved program versions of a merge.
- Code can be generated with a main method that calls one or all methods in a component.
- Conflicts can be identified by our approach and warnings are reported.

6.5 Evaluation

We evaluate our approach on synthetic and empirical data. The synthetic data is used to show the technical validity of our prototype. We use data generated by mutation testing. The empirical data is used to show the empirical relevance of our approach. This data is gathered from GitHub. The data can be found online.[6]

6.5.1 Technical Validation

This section evaluates the technical validity of our prototype on synthetic data.

Method

We generate a dataset using software mutation testing [41]. The original program version O is defined to be a Java program. We run the mutation testing tool Major [80] on the program to generate a set of mutations $\mathcal{U}$ of O. We form the Cartesian product of set $\mathcal{U}$ with itself, and define branches A and B accordingly. Finally, we use Git's default merge to produce the merge commit M.

Most program versions O, A and B merge without problems. However, Git's textual merge potentially produces semantically conflicting program versions M. Our approach should detect such semantic merge conflicts.

[6] `https://github.com/ward/semantic-merge-conflicts-scam2023`

```
1  SIM <- 20000
2  hits <- NULL
3
4  # Our true lables identified by manual analysis.
5  Ytrue <- c(1,1,1,0,0,0,0,0,0,0)
6
7  for(i in 1:SIM){
8
9    # Classify conflict randomly.
10   Y <- rbinom(10, size = 1, prob=0.3)
11
12   hit <- sum(Y == Ytrue)
13   hits <- cbind(hits, hit)
14 }
15
16 hist(hits, breaks = 10, xlab = "Number␣of␣correct␣lables", main = "Histogram␣of
      ␣hits")
17
18 table(hits)
```

Listing 6.11: Small simulation showing the probabilities of correctly labelling ten merges.

For the resulting 4-tuples (O, A, B, M) that we have generated, we manually classified the merge M as a semantic merge conflict or not. We run our prototype and compare the output to this manually tagged baseline.

Results

For the input program O, we use code from *Project Euler*, a website centred around mathematical programming challenges [45].

We generate a set $\mathcal{U}$ with 34 different mutants of the original program. The Cartesian product results in 561 combinations for pairs of A and B with corresponding merge M. We exclude symmetric pairs. We also exclude 123 pairs (22%) that are reported as a trivial textual conflict by Git's default merge. We randomly sample ten pairs from the remaining 438 (78%) for a manual analysis.

We manually classify three of the corresponding merges as semantic conflicts and seven as valid. Our tool classifies all merges correctly.

The chance of labelling ten merges correctly using a random classifier is smaller than 0.2%. We refer to Listing 6.11, a small simulation showing this.

Threats to Validity

The low complexity of the input program is a threat to validity for this technical evaluation. We used an existing program instead of creating an input program

as part of this evaluation. This makes the results more realistic. However, the technical limitations, described in Section 6.4, constrain the selection of our input program.

Our manual classification of semantic conflicts may be influenced by our understanding of our approach. This threat is hard to mitigate. We deploy our dataset online to enable the replication and revision of our approach.

The mutation generation is limited in that every mutation introduces exactly one change to the program. However, both changes affect the same parts of the program which is still a challenging situation.

6.5.2 Empirical Validation

The empirical evaluation of our approach is challenging because real semantic merge conflicts are relatively rare. There is no standard dataset that can be used for benchmarking. Our empirical evaluation is therefore split into two parts.

1. In a first part, we aim to classify semantic merge conflicts *retroactively*. We use heuristics on the revision history following a merge. We do this to get promising merge candidates that are interesting for an in-depth discussion. We need this alternative to a regular sample from GitHub, since semantic merge conflicts are relatively rare. Manually tagging a rare class on a regular sample is unrealistic due to the small number of positive (or negative) cases.

2. In a second part, we apply our tool to such candidates and check if we can identify the conflicts *proactively* using our approach. This is done without having the subsequent commit history that indicates a conflict. We discuss these cases in-depth in Table 6.1.

We structure this part of the evaluation accordingly.

Retroactive Method

We use a dataset of Java projects that use Maven as their build tool, introduced by Cavalcanti et al. [27]. The dataset lists merges, their parents, and metadata on the build and test success for the merge commits.[7] However, Cavalcanti et al. study syntactic merge conflicts by comparing structured and semi-structured approaches to merging. Hence, this dataset does not immediately work for the

[7]The dataset did not contain build and test outcomes for the parents of the merge commits. As Cavalcanti et al. [27] note, and as we discussed in our study in Chapter 5, this is not enough information to confirm whether a conflict caused a failure or whether the prior commits already had failing builds.

evaluation of our approach. We use the same projects as Cavalcanti et al., but do not use their list of merges.

For the Java projects, we collect all merges M, the corresponding parents A, B, and the first common ancestor O. We ignored rare situations where there are more than two parent commits. We also ignored trivial merges where A is a parent of B, or vice versa (see our discussion accompanying Figure 2.7 in Section 2.3.3). To identify O, we use Git's built-in common base finding algorithm.

We apply the following heuristics to filter for interesting candidates that can potentially be classified as a semantic merge conflict.

- We filter for *merges with an overlap* in the modifications. We apply stratification (or group-by) to diversify the commits we consider. For one half of the merges, we require that versions A and B change the same file. For the other half, we require that the same line was changed.

- We filter for merges with a suspicious commit history following the merge. We apply a light-weight version of the SZZ algorithm [144] (also used by Mockus and Votta [110] and Ray et al. [135]) to *determine whether the commit directly following the merge is a bug fixing commit*. Notably, this light-weight approach does not require a well-used issue tracker to identify bug fixing commits. Instead, bug fixes are identified by means of containing certain strings in their commit message, e.g., `fix` or `bug`.

  We do not go for the alternative of considering bug fixing commits further down in the tree of descendants. In previous experiments, we noticed that this introduces noise both due to an overlap between merges and due to a general increase in the number of fixes unrelated to the merge commit.

- The bug fix following the merge does not always relate to a bug introduced in the merge. Hence, the changes by the *fix and the merge need to overlap*, too. We filter for bug fixing commits that change at least one line or file (see earlier discussion on stratifications) of the lines or files changed between versions O and M.

The heuristics are relevant to reduce the candidates subject to our subsequent manual tagging of conflicts. They produce a pool of candidates with a realistic chance of being a semantic merge conflict. Executing a manual analysis on a random sample of real-world merges on GitHub would be unrealistic. Our method corresponds to a standard practice in evaluating information retrieval systems, referred to as *relevance judgement* or *pooling method* (see Spärck Jones and Rijsbergen [146, page 13] or more recently Baeza-Yates and Ribeiro-Neto [10, page 158]).

Real vs.			**Violating PCs**	
Synthetic	**Project**	**Merge**	**at merge**	**after fix**
R	google/j2objc [59]	79781f8	6 / 6	4 / 6
R	tcurdt/jdeb [34]	e9ceff5	3 / 15	3 / 15
R	welovecoding/ editorconfig-netbeans [168]	99578c4	2 / 11	1 / 11
R	larsga/Duke [54]	7c65f5e	—	—
R	spotify-web-api-java/ spotify-web-api-java [147]	675a0d2	—	—
R/S	google/j2objc	79781f8'	2 / 2	0 / 2
S	—	—	0	not 0

Table 6.1: Overview of the cases studied in RQ2. The first five cases are extracted from real-world projects by means of a manual examination following our retroactive semantic merge conflict identification. Case six is a synthetic adjustment of one such real-world case. Case seven is a synthetic case added to indicate a limitation in our approach.

Retroactive Results

We manually examined 500 merges in 152 projects identified by our retroactive method. We identified 55 semantic merge conflicts (11% of manually examined merges). Of these, 50 were caught by the compiler (91% of semantic merge conflicts, 10% of manually examined merges) and include cases such as `imports` getting out of sync, a method with the same signature being added twice, and a variable rename in version A while version B added a usage of the old variable name. The five remaining semantic merge conflicts are the interesting ones for our approach (9% of semantic merge conflicts, 1% of manually examined merges). These five conflicts can be found in Table 6.1. The two rightmost columns are part of our proactive evaluation. The project name is as it appears in the original dataset by Cavalcanti et al. [27] and corresponds to the `username/projectname` pattern used by Github. The two final rows in Table 6.1 represent two synthetic cases we use to expand the discussion in our proactive evaluation.

All five semantic merge conflicts were identified by the heuristic of overlapping changes to files, not overlapping changes to lines (see the stratification discussed in the retroactive method above). All five conflicts include a bug fixing commit message using the keyword `fix`. Two of the messages also use the keyword `merge`.

Proactive Method

In the following part, we evaluate our proactive approach using symbolic execution empirically by running it on the pool of candidates that we have identified in the previous part of the evaluation.

To get a more exhaustive discussion, we complement the candidates by (1) one adaptation that simplifies the control flow and (2) another synthetic case where our approach does not work as expected.

Furthermore, we also apply the prototype implementation of our approach on another five merge commits that are *not* followed by a fix, i.e., merge commits that did not cause a conflict. We chose them randomly and manually verified there was no (obvious to us) conflict present. They are not included in the table that follows, but we do discuss the results for these merge commits in the text.

We first provide an overview of the results and then discuss each semantic merge conflict case in the table in more detail.

Proactive Results

The results are described in Table 6.1. The table only includes the merges that are followed by a fix and that we have manually tagged as a conflict. The online dataset contains all candidates. We structure the table as follows:

- The first column (Real vs. Synthetic) indicates a discussion of a real-world or synthetic case. The majority of the cases are real. Two synthetic cases are added to make the discussion exhaustive.
- The second column (Project) reports on the GitHub project name in the format `username/projectname`.
- The third column (Merge) lists the abbreviated SHA-1 of the merge.
- Column four and five (Violating PCs at the merge or after the fix) give the actual classification in terms of number of violating path conditions. This number is either given for the merge or for after the fix. Eventually, we expect this number to decrease after a fix if our approach works correctly. The number of violations can be derived from our merge conflict definition in terms of set semantics (see 6.3.4).

We sum up the insights to be gained from the table. Our approach using symbolic execution correctly detects the decrease in number of violations from merge commit to bug fixing commit for three of the merges. For one of those three merges, the violations disappear entirely in the bug fixing commit. For one merge, our approach reports no change in the number of violating path conditions when comparing merge commit to bug fixing commit. In one synthetic case, the number of violating path conditions actually increases once the bug is fixed. Two merges cannot be processed due to technical limitations, see the description that follows for more information.

For the five merge commits that did not have a semantic merge conflict, i.e., there was no bug fix commit following the merge conflict and we did not discover anything in a manual validation, the prototype implementation of our approach produces no warning in four cases. In one of the five, our prototype incorrectly detects a semantic merge conflict. These five merge commits are not listed in the table.

Summary This shows that in specific cases, our approach using symbolic execution is a promising complement to existing techniques for detecting merge conflicts. However, there are some cases for which the approach does not work as expected.

Proactive Results — Details

We describe each of the cases in Table 6.1 in further detail. The order here follows the order of the rows in Table 6.1.

1. In our first case, the branches A and B add the same behaviour. In branch A, the behaviour is added through a method call, protected by a boolean flag. In branch B, the behaviour is directly added to the method's body and not protected by a boolean flag. The merge includes both modifications, which is a clear semantic conflict. In the bug fix commit, the code added in branch A is removed. However, the code of branch A includes a boolean flag, but the fix does not. Hence, our prototype still reports on some violations after the fix, but fewer violations compared to the merge M. This real-world example is comparable to our example in Listings 6.1 to 6.4 which we discuss in Section 6.3.3.

2. In the second case, branch A and B move code around and make some other changes. The result is code duplication in the merge M which renders the code added by branch A unreachable. The fix removes the unreachable code. Our prototype tool warns about the behaviour from branch A disappearing for the merge M due to it being unreachable. However, after the fix,

the behaviour is still missing, so the number of violations does not change; our tool still warns about behaviour from branch A disappearing. This is a case where our prototype does not work as expected. This situation can be worth exploring in future work.

3. In the third case, a bug is present in the origin O. In branch A, the method that contains the bug is fixed. In branch B, the call to the method is commented out. In merge M, the method was thus fixed (due to A), but not called (due to B). The bug fixing commit uncomments the method call. Our tool reports that the changes from A disappear in M. In the fixed version, the tool reports that the effect of B's commenting out of code has disappeared. The fix decreases the number of reported violating path conditions.

4. In case four, the changes causing a semantic merge conflict involve a `try-catch` and the exception being raised within it. Our prototype cannot analyse this case due to the technical limitations of Symbolic PathFinder, which does not handle `try-catch` statements. We see no obvious way to simplify this case into something that can be analysed in the context of our prototype.

   We assume that our approach can spot such a conflict if `try-catch` statements were to be supported by Symbolic PathFinder; we do not see an inherent theoretical limitation for this in our approach.

5. In case five, branch A modifies the method's return type. In branch B, a method with the same name is added, overloading the old method by adding a new parameter. The return type of this new method in branch B is the same return type as in the original method before the change of branch A. In merge M, both methods are present: (1) the method from A with the new return type and (2) the method from B with the old return type and the added parameter. The fix updates the return type of the method of branch B to match the return type of branch A.

   This conflict is something our approach is unable to spot. There are no path conditions to compare and calling it a bug relies on guessing whether the developer intended to have different return types or not.

6. Case six is an adaptation of merge 79781f8 from google/j2objc, the first case we discussed. The boolean flag that was added in version A of the original case, is not added here. Instead, the added method is always called. Thus, A and B behave entirely the same, while the behaviour is duplicated in M. The fix removes the method call from A. The fix behaves exactly like A and B. We detect no more violations after the fix.

7. Case seven is synthetic. Consider branch A and B, which add a `parse` and a `sanitise` function, respectively. The plan of the developers is that, once merged, `parse` will make use of `sanitise` to clean up its input. When merging, however, this call is not added to `parse`. Our tool reports no violations. In the fix, `parse` does call `sanitise`. Our tool reports a conflict: the behaviour of `parse` changes in M (the "fix" here) compared to its behaviour in branch A. We add this synthetic case to our dataset for our evaluation to show the limitation of our prototype.

Threats to Validity

Comparable to the technical part of this evaluation section, our manual classification of merge conflicts may be biased by our expectations. We mitigate this by the explicit discussion when describing our proactive results. We also deploy our dataset online[8] to allow replication and revisions of our approach.

Different to the technical part of the evaluation that uses mutations, we cannot make any objective statements on the developer's original intention in the real-world cases. In our evaluation, we try to mitigate this problem by manually inspecting all involved commits and the commit messages in depth. In real-world usage, we believe that a developer will easily be able to judge warnings by our approach due to their familiarity with the code.

While we examined 500 merges, the low size of positive cases is another threat to this analysis and illustrates the rare nature of semantic merge conflicts. We are not computing any confidence intervals for which such sample size matters. We try to mitigate the problem of a low number of positives by favouring a qualitative and exhaustive discussion of real cases, including synthetic cases that logically follow from the real cases we have spotted.

6.6 Conclusion

In this chapter, we developed an approach that detects semantic merge conflicts by means of symbolic execution. We defined the program semantics as path conditions, as produced by a symbolic execution engine, and checked whether the conditions satisfy established rules that indicate a merge conflict. Our usage of symbolic execution to check these rules is novel. We implemented a prototype for the evaluation of our approach.

We evaluated the technical validity of our prototype using synthetic data. We generated the synthetic data through mutation testing.

We evaluated our approach empirically by following our research questions:

[8] `https://github.com/ward/semantic-merge-conflicts-scam2023`

RQ1 We needed empirical data from GitHub to show the empirical relevance of our approach, but semantic merge conflicts are relatively rare and hard to spot manually. Hence, we defined a retroactive method to perform an approximate classification of semantic merge conflicts that scales. The method is based on the commit history following the merge. We used the method to compute a pool of 500 potential semantic merge conflicts, which we examined manually and used for answering RQ2.

RQ2 We used the pool of manually classified semantic merge conflicts to evaluate our approach to detect semantic merge conflicts by symbolic execution. We discussed the application of our prototype qualitatively for five cases from our pool of real-world semantic merge conflicts.

Our approach using symbolic execution correctly detects changes in semantic violations in three out of five semantic merge conflicts. The evaluation showed in specific cases, our approach using symbolic execution is a promising extension to existing mechanisms to merge conflict detection.

Our prototype is limited by the technical constraints of the symbolic execution engine we used. Future work needs to focus on automation and the integration of our approach into the continuous integration pipeline. We also consider the combination of symbolic execution and data-driven approaches, such as deep learning, to be promising future work.

Chapter 7

Conclusion

At the beginning of this dissertation, we discussed the need for automated techniques for problematic commits in version control software. In particular, we focused on two types of problematic commits: (1) composite commits, which contain changes belonging to different unrelated tasks, and (2) merge commits, which may give rise to semantic merge conflicts due to an unintended interaction in the behaviour of the two branches being merged together.

We performed a large-scale study into the prevalence of syntactic and semantic merge conflicts. We found that half of the 348 analysed projects deal with a breaking merge commit at least once every 43 merge commits. For a quarter of projects, this is at least once every 17 merge commits. We found that breaking merge commits occur less often than breaking regular commits and are often repaired the same day with fewer than ten lines of code.

We proposed an automated approach to identify and reduce the two aforementioned types of problematic commits. Our first approach is data flow driven and comprises an algorithm for untangling composite commits. Our second approach is control flow driven and comprises an algorithm for detecting semantic merge conflicts by looking for potentially undesired changes in execution paths.

We evaluated both approaches by means of a prototype that we developed. To evaluate our commit untangling approach, we first analysed and further refined an established dataset of composite commits. Our technique identifies composite commits, but creates more fine-grained clusters of changes than expected. Each cluster of changes does stay within one single task. To evaluate our approach to detect semantic merge conflicts, we gathered our own dataset. We defined and evaluated a retroactive method to identify semantic merge conflicts based on the commit history following a merge commit. We computed a pool of 500 potential semantic merge conflicts and examined them manually. We evaluated our approach using the manually classified semantic merge conflicts. We discussed our prototype qualitatively through a case study in which we applied the prototype to five real-world semantic merge conflicts. Our approach detected semantic conflicts in three of the five cases. The evaluation showed that our approach is a promising complement to existing techniques for detecting merge conflicts.

7.1 Revisiting the Contributions

Composite Commit Untangling

We proposed an automated approach for untangling composite commits. The approach considers fine-grained changes made to an abstract syntax tree. It connects each change to a node in a program dependence graph. Our approach then slices around the nodes in the program dependence graph. Based on an overlap in the produced slices, our approach decides which changes belong together. We applied this technique in two ways: (1) we used it to decide whether a commit is composite or single-task and (2) we used it to untangle composite commits into clusters of related changes. We evaluated this technique on a dataset of composite commits (see the following contribution) and found that it identifies single-task and composite commits. We also found that the technique creates more fine-grained clusters of changes than the tasks in a composite commit. A cluster of changes does generally stay within its task.

Composite Commit Dataset Refinement

To evaluate the above approach, we started from a well-established dataset of composite commits originating from five Java projects. We performed an automated cleaning step of the dataset. Afterwards, we performed a manual verification of the commits and whether they were correctly classified. The resulting dataset forms a contribution on its own as it provides researchers with a refined set of composite commits to evaluate techniques on.

Empirical Study into the Prevalence of Merge Conflicts

We performed a large-scale study into the prevalence of syntactic and semantic merge conflicts. Combining information from Github and Travis CI, we identified merge commits and their build status in a continuous integration service. We found that half of the 348 analysed projects deal with a breaking merge commit at least once every 43 merge commits. For a quarter of projects, this occurs at least once every 17 merge commits. We found breaking merge commits to occur less often than breaking regular commits. Repairing them was often done the same day and by changing fewer than ten lines of code. The effort was concentrated in the source code as opposed to test code or other project files.

Semantic Merge Conflict Detection

We proposed an automated approach for detecting semantic merge conflicts using symbolic execution. We defined the program semantic in terms of the path conditions as produced by a symbolic execution engine. Our technique uses the path conditions from the different versions in a merge

and checks whether they satisfy established rules that recognise the presence of a merge conflict. We evaluated the technical validity of our technique through synthetic data. We evaluated the technique empirically by means of a case study of five real-world semantic merge conflicts. We found that in specific cases our approach is a promising extension to existing mechanisms to merge conflict detection.

Retroactive History-Based Semantic Merge Conflict Detection
: We defined a heuristics-based retroactive method to perform an approximate and scalable classification of semantic merge conflicts. This retroactive method is based on the commit history following a merge. We used the method to compute a pool of 500 potential semantic merge conflicts, which we manually examined and categorised.

Prototype Implementation of Untangling and Detection Algorithms
: We developed a prototype implementing our composite commit untangling technique and another prototype implementing our semantic merge conflict detection technique. Both are written in Java and work on Java code. The prototypes are used in the evaluation of both techniques. The prototypes are open source and made available at [113].

7.2 Limitations and Future Work

We discuss some of the limitations we encountered in our approaches and consider future avenues of work.

For an industrial setting, our techniques need to perform their analysis reasonably fast. Our composite commit untangling prototype only took a few seconds for the majority of analysed commits. However, several larger commits took over a minute to analyse. The time taken for the actual analysis of our semantic merge conflict detection prototype in our experiments is short (typically under ten seconds). However, this can at least partially attributed to the technical details and setup we described. We do not know how the observed performance in this controlled setting generalises to more realistic cases, but we are aware that symbolic execution time can easily grow exponentially. Optimising the code of both research prototypes is likely to be in order, but requires engineering effort.

Equally important for an industrial setting is a certain amount of automation. For our semantic merge conflict detection algorithm, this is not currently the case. As we have described, we still perform various manual setup tasks. We believe these can be automated, but the engineering effort first has to be made.

We built our prototypes on top of other tools that come with their own limitations. Both TinyPDG and Symbolic PathFinder cannot handle some types of Java

statements, such as a `try-catch`. This makes our prototypes unusable in an industrial setting. As with time optimisations and automation, supporting more of the Java language or its standard library requires a significant engineering effort. This is outside the scope of this dissertation.

Separate from technical limitations in the tools we built on top of, our prototypes cannot handle some cases. Some of these limitations may be inherent to our approaches. Future work should aim to delineate these limitations exactly, providing an opportunity to conceive different approaches for those cases. For our untangling technique, this includes ideas such as considering different clustering criteria for the fine-grained changes or experimenting with different slicing techniques. In our semantic merge conflict detection technique, we could look into defining a more relaxed equivalency between path conditions or adjusting the merge conflict rules we check against. Alternatively, it is worth exploring how to combine our approaches with related work and identifying any that prove to complement them.

We believe there is also an opportunity to explore a combination of both our techniques to improve semantic merge conflict detection. The underlying idea here is as follows. A merge combines the features, bug fixes and refactorings from two branches. A composite commit too is a combination of different features, bug fixes and refactorings. Our untangling algorithm might be used as a first step, to identify parts of the merge that potentially interact. Our semantic merge conflict detection technique can then run on the resulting clusters across different versions. This could help point the heavier symbolic execution analysis in the direction of smaller parts of the code by first running the more light-weight untangling algorithm.

7.3 Closing Remarks

In this dissertation, we focused on a backbone of tools in a contemporary developer's toolbox: version control software. Specifically, we looked into two types of problematic commits: composite commits and merge commits. While following proper practices can mitigate many of the issues associated with both types, in reality developers do not necessarily know about these or are unwilling to spend the time and effort upfront to avoid them. We performed a study into the prevalence of syntactic and semantic merge conflicts at a large scale. We explored data flow and control flow driven approaches to untangle and detect the problematic commits, respectively. We proposed and evaluated a commit untangling technique based on a combination of fine-grained changes and slicing in program dependence graphs. We proposed and evaluated a semantic merge conflict detection technique which combines path conditions, produced by a symbolic execution engine, across the different program versions in a merge and validates

them against certain rules that indicate a merge conflict. We have shown that both approaches can be an addition to a developer's setup and can mitigate further problems down the line.

Bibliography

[1] Paola R. G. Accioly, Paulo Borba and Guilherme Cavalcanti. 'Understanding semi-structured merge conflict characteristics in open-source Java projects'. In: *Empirical Software Engineering* 23.4 (2018), pp. 2051–2085. DOI: 10.1007/s10664-017-9586-1.

[2] Luis Amaral, Marcos C. Oliveira, Welder Luz, Jose Fortes, Rodrigo Bonifacio, Daniel Alencar, Eduardo Monteiro, Gustavo Pinto and David Lo. 'How (Not) to Find Bugs: The Interplay Between Merge Conflicts, Co-Changes, and Bugs'. In: *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2020. DOI: 10.1109/icsme46990.2020.00049.

[3] Glenn Ammons, Rastislav Bodík and James R. Larus. 'Mining Specifications'. In: *Symposium on Principles of Programming Languages (POPL)*. Ed. by John Launchbury and John C. Mitchell. Association for Computing Machinery (ACM), Jan. 2002, pp. 4–16. DOI: 10.1145/565816.503275.

[4] Sven Apel, Olaf Leßenich and Christian Lengauer. 'Structured merge with auto-tuning: balancing precision and performance'. In: *IEEE/ACM International Conference on Automated Software Engineering, (ASE)*. Ed. by Michael Goedicke, Tim Menzies and Motoshi Saeki. ACM, 2012, pp. 120–129. DOI: 10.1145/2351676.2351694.

[5] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer and Christian Kästner. 'Semistructured merge: rethinking merge in revision control systems'. In: *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) and 13th European Software Engineering Conference (ESEC)*. Ed. by Tibor Gyimóthy and Andreas Zeller. ACM, 2011, pp. 190–200. DOI: 10.1145/2025113.2025141.

[6] Brad Appleton, Stephen P. Berczuk, Ralph Cabrera and Robert Orenstein. 'Streamed Lines: Branching Patterns for Parallel Software Development'. In: *Pattern Languages of Programs Conference (PLoP)*. 1998.

[7] Ryo Arima, Yoshiki Higo and Shinji Kusumoto. 'A Study on Inappropriately Partitioned Commits — How Much and What Kinds of IP Commits in Java Projects?' In: *International Conference on Mining Software Repositories (MSR)*. Ed. by Andy Zaidman, Yasutaka Kamei and Emily Hill. ACM, May 2018, pp. 336–340. DOI: 10.1145/3196398.3196406.

[8] Dimitar Asenov, Balz Guenat, Peter Müller and Martin Otth. 'Precise Version Control of Trees with Line-Based Version Control Systems'. In: *International Conference on Fundamental Approaches to Software Engineering (FASE)*. Ed. by Marieke Huisman and Julia Rubin. Vol. 10202. Lecture Notes in Computer Science. Springer, 2017, pp. 152–169. DOI: 10.1007/978-3-662-54494-5_9.

[9] Alberto Bacchelli and Christian Bird. 'Expectations, Outcomes, and Challenges of Modern Code Review'. In: *International Conference on Software Engineering (ICSE)*. Ed. by David Notkin, Betty H. C. Cheng and Klaus Pohl. IEEE Computer Society, 2013, pp. 712–721. DOI: 10.1109/ICSE.2013.6606617.

[10] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern information retrieval*. ACM Press, 1999, p. 513. ISBN: 0-201-39829-X.

[11] Mike Barnett, Christian Bird, João Brunet and Shuvendu K. Lahiri. 'Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Antonia Bertolino, Gerardo Canfora and Sebastian G. Elbaum. IEEE Computer Society, 2015, pp. 134–144. DOI: 10.1109/ICSE.2015.35.

[12] Olga Baysal, Oleksii Kononenko, Reid Holmes and Michael W. Godfrey. 'The influence of non-technical factors on code review'. In: *Working Conference on Reverse Engineering (WCRE)*. Ed. by Ralf Lämmel, Rocco Oliveto and Romain Robbes. IEEE Computer Society, Oct. 2013, pp. 122–131. DOI: 10.1109/wcre.2013.6671287.

[13] Moritz Beller, Georgios Gousios and Andy Zaidman. 'TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration'. In: *International Conference on Mining Software Repositories (MSR)*. Ed. by Jesús M. González-Barahona, Abram Hindle and Lin Tan. IEEE Computer Society, 2017, pp. 447–450. DOI: 10.1109/MSR.2017.24.

[14] Brian Berliner. 'CVS II: Parallelizing Software Development'. In: *Winter 1990 USENIX Conference*. Vol. 341. Washington, D.C., Jan. 1990.

[15] Brian Berliner and Nayan B. Ruparelia. 'Early days of CVS'. In: *ACM SIGSOFT Software Engineering Notes* 35.5 (Oct. 2010), pp. 5–6. DOI: 10.1145/1838687.1838689.

[16] Valdis Berzins. 'Software Merge: Semantics of Combining Changes to Programs'. In: *ACM Transactions on Programming Languages and Systems* (1994). DOI: 10.1145/197320.197403.

[17] Christian Bird and Thomas Zimmermann. 'Assessing the Value of Branches with What-If Analysis'. In: *Symposium on the Foundations of Software Engineering (FSE)*. Ed. by Will Tracz, Martin P. Robillard and Tevfik Bultan. ACM, 2012. DOI: 10.1145/2393596.2393648.

[18] Amiangshu Bosu, Michaela Greiler and Christian Bird. 'Characteristics of Useful Code Reviews: An Empirical Study at Microsoft'. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, May 2015. DOI: 10.1109/msr.2015.21.

[19] Ahmed Bouajjani, Constantin Enea and Shuvendu Lahiri. 'Abstract Semantic Diffing of Evolving Concurrent Programs'. In: *Static Analysis Symposium*. 2017. DOI: 10.1007/978-3-319-66706-5_3.

[20] Caius Brindescu, Iftekhar Ahmed, Rafael Leano and Anita Sarma. 'Planning for Untangling: Predicting the Difficulty of Merge Conflicts'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020, pp. 801–811. DOI: 10.1145/3377811.3380344.

[21] Caius Brindescu, Yenifer Ramirez, Anita Sarma and Carlos Jensen. 'Lifting the Curtain on Merge Conflict Resolution: A Sensemaking Perspective'. In: *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2020. DOI: 10.1109/icsme46990.2020.00057.

[22] H. M. Brown. 'CLEAR: Support Software for Large Systems'. In: *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. Ed. by J. N. Buxton and B. Randell. CLEAR was a presentation given at the conference. A transcript of the presentation, the slides, and the discussion afterwards is provided in this document. 1969, 53–60 (40–45). URL: https://eprints.ncl.ac.uk/file_store/production/55593/5C8B829F-C598-48F6-8726-0EF473BB7338.pdf (visited on 11/01/2023).

[23] Yuriy Brun, Reid Holmes, Michael D. Ernst and David Notkin. 'Early Detection of Collaboration Conflicts and Risks'. In: *IEEE Transactions on Software Engineering* 39.10 (Oct. 2013), pp. 1358–1375. DOI: 10.1109/TSE.2013.28.

[24] Yuriy Brun, Reid Holmes, Michael D. Ernst and David Notkin. 'Proactive Detection of Collaboration Conflicts'. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*. Ed. by Tibor Gyimóthy and Andreas Zeller. ACM, Sept. 2011, pp. 168–178. DOI: 10.1145/2025113.2025139.

[25] Jim Buffenbarger. 'Syntactic Software Merging'. In: *Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Selected Papers*. Ed. by Jacky Estublier. Vol. 1005. Lecture Notes in Computer Science. Springer, 1995, pp. 153–172. DOI: 10.1007/3-540-60578-9_14.

[26] Cristian Cadar and Koushik Sen. 'Symbolic Execution for Software Testing: Three Decades Later'. In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 82–90. DOI: 10.1145/2408776.2408795.

[27] Guilherme Cavalcanti, Paulo Borba, Georg Seibt and Sven Apel. 'The Impact of Structure on Software Merging: Semistructured versus Structured Merge'. In: *International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1002–1013. DOI: 10.1109/ASE.2019.00097.

[28] Scott Chacon and Ben Straub. *Pro Git*. v2.1.359. Apress, 2022.

[29] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina and Jennifer Widom. 'Change Detection in Hierarchically Structured Information'. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. Ed. by H. V. Jagadish and Inderpal Singh Mumick. ACM Press, June 1996, pp. 493–504. DOI: 10.1145/233269.233366.

[30] Siyu Chen, Shengbin Xu, Yuan Yao and Feng Xu. 'Untangling Composite Commits by Attributed Graph Clustering'. In: *13th Asia-Pacific Symposium on Internetware*. ACM, June 2022. DOI: 10.1145/3545258.3545267.

[31] Ben Collins-Sussman, Brian W. Fitzpatrick and C. Michael Pilato. *Version Control with Subversion: For Subversion 1.7*. 2011.

[32] Patrick Cousot and Radhia Cousot. 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'. In: *Symposium on Principles of Programming Languages*. Ed. by Robert M. Graham, Michael A. Harrison and Ravi Sethi. ACM, Jan. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[33] Marcela Cunha, Paola Accioly and Paulo Borba. 'The Private Life of Merge Conflicts'. In: *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*. Ed. by Marcelo de Almeida Maia, Fabiano A. Dorça, Rafael Dias Araújo, Christina von Flach, Elisa Yumi Nakagawa and Edna Dias Canedo. ACM, Oct. 2022, pp. 353–362. DOI: 10.1145/3555228.3555240.

[34] Torsten Curdt. *jdeb: This library provides an Ant task and a Maven plugin to create Debian packages from Java builds in a truly cross platform manner*. URL: https://github.com/tcurdt/jdeb (visited on 15/12/2022).

[35] Léuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger and Joao Moisakis. 'Detecting Semantic Conflicts via Automated Behavior Change Detection'. In: *International Conference on Software Maintenance and Evolution (ICSME)*. 2020. DOI: 10.1109/ICSME46990.2020.00026.

[36] Léuson Da Silva, Paulo Borba and Arthur Pires. 'Build conflicts in the wild'. In: *Journal of Software: Evolution and Process* 34.4 (Mar. 2022). DOI: 10.1002/smr.2441.

[37] *Darcs*. URL: https://darcs.net/ (visited on 21/11/2023).

[38] Dan Davison. *delta - A syntax-highlighting pager for git, diff, and grep output*. URL: https://github.com/dandavison/delta (visited on 22/11/2023).

[39] Leonardo de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[40] Cleidson R. B. de Souza, David Redmiles and Paul Dourish. '"Breaking the code", moving between private and public work in collaborative software development'. In: *International ACM SIGGROUP Conference on Supporting Group Work (GROUP)*. Ed. by Kjeld Schmidt, Mark Pendergast, Marilyn Tremaine and Carla Simone. ACM, Nov. 2003, pp. 105–114. DOI: 10.1145/958160.958177.

[41] Richard A. DeMillo, Richard J. Lipton and Frederik G. Sayward. 'Hints on Test Data Selection: Help for the Practicing Programmer'. In: *Computer* 11.4 (1978), pp. 34–41. DOI: 10.1109/C-M.1978.218136.

[42] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou and Stéphane Ducasse. 'Untangling Fine-Grained Code Changes'. In: *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Ed. by Yann-Gaël Guéhéneuc, Bram Adams and Alexander Serebrenik. IEEE Computer Society, Mar. 2015, pp. 341–350. DOI: 10.1109/SANER.2015.7081844.

[43] *diff3(1)*. URL: https://man.openbsd.org/diff3.1 (visited on 13/02/2024).

[44] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik and Shuvendu Lahiri. 'DeepMerge: Learning to Merge Programs'. In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 1599–1614. DOI: 10.1109/tse.2022.3183955.

[45] Project Euler. *Problem 1: Multiples of 3 or 5*. URL: https://projecteuler.net/problem=1 (visited on 15/12/2022).

[46] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez and Martin Monperrus. 'Fine-grained and accurate source code differencing'. In: *International Conference on Automated Software Engineering (ASE)*. Ed. by Ivica Crnkovic, Marsha Chechik and Paul Grünbacher. ACM, 2014, pp. 313–324. DOI: 10.1145/2642937.2642982.

[47] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. 'The Program Dependence Graph and Its Use in Optimization'. In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987), pp. 319–349. DOI: 10.1145/24039.24041.

[48] Beat Fluri, Michael Wursch, Martin Pinzger and Harald C. Gall. 'Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction'. In: *IEEE Transactions on Software Engineering* 33.11 (Nov. 2007), pp. 725–743. DOI: 10.1109/TSE.2007.70731.

[49] Nad Friedman and Github. *GitHub Actions now supports CI/CD, free for public repositories*. 2019. URL: https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/ (visited on 19/02/2024).

[50] Carlos Galindo, Jens Krinke, Sergio Pérez and Josep Silva. 'Field-Sensitive Program Slicing'. In: *International Conference on Software Engineering and Formal Methods (SEFM)*. Ed. by Bernd-Holger Schlingloff and Ming Chai. Vol. 13550. Lecture Notes in Computer Science. Springer International Publishing, 2022, pp. 74–90. ISBN: 9783031171086. DOI: 10.1007/978-3-031-17108-6_5.

[51] Carlos Galindo, Marisa Llorens, Sergio Pérez and Josep Silva. 'Slicing Shared-Memory Concurrent Programs, The Threaded System Dependence Graph Revisited'. In: *International Conference on Software Maintenance and Evolution (ICSME)*. Oct. 2023, pp. 73–83. DOI: 10.1109/ICSME58846.2023.00019.

[52] Carlos Galindo, Sergio Pérez and Josep Silva. 'Exception-sensitive program slicing'. In: *Journal of Logical and Algebraic Methods in Programming* 130 (Jan. 2023), p. 100832. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2022.100832.

[53] Carlos Galindo, Sergio Pérez and Josep Silva. 'Program slicing of Java programs'. In: *Journal of Logical and Algebraic Methods in Programming* 130 (Jan. 2023). ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2022.100826.

[54] Lars Marius Garshol. *Duke: Duke is a fast and flexible deduplication engine written in Java*. URL: https://github.com/larsga/Duke (visited on 12/12/2022).

[55] Gleiph Ghiotto, Leonardo Murta, Marcio Barros and André van der Hoek. '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.

[56] GitHub. *GitHub*. URL: https://github.com/ (visited on 05/04/2023).

[57] Patrice Godefroid. 'Compositional Dynamic Test Generation'. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, Jan. 2007, pp. 47–54. DOI: 10.1145/1190216.1190226.

[58] Patrice Godefroid, Nils Klarlund and Koushik Sen. 'DART: Directed Automated Random Testing'. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*. Ed. by Vivek Sarkar and Mary W. Hall. June 2005, pp. 213–223. DOI: 10.1145/1065010.1065036.

[59] Google. *J2ObjC: A Java to iOS Objective-C translation tool and runtime*. URL: https://github.com/google/j2objc (visited on 02/12/2022).

[60] Georgios Gousios. 'The GHTorrent Dataset and Tool Suite'. In: *International Conference on Mining Software Repositories (MSR)*. Ed. by Thomas Zimmermann, Massimiliano Di Penta and Sunghun Kim. IEEE Computer Society, May 2013, pp. 233–236. DOI: 10.1109/MSR.2013.6624034.

[61] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey and Arie van Deursen. 'Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Antonia Bertolino, Gerardo Canfora and Sebastian G. Elbaum. IEEE Computer Society, May 2015, pp. 358–368. DOI: 10.1109/ICSE.2015.55.

[62] Dick Grune. *Concurrent Versions System, A Method for Independent Cooperation*. Tech. rep. Vrije Universiteit, 1986.

[63] Klaus Havelund and Thomas Pressburger. 'Model checking Java programs using Java PathFinder'. In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 366–381. DOI: 10.1007/s100090050043.

[64] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher A. Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristof Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo-Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matúš Sulír, Fatemeh Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber and Johannes Erbel. 'A fine-grained data set and analysis of tangling in bug fixing commits'. In: *Empirical Software Engineering* 27.6 (July 2022). DOI: 10.1007/s10664-021-10083-5.

[65] Sergio Luis Herrera Gonzalez and Piero Fraternali. 'Almost Rerere: Learning to resolve conflicts in distributed projects'. In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 2255–2271. DOI: 10.1109/tse.2022.3215289.

[66] Kim Herzig, Sascha Just and Andreas Zeller. 'The impact of tangled code changes on defect prediction models'. In: *Empirical Software Engineering* 21.2 (Apr. 2016), pp. 303–336. DOI: 10.1007/s10664-015-9376-6.

[67] Kim Herzig and Andreas Zeller. 'The Impact of Tangled Code Changes'. In: *International Conference on Mining Software Repositories (MSR)*. Ed. by Thomas Zimmermann, Massimiliano Di Penta and Sunghun Kim. IEEE Computer Society, May 2013, pp. 121–130. DOI: 10.1109/MSR.2013.6624018.

[68] Yoshiki Higo. *TinyPDG: A library for building intraprocedural PDGs for Java programs*. URL: https://github.com/YoshikiHigo/TinyPDG.

[69] Yoshiki Higo and Shinji Kusumoto. 'Code Clone Detection on Specialized PDGs with Heuristics'. In: *European Conference on Software Maintenance and Reegineering (CSMR)*. Ed. by Tom Mens, Yiannis Kanellopoulos and Andreas Winter. IEEE Computer Society, Mar. 2011, pp. 75–84. DOI: 10.1109/CSMR.2011.12.

[70] Yoshiki Higo and Shinji Kusumoto. 'Enhancing Quality of Code Clone Detection with Program Dependency Graph'. In: *Working Conference on Reverse Engineering (WCRE)*. Ed. by Andy Zaidman, Giuliano Antoniol and Stéphane Ducasse. IEEE Computer Society, Oct. 2009, pp. 315–316. DOI: 10.1109/WCRE.2009.39.

[71] D. Richard Hipp and Fossil Contributors. *Fossil Versus Git*. URL: https://fossil-scm.org/home/doc/trunk/www/fossil-v-git.wiki (visited on 18/01/2023).

[72] D. Richard Hipp and Fossil Contributors. *Fossil: Home*. URL: https://fossil-scm.org/home/doc/trunk/www/index.wiki (visited on 18/01/2023).

[73] Susan Horwitz. 'Identifying the Semantic and Textual Differences Between Two Versions of a Program'. In: *Conference on Programming Language Design and Implementation (PLDI)*. 1990. DOI: 10.1145/93542.93574.

[74] Susan Horwitz, Jan Prins and Thomas Reps. 'Integrating Noninterfering Versions of Programs'. In: *ACM Transactions on Programming Languages and Systems* 11.3 (1989), pp. 345–387. DOI: 10.1145/65979.65980.

[75] Susan Horwitz, Thomas Reps and David Binkley. 'Interprocedural Slicing Using Dependence Graphs'. In: *ACM Transactions on Programming Languages and Systems* 12.1 (Jan. 1990), pp. 26–60. DOI: 10.1145/77606.77608.

[76] J.J. Hunt and W.F. Tichy. 'Extensible language-aware merging'. In: *International Conference on Software Maintenance, 2002. Proceedings.* IEEE Comput. Soc. DOI: 10.1109/icsm.2002.1167812.

[77] Giuseppe Iaffaldano, Igor Steinmacher, Fabio Calefato, Marco Aurélio Gerosa and Filippo Lanubile. 'Why do developers take breaks from contributing to OSS projects?: a preliminary analysis'. In: *International Workshop on Software Health (SoHeal)*. Ed. by Bram Adams, Eleni Constantinou, Tom Mens, Kate Stewart and Gregorio Robles. IEEE / ACM, May 2019, pp. 9–16. DOI: 10.1109/SoHeal.2019.00009.

[78] Daniel Jackson and David A. Ladd. 'Semantic Diff: A Tool for Summarizing the Effects of Modifications'. In: *International Conference on Software Maintenance (ICSM)*. 1994. DOI: 10.1109/ICSM.1994.336770.

[79] JetBrains. *What version control systems do you regularly use? – Team Tools – The State of Developer Ecosystem in 2023*. URL: https://www.jetbrains.com/lp/devecosystem-2023/team-tools/#tools_vcs (visited on 02/01/2024).

[80] René Just. 'The Major mutation framework: efficient and scalable mutation analysis for Java'. In: *International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 2014. DOI: 10.1145/2610384.2628053.

[81] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán and Daniela E. Damian. 'An in-depth study of the promises and perils of mining GitHub'. In: *Empirical Software Engineering* 21.5 (2016), pp. 2035–2071. DOI: 10.1007/s10664-015-9393-5. URL: https://doi.org/10.1007/s10664-015-9393-5.

[82] Bakhtiar Khan Kasi and Anita Sarma. 'Cassandra: proactive conflict minimization through optimized task scheduling'. In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by David Notkin, Betty H. C. Cheng and Klaus Pohl. IEEE Computer Society, 2013, pp. 732–741. DOI: 10.1109/ICSE.2013.6606619.

[83] David Kawrykow and Martin P. Robillard. 'Non-essential changes in version histories'. In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM, May 2011. DOI: 10.1145/1985793.1985842.

[84] Sanjeev Khanna, Keshav Kunal and Benjamin C. Pierce. 'A Formal Investigation of diff3'. In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings*. Ed. by Vikraman Arvind and Sanjiva Prasad. Vol. 4855. Lecture Notes in Computer Science. Springer, 2007, pp. 485–496. DOI: 10.1007/978-3-540-77050-3_40.

[85] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr. and Andreas Zeller. 'Predicting Faults from Cached History'. In: *29th International Conference on Software Engineering (ICSE)*. IEEE, May 2007. DOI: 10.1109/icse.2007.66.

[86] James C. King. 'Symbolic Execution and Program Testing'. In: *Communications of the ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[87] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta and Shinji Kusumoto. 'Hey! Are You Committing Tangled Changes?' In: *International Conference on Program Comprehension (ICPC)*. 2014. DOI: 10.1145/2597008.2597798.

[88] Pavneet Singh Kochhar, Yuan Tian and David Lo. 'Potential biases in bug localization'. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, Sept. 2014. DOI: 10.1145/2642937.2642997.

[89] Oleksii Kononenko, Olga Baysal and Michael W. Godfrey. 'Code review quality'. In: *Proceedings of the 38th International Conference on Software Engineering*. ACM, May 2016. DOI: 10.1145/2884781.2884840.

[90] Bogdan Korel and Janusz Laski. 'Dynamic program slicing'. In: *Information Processing Letters* 29.3 (Oct. 1988), pp. 155–163. ISSN: 0020-0190. DOI: 10.1016/0020-0190(88)90054-3.

[91] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M. Eskofier and Michael Philippsen. 'Automatic Clustering of Code Changes'. In: *International Conference on Mining Software Repositories (MSR)*. Ed. by Miryung Kim, Romain Robbes and Christian Bird. ACM, May 2016, pp. 61–72. DOI: 10.1145/2901739.2901749.

[92] Shuvendu K. Lahiri, Kapil Vaswani and Charles Antony Richard Hoare. 'Differential Static Analysis: Opportunities, Applications, and Challenges'. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER)*. 2010. DOI: 10.1145/1882362.1882405.

[93] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian Kästner and Claus Hunsen. 'Indicators for merge conflicts in the wild: survey and empirical study'. In: *Automated Software Engineering* 25.2 (Sept. 2018), pp. 279–313. DOI: 10.1007/s10515-017-0227-0.

[94] Yi Li, Julia Rubin and Marsha Chechik. 'Semantic Slicing of Software Version Histories'. In: *International Conference on Automated Software Engineering (ASE)*. 2015. DOI: 10.1109/ASE.2015.47.

[95] Yi Li, Chenguang Zhu, Julia Rubin and Marsha Chechik. 'Precise semantic history slicing through dynamic delta refinement'. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, Aug. 2016. DOI: 10.1145/2970276.2970336.

[96] Tom Lord. *Re: svn diff, svn merge, and vendor branches (long) (Subversion Dev mailing list)*. 2002. URL: https://svn.haxx.se/dev/archive-2002-12/0822.shtml (visited on 16/02/2023).

[97] Tom Lord and Andy Tai. *GNU arch*. URL: https://www.gnu.org/software/gnu-arch/ (visited on 17/01/2023).

[98] Victor da C. Luna Freire, João Brunet and Jorge C. A. de Figueiredo. 'Automatic Decomposition of Java Open Source Pull Requests: A Replication Study'. In: *SOFSEM 2018: Theory and Practice of Computer Science*. Springer International Publishing, 2018, pp. 255–268. DOI: 10.1007/978-3-319-73117-9_18.

[99] Wardah Mahmood, Moses Chagama, Thorsten Berger and Regina Hebig. 'Causes of Merge Conflicts: A Case Study of Elasticsearch'. In: *International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, Feb. 2020. DOI: 10.1145/3377024.3377047.

[100] 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, Feb. 2019. DOI: 10.1109/saner.2019.8668012.

[101] Shane McKee, Nicholas Nelson, Anita Sarma and Danny Dig. 'Software Practitioner Perspectives on Merge Conflicts and Resolutions'. In: *International Conference on Software Maintenance and Evolution (ICSME)*. 2017. DOI: 10.1109/ICSME.2017.53.

[102] Larry McVoy. *A solution for growing pains*. Posted on the linux-kernel mailing list. 1998. URL: https://lkml.org/lkml/1998/9/30/122 (visited on 17/01/2023).

[103] Larry McVoy. *BitKeeper features*. 1999. URL: https://web.archive.org/web/19991002233835/http://www.bitkeeper.com/bk03.html (visited on 17/01/2023).

[104] Larry McVoy. *BitKeeper: Current Status*. 2000. URL: https://web.archive.org/web/20000617153000/http://bitkeeper.com/bk07.html (visited on 17/01/2023).

[105] Tom Mens. 'A State-of-the-Art Survey on Software Merging'. In: *IEEE Transactions on Software Engineering* 28.5 (May 2002), pp. 449–462. DOI: 10.1109/TSE.2002.1000449.

[106] Pierre-Étienne Meunier and Florent Becker. *Pijul*. URL: https://pijul.org/ (visited on 21/11/2023).

[107] Pierre-Étienne Meunier and Florent Becker. *Why Pijul - The Pijul manual*. URL: https://pijul.org/manual/why_pijul.html (visited on 21/11/2023).

[108] Chris Mills, Jevgenija Pantiuchina, Esteban Parra, Gabriele Bavota and Sonia Haiduc. 'Are Bug Reports Enough for Text Retrieval-Based Bug Localization?' In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2018. DOI: 10.1109/icsme.2018.00046.

[109] Chris Mills, Esteban Parra, Jevgenija Pantiuchina, Gabriele Bavota and Sonia Haiduc. 'On the relationship between bug reports and queries for text retrieval-based bug localization'. In: *Empirical Software Engineering* 25.5 (July 2020), pp. 3086–3127. DOI: 10.1007/s10664-020-09823-w.

[110] Audris Mockus and Lawrence G. Votta. 'Identifying Reasons for Software Changes Using Historic Databases'. In: *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2000, pp. 120–130. DOI: 10.1109/ICSM.2000.883028.

[111] Emerson Murphy-Hill and Andrew P. Black. 'Refactoring Tools: Fitness for Purpose'. In: *IEEE Software* 25.5 (Sept. 2008), pp. 38–44. ISSN: 1937-4194. DOI: 10.1109/MS.2008.123.

[112] Ward Muylaert. 'Function Summaries in Dynamic Symbolic Execution for JavaScript'. MA thesis. Vrije Universiteit Brussel, 2015.

[113] Ward Muylaert. *Ward Muylaert - GitLab*. URL: https://gitlab.soft.vub.ac.be/ward (visited on 20/02/2024).

[114] Ward Muylaert and Coen De Roover. 'Prevalence of Botched Code Integrations'. In: *14th International Conference on Mining Software Repositories (MSR)*. Ed. by Jesús M. González-Barahona, Abram Hindle and Lin Tan. IEEE Computer Society, May 2017, pp. 503–506. DOI: 10.1109/MSR.2017.40.

[115] Ward Muylaert and Coen De Roover. 'Untangling Composite Commits Using Program Slicing'. In: *18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, Sept. 2018, pp. 193–202. DOI: `10.1109/SCAM.2018.00030`.

[116] Ward Muylaert and Coen De Roover. 'Untangling Source Code Changes Using Program Slicing'. In: *BElgian-NEtherlands software eVOLution symposium (BENEVOL)*. Ed. by Serge Demeyer, Ali Parsai, Gulsher Laghari and Brent van Bladel. Vol. 2047. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 36–38. URL: `https://ceur-ws.org/Vol-2047/BENEVOL_2017_paper_10.pdf`.

[117] Ward Muylaert, Johannes Härtel and Coen De Roover. 'Symbolic Execution to Detect Semantic Merge Conflicts'. In: *23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Ed. by Leon Moonen, Christian D. Newman and Alessandra Gorla. IEEE, Oct. 2023, pp. 186–197. DOI: `10.1109/SCAM59687.2023.00028`.

[118] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma and Danny Dig. 'The life-cycle of merge conflicts: processes, barriers, and strategies'. In: *Empirical Software Engineering* 24.5 (Feb. 2019), pp. 2863–2906. DOI: `10.1007/s10664-018-9674-x`.

[119] Hoan Anh Nguyen, Anh Tuan Nguyen and Tien N. Nguyen. 'Filtering Noise in Mixed-Purpose Fixing Commits to Improve Defect Prediction and Localization'. In: *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2013, pp. 138–147. DOI: `10.1109/ISSRE.2013.6698913`.

[120] Jonathan Nieder, Stefan Beller and Git Contributors. *hash-function-transition Documentation*. URL: `https://git-scm.com/docs/hash-function-transition` (visited on 25/01/2023).

[121] N. Niu, S. Easterbrook and M. Sabetzadeh. 'A category-theoretic approach to syntactic software merging'. In: *International Conference on Software Maintenance (ICSM)*. IEEE, 2005. DOI: `10.1109/icsm.2005.6`.

[122] Yannic Noller, Corina Pasareanu, Marcel Bohme, Youcheng Sun, Hoang Lam Nguyen and Lars Grunske. 'HyDiff: Hybrid Differential Software Analysis'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020, pp. 1273–1285. DOI: `10.1145/3377811.3380363`.

[123] Open Hub. *Compare Repositories*. 2024. URL: `https://openhub.net/repositories/compare` (visited on 02/01/2024).

[124] 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, Sept. 2019. DOI: 10.1109/esem.2019.8870173.

[125] Hristina Palikareva, Tomasz Kuchta and Cristian Cadar. 'Shadow of a Doubt: Testing for Divergences Between Software Versions'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Laura K. Dillon, Willem Visser and Laurie A. Williams. ACM, 2016, pp. 1181–1192. DOI: 10.1145/2884781.2884845.

[126] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri and Mike Kaufman. 'Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis'. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, May 2021. DOI: 10.1109/icse43902.2021.00077.

[127] Profir-Petru Pârtachi, Santanu Kumar Dash, Miltiadis Allamanis and Earl T. Barr. 'Flexeme: untangling commits using lexical flows'. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Nov. 2020. DOI: 10.1145/3368089.3409693.

[128] Nimrod Partush and Eran Yahav. 'Abstract Semantic Differencing for Numerical Programs'. In: *Static Analysis Symposium (SAS)*. 2013. DOI: 10.1007/978-3-642-38856-9_14.

[129] Nimrod Partush and Eran Yahav. 'Abstract Semantic Differencing via Speculative Correlation'. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 2014. DOI: 10.1145/2660193.2660245.

[130] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person and Mark Pape. 'Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software'. In: *International Symposium on Software Testing and Analysis (ISSTA)*. Ed. by Barbara G. Ryder and Andreas Zeller. ACM, 2008, pp. 15–26. DOI: 10.1145/1390630.1390635.

[131] Fabrizio Pastore, Leonardo Mariani and Daniela Micucci. 'BDCI: behavioral driven conflict identification'. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. Ed. by Eric Bodden, Wilhelm Schäfer, Arie van Deursen and Andrea Zisman. ACM, 2017, pp. 570–581. DOI: 10.1145/3106237.3106296.

[132] Sergio Pérez Rubio. 'Analysis Techniques for Software Maintenance'. PhD thesis. Universitat Politècnica de València, Apr. 2023.

[133] Dewayne E. Perry, Harvey P. Siy and Lawrence G. Votta. 'Parallel Changes in Large-Scale Software Development: An Observational Case Study'. In: *ACM Transactions on Software Engineering and Methodology* 10.3 (July 2001), pp. 308–337. DOI: 10.1145/383876.383878.

[134] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum and Corina S. Pasareanu. 'Differential Symbolic Execution'. In: *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, Nov. 2008, pp. 226–237. DOI: 10.1145/1453101.1453131.

[135] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli and Premkumar Devanbu. 'On the "Naturalness" of Buggy Code'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Laura K. Dillon, Willem Visser and Laurie A. Williams. ACM, 2016, pp. 428–439. DOI: 10.1145/2884781.2884848.

[136] Eric Raymond. *Understanding Version-Control Systems (DRAFT)*. URL: http://www.catb.org/~esr/writings/version-control/version-control.html (visited on 17/01/2023).

[137] Marc J. Rochkind. 'The source code control system'. In: *IEEE Transactions on Software Engineering* SE-1.4 (Dec. 1975), pp. 364–370. DOI: 10.1109/tse.1975.6312866.

[138] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko and Alberto Bacchelli. 'Modern Code Review: A Case Study at Google'. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, May 2018. DOI: 10.1145/3183519.3183525.

[139] Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba and Sven Apel. 'Leveraging Structure in Software Merge: An Empirical Study'. In: *IEEE Transactions on Software Engineering* 48.11 (Nov. 2022), pp. 4590–4610. DOI: 10.1109/tse.2021.3123143.

[140] Koushik Sen, Darko Marinov and Gul Agha. 'CUTE: A Concolic Unit Testing Engine for C'. In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC-FSE)*. Ed. by Michel Wermelinger and Harald C. Gall. ACM, 2005, pp. 263–272. DOI: 10.1145/1081706.1081750.

[141] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin and Qianxiang Wang. 'IntelliMerge: A Refactoring-aware Software Merging Technique'. In: *OOPSLA*. Vol. 3. OOPSLA. New York, NY, USA: ACM, Oct. 2019, 170:1–170:28. DOI: 10.1145/3360596.

[142] Bowen Shen, Muhammad Ali Gulzar, Fei He and Na Meng. 'A Characterization Study of Merge Conflicts in Java Projects'. In: *ACM Transactions on Software Engineering and Methodology* 32.2 (July 2022), 40:1–40:28. DOI: 10.1145/3546944.

[143] Josep Silva. 'A Vocabulary of Program Slicing-Based Techniques'. In: *ACM Computing Surveys* 44.3 (June 2012). DOI: 10.1145/2187671.2187674.

[144] Jacek Śliwerski, Thomas Zimmermann and Andreas Zeller. 'When Do Changes Induce Fixes? (On Fridays.)' In: *Mining Software Repositories (MSR)*. ACM, 2005. DOI: 10.1145/1083142.1083147.

[145] Marcelo Sousa, Isil Dillig and Shuvendu K. Lahiri. 'Verified Three-Way Program Merge'. In: *PACMPL* 2.OOPSLA (2018), 165:1–165:29. DOI: 10.1145/3276535.

[146] Karen Spärck Jones and Cornelis Joost van Rijsbergen. *Report on the need for and provision of an "ideal" information retrieval test collection*. Tech. rep. Computer Laboratory, University of Cambridge, 1975.

[147] Spotify Web API Java. *Spotify Web API Java: A Java wrapper for Spotify's Web API*. URL: https://github.com/spotify-web-api-java/spotify-web-api-java (visited on 13/12/2022).

[148] Spring Cloud. *Spring Cloud Config: External configuration (server and client) for Spring Cloud*. URL: https://github.com/spring-cloud/spring-cloud-config (visited on 18/02/2024).

[149] Reinout Stevens and Coen De Roover. 'Extracting Executable Transformations from Distilled Code Changes'. In: *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017. DOI: 10.1109/SANER.2017.7884619.

[150] Reinout Stevens and Coen De Roover. 'Querying the History of Software Projects using QwalKeko'. In: *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, 2014, pp. 585–588. DOI: 10.1109/ICSME.2014.101.

[151] Quentin Stiévenart, David W. Binkley and Coen De Roover. 'Static stack-preserving intra-procedural slicing of webassembly binaries'. In: *International Conference on Software Engineering (ICSE)*. ACM, May 2022, pp. 2031–2042. DOI: 10.1145/3510003.3510070.

[152] Chunga Sung, Shuvendu K. Lahiri, Mike Kaufman, Pallavi Choudhury and Chao Wang. 'Towards Understanding and Fixing Upstream Merge Induced Conflicts in Divergent Forks: An Industrial Case Study'. In: *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2020. DOI: 10.1145/3377813.3381362.

[153] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan and Shuvendu K. Lahiri. 'Program merge conflict resolution via neural transformers'. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Nov. 2022. DOI: 10.1145/3540250.3549163.

[154] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang and Sunghun Kim. 'How Do Software Engineers Understand Code Changes? - An Exploratory Study in Industry'. In: *International Symposium on the Foundations of Software Engineering (FSE)*. Ed. by Will Tracz, Martin P. Robillard and Tevfik Bultan. ACM, 2012. DOI: 10.1145/2393596.2393656.

[155] Yida Tao and Sunghun Kim. 'Partitioning Composite Code Changes to Facilitate Code Review'. In: *International Conference on Mining Software Repositories (MSR)*. Ed. by Massimiliano Di Penta, Martin Pinzger and Romain Robbes. IEEE Computer Society, 2015, pp. 180–190. DOI: 10.1109/MSR.2015.24.

[156] Walter F. Tichy. *Design, Implementation and Evaluation of a Revision Control System*. Tech. rep. 323. Purdue University, 1982.

[157] Walter F. Tichy. 'Design, Implementation, and Evaluation of a Revision Control System'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Yutaka Ohno, Victor R. Basili, Hajime Enomoto, Koji Kobayashi and Raymond T. Yeh. IEEE Computer Society, 1982, pp. 58–67.

[158] Walter F. Tichy. *RCS: A System for Version Control*. Tech. rep. 394. Purdue University, 1984.

[159] Linus Torvalds. *Re: Kernel SCM saga..* 2005. URL: https://marc.info/?l=linux-kernel%5C&m=111288700902396 (visited on 17/01/2023).

[160] Sheikh Shadab Towqir, Bowen Shen, Muhammad Ali Gulzar and Na Meng. 'Detecting Build Conflicts in Software Merge for Java Programs via Static Analysis'. In: *International Conference on Automated Software Engineering (ASE)*. 2022. DOI: 10.1145/3551349.3556950.

[161] Travis CI. *The new pricing model for travis-ci.com*. 2020. URL: https://www.travis-ci.com/blog/2020-11-02-travis-ci-new-billing/ (visited on 19/02/2024).

[162] Travis CI. *Travis-CI*. URL: https://www.travis-ci.com/ (visited on 05/04/2023).

[163] Jason Tsay, Laura Dabbish and James Herbsleb. 'Influence of social and technical factors for evaluating contribution in GitHub'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Pankaj Jalote, Lionel C. Briand and André van der Hoek. ACM, May 2014, pp. 356–366. DOI: 10.1145/2568225.2568315.

[164] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo and Sven Apel. 'Challenges of Resolving Merge Conflicts: A Mining and Survey Study'. In: *IEEE Transactions on Software Engineering* 48.12 (Dec. 2022), pp. 4964–4985. DOI: 10.1109/tse.2021.3130098.

[165] Maarten Vandercammen. 'Inter-process Concolic Testing of Full-stack JavaScript Web Applications'. PhD thesis. Vrije Universiteit Brussel, Oct. 2023. ISBN: 9789464443806.

[166] *vimdiff*. URL: https://git-scm.com/docs/vimdiff/en (visited on 22/11/2023).

[167] Min Wang, Zeqi Lin, Yanzhen Zou and Bing Xie. 'CoRA: Decomposing and Describing Tangled Code Changes for Reviewer'. In: *International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1050–1061. DOI: 10.1109/ASE.2019.00101.

[168] We Love Coding. *EditorConfig Netbeans Plugin: A Netbeans IDE plugin supporting the EditorConfig standard*. URL: https://github.com/welovecoding/editorconfig-netbeans (visited on 06/12/2022).

[169] Mark Weiser. 'Program Slicing'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Seymour Jeffrey and Leon G. Stucki. IEEE Computer Society, 1981, pp. 439–449.

[170] Mark Weiser. 'Program Slicing'. In: *IEEE Transactions on Software Engineering* SE-10.4 (July 1984), pp. 352–357. DOI: 10.1109/tse.1984.5010248.

[171] Bernhard Westfechtel. 'Structure-Oriented Merging of Revisions of Software Documents'. In: *Proceedings of the 3rd International Workshop on Software Configuration Management, Trondheim, Norway, June 12-14, 1991*. Ed. by Peter H. Feiler. ACM Press, 1991, pp. 68–79. DOI: 10.1145/111062.111071.

[172] Thorsten Wuensche, Artur Andrzejak and Sascha Schwedes. 'Detecting Higher-Order Merge Conflicts in Large Software Projects'. In: *International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Oct. 2020. DOI: 10.1109/icst46399.2020.00043.

[173] Jian Zhou, Hongyu Zhang and David Lo. 'Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports'. In: *34th International Conference on Software Engineering (ICSE)*. IEEE, June 2012. DOI: `10.1109/icse.2012.6227210`.

[174] Martin von Zweigbergk. *Jujutsu - a version control system*. URL: `https://github.com/martinvonz/jj` (visited on 21/11/2023).