UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Léuson Mário Pedro da Silva

Detecting, Understanding, and Resolving Build and Test Conflicts

Recife

2022

Léuson Mário Pedro da Silva

Detecting, Understanding, and Resolving Build and Test Conflicts

A Ph.D. Thesis presented to the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

Concentration Area: Software Engineering and Programing Languages

Advisor: Prof. Dr. Paulo Henrique Monteiro Borba

Co-Advisor: Prof. Dr. Thorsten Berger

Recife

2022

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S586d Silva, Léuson Mário Pedro da
Detecting, understanding, and resolving build and test conflicts / Léuson Mário Pedro da Silva. – 2022.
150 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022.
Inclui referências.

1. Engenharia de software. 2. Integração de código. I. Borba, Paulo Henrique Monteiro (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2022 – 61

Leuson Mario Pedro da Silva

“Detecting, Understanding, and Resolving Build and Test Conflicts”

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 24/02/2022.

_____

Orientador: Prof. Dr. Paulo Henrique Monteiro Borba

BANCA EXAMINADORA

_____

Prof. Dr. Márcio Lopes Cornélio
Centro de Informática / UFPE

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda
Centro de Informática / UFPE

_____

Prof. Dr. Everton Leandro Galdino Alves
Centro de Engenharia Elétrica e Informática / UFCG

_____

Prof. Dr. Márcio de Oliveira Barros
Centro de Ciências Exatas e Tecnologia / UERJ

_____

Prof. Dr. Rodrigo Bonifácio de Almeida
Departamento de Ciência da Computação / UnB

I dedicate this work to everyone who gave me the necessary support to get here.

ACKNOWLEDGEMENTS

Durante a realização deste trabalho, pude contar com o apoio e suporte de colegas e amigos, que me ajudaram durante todo este processo. Assim, gostaria de falar a todas estas pessoas: muito obrigado.

Inicialmente, gostaria de agradecer a minha família pelo apoio, confiança e acolhimento que recebi em todos os desafios, que decidi enfrentar. Em muitos momentos, tenho consiência de que foi difícil entender e apoiar as minhas decisões, pois não tínhamos entendimento do que eu estava conquistando. Em especial, gostaria de agradecer a minha mãe, que sempre se manteve forte e firme em garantir as condições mínimas para que eu estudasse.

Agradeço ao meu orientador Paulo Borba, pela sua dedicação, bondade e excelência durante toda a orientação deste trabalho. Ao longo dos últimos anos, pude desenvolver habilidades técnicas e pessoais, que só foram possíveis por conta do seu olhar cuidadoso. Obrigado por todo os ensinamentos e oportunidades, que me permitiram tornar o profissional que eu sempre quis ser. No mesmo sentido, estendo meus agradecimentos ao meu co-orientador Thorsten Berger, pela colaboração neste trabalho, bem como pela amistosa receptividade durante meu período de intercâmbio na universidade de Chalmers.

Agradeço também a João Moisakis, Arthur Pires e Wardah Mahmood pela colaboração nos artigos, que foram frutos deste trabalho. Adicionalmente, gostaria de agradecer a Toni Maciel, Vinícius Leite e Aldiberg Gomes pela dedicação nos trabalhos realizados, enriquecendo ainda mais esta tese. Adicionalmente, agradeço aos colegas Galileu, Matheus, Rafael e Thaís pelo suporte durante a realização dos estudos.

Agradeço aos professores Breno Miranda, Márcio Cornélio, Everton Alves, Márcio Barros e Rodrigo Bonifácio por terem aceito nosso convite para avaliar esta tese. Adicionalmente, gostaria de agradecer aos professores Leonardo Murta e Marcelo D'Amorim pelas suas contribuições durante meu exame de qualificação. Suas contribuições trouxeram ganhos para este e futuros trabalhos.

Agradeço aos amigos que fiz durante todo este processo. Assim, agradeço a João Carlos, Beatriz, Marcela, Camila, Rogério, Bia, Karine, Neto, Paola, Guilherme, Renê, Miriam, Rashida e Yang; vocês tornaram este processo mais leve e me trouxeram alegria nos momentos de angústia. Em especial gostaria de agradecer a minha amiga Klissiomara Dias, por todos os conselhos, risos e conversas; você foi a minha base, quando eu me sentia sem rumo. Muito obrigado pela sua amizade e confiança. Agradeço também a Wardah Mahmood, que foi a minha "bestie" e parceira, durante minha estadia em Gotemburgo; lhe conhecer me fez refletir sobre como há seres humanos doces e incríveis ao nosso redor.

Agradeço ao CIn, este centro que me acolheu desde 2016, quando comecei o meu mestrado. Tenho muito orgulho de fazer parte deste centro e tê-lo como parte da minha história. Gostaria de agradecer aos membros do SPG pelas contribuições e conselhos, bem

como os demais integrantes do LabES pelos momentos de descontração.

Agradeço a Recife, pelas suas belezas, particularidades e acolhimento. Vivi os momentos mais felizes e intensos da minha vida aqui. E de onde eu estiver, eu sempre terei um espaço reservado no meu coração a esta cidade.

Por último, não menos importante, agradeço à Deus, por tudo.

ABSTRACT

During collaborative software development, developers usually adopt branching and merging practices when making their contributions, allowing them to contribute to a software project independently. Despite such benefits, branching and merging come at a cost — the need to merge software and to resolve merge conflicts, which often occur in practice. While modern merge techniques, such as 3-way or structured merge, can automatically resolve many such conflicts, they fail when conflicts arise at the semantic level, known as *semantic conflicts*. These conflicts are revealed by failures when building and testing integrated code, *build* and *test conflicts*, respectively. Detecting such semantic conflicts requires understanding the software's behavior, which is beyond the capabilities of most existing merge and assistive tools. To address the need for better assistive tools, we investigate semantic conflict occurrence by finding their causes and proposing tools that could support developers when facing these conflicts during merge scenario integrations. Initially, we perform a study investigating the frequency, structure, and adopted resolution patterns of build conflicts by empirically analyzing 451 open-source Java projects. As a result, we provide a catalogue of conflicts with 239 occurrences spread into six categories. Most build conflicts are caused by missing declarations removed or renamed by one developer but referenced by someone else. Further, analyzing some of these conflicts, we also report a catalogue of resolution patterns. In order to evaluate the occurrence of test conflicts, we adopt a different approach as these conflicts involve the semantics of a program. Consequently, they can not be detected during the compilation phase of the build process. This way, initially, we perform a second study investigating their occurrence by exploring the automated creation of unit tests as partial specifications to detect conflicts. Relying on a ground-truth dataset of more than 80 mutually integrated changes' pairs on class elements from 51 software merge scenarios, we manually analyzed them and investigated whether test conflicts exist. Next, we systematically explore the detection of conflicts through four unit-test generation tools, as also the adoption of Testability Transformations aiming to increase the testability of the code under analysis. As a result, we provide a catalogue of 28 test conflicts, of which 9 of them were detected by our approach. Our results show that the best approach to detect conflicts involves combining the tools Differential EvoSuite and EvoSuite applied with Testability Transformations. As a final contribution, we present SAM, a semantic merge tool based on unit test generation, which warns developers about prominent conflicts on ongoing merge scenario integration. Overall, our results bridge a gap in the literature regarding the occurrence of code integration semantic conflicts during software development. Based on each conflict type, we investigate their causes and options to deal with them. While build conflicts could be automatically detected and fixed by an automatic repair tool, test conflicts might be detected by a semantic merge tool based on unit test generation.

Keywords: code integration; semantic conflict; collaborative development.

RESUMO

Durante o desenvolvimento colaborativo de software, desenvolvedores adotam práticas de criação e integração de ramos de desenvolvimento quando trabalham em suas contribuições, permitindo que eles contribuam para um projeto de software independentemente. Apesar destes benefícios, estas práticas vem com um custo, a necessidade de integrar software e resolver conflitos de merge. Enquanto técnicas modernas de merge podem resolver automaticamente muitos destes conflitos, elas falham quando o conflito surge no nível semântico conhecidos como conflitos semânticos. Estes conflitos são revelados por falhas durante o processo de *build* e execução de testes do código integrado conhecidos como conflitos de *build* e teste, respectivamente. Detectar estes conflitos semânticos requer um entendimento do comportamento do software, o que está além da capacidade da maioria das ferramentas assistentes e de integração de código. Dada a necessidade de melhores ferramentas assistentes, nós investigamos a ocorrência de conflitos semânticos identificando suas causas e propondo ferramentas que possam apoiar os desenvolvedores. Inicialmente, nós realizamos um estudo identificando a frequência, estrutura e padrões de resolução adotados em conflitos de *build* analisando empiricamente 451 projetos Java *open-source*. Como resultado, nós provemos um catálogo de conflitos com 239 ocorrências divididos em seis categorias. A maioria dos conflitos de *build* são causados por declarações não-resolvidas, removidas ou renomeadas por um desenvolvedor mas referenciadas por outra pessoa. Além disso, analisando alguns destes conflitos, nós também reportamos um catálogo de padrões de resolução. Para avaliar a ocorrência de conflitos de teste, nós adotamos uma abordagem diferente, pois estes conflitos envolvem a semântica de um programa. Consequentemente, eles não podem ser detectados durante a fase de compilação do processo de *build*. Desta forma, nós realizamos um segundo estudo investigando sua ocorrência, explorando a criação automática de testes unitários como especificações parciais para detectar conflitos. Baseando-se em um conjunto de mais de 80 mudanças mútuas em elementos de classes de 51 cenários de merge com *ground-truth*, nós analisamos e investigamos se conflitos de teste existiam. Em seguida, nós exploramos sistematicamente a detecção de conflitos por meio de ferramentas de geração de testes, como também a adoção de Transformações de Testabilidade visando aumentar a testabilidade do código em análise. Como resultado, nós apresentamos um catálogo de 28 conflitos de testes, dos quais 9 foram detectados por nossa abordagem. Nossos resultados mostram que a melhor abordagem para detectar conflitos envolve a combinação das ferramentas Differential EvoSuite e EvoSuite aplicadas com Transformações de Testabilidade. Como contribuição final, nós apresentamos SAM, uma ferramenta de merge semântica baseada na geração de testes unitários, que avisa desenvolvedores sobre conflitos em cenários de merge em andamento. No geral, nossos resultados se atentam a uma lacuna na literatura sobre a ocorrência de conflitos de integração de código semânticos durante o desenvolvimento de software. Baseado no tipo de conflito, nós investigamos suas causas e opções para lidar com

eles. Enquanto conflitos de *build* podem ser detectados e solucionados por uma ferramenta de reparo automático, conflitos de teste poderiam ser detectados por uma ferramenta de merge semântica baseada na geração de testes unitários.

Palavras-chaves: integração de código; conflitos semânticos; desenvolvimento colaborativo.

LIST OF FIGURES

Figure 1 – Centralised version control paradigm. The arrows among the individual (blue) and central (red) repositories indicate that only the central repository can accept others' changes. Individual repositories use the central one to update themselves with new information (synchronization process). Changes from individual repositories are only accessible if the central repository already holds them. 23
Figure 2 – Decentralised version control paradigm. The arrows among the individual repositories (blue) indicate each repository can accept changes and update themselves from the others. Individual repositories are accessed directly without any intermediary repository. 23
Figure 3 – Merge scenario with merge conflict. 25
Figure 4 – Build conflicts in practice. On the bottom, blue and pink circles represent commits done by Rachel and Lucas, respectively. In contrast, the orange circle represents the previous commit in the main branch before the described merge scenario. Above some developers' commits, gray squares represent the build processes done locally on the developers' workspace (✓, for successful builds). Black arrows link commits to their ancestors. Finally, on the top, green arrows associate commits from the remote repository with their build processes (blue boxes) on the Continuous Integration (CI) service. Above each build, we present its status (✓ and **!**, for successful and errored builds, respectively). For simplicity, we do not present commits done on developers' private repositories. 27
Figure 5 – A merge of two changes (each parent added one of the highlighted lines) that are semantically conflicting. 30
Figure 6 – Java class given as input for unit test generation tool. 32
Figure 7 – Test class generated by Randoop based on given target class and method name. 33
Figure 8 – Build conflicts: empirical study setup. The first step (Section 3.1.2.1) mines project repositories. Step two (Section 3.1.2.2) filters merge scenarios and yield build conflict candidates. Finally, step three (Section 3.1.2.3) classifies build conflict causes and resolution patterns. 38

Figure 9 – All adopted steps for the detection of build conflicts. In the first step, merge scenarios are classified based on their main associated build breakage types (related or not with environmental issues). In the second step, for each merge scenario with a possible conflict, our scripts extract the associated build breakage cause. Finally, in step three, conflicts are detected based on syntactic analysis of parent contributions' changes, a new heuristic involving parent build statuses and merge scenario state, or manual analysis. 43
Figure 10 – Unimplemented Method conflict caused by new method added on old interface. 52
Figure 11 – Unimplemented Method conflict caused by interface class location update. 52
Figure 12 – Unimplemented Method conflict caused by method name update. . . . 53
Figure 13 – Incompatible Method Signature Conflict caused by method signature update. 54
Figure 14 – Project Rules conflict caused by unfollowed project guideline. 55
Figure 15 – Build conflict theory for Java programming language. Changes' pairs for the Java programming language that might result in build conflict when performed in parallel. We report 15 changes' pairs, which 10 of them are covered by our results. 61
Figure 16 – Syntactic diff used to identify the new variable name 67
Figure 17 – Overview of steps performed by SAM and its workflow. 75
Figure 18 – Heuristics for conflict detection criteria. Each conflict criterion is based on the output of a given test considering the analyzed versions of the system corresponding to the merge scenario commits. The green checkmark stands for a test with a passed output, while the red X stands for a failing output. 79
Figure 19 – A test case revealing an interference from the example presented in Figure 5. 79
Figure 20 – A merge of two changes (each parent removed one of the highlighted lines) that are semantically conflicting. 80
Figure 21 – A conflict occurs and is propagated trhough a private field. 83
Figure 22 – Class with complex required objects, given as input for unit test tools. 85
Figure 23 – Irrelevant test case generated by unit test tools due to not well-formed objects. 86
Figure 24 – Example of a serialized object given as input for unit test tools. 86
Figure 25 – Relevant generated test case using serialized objects as input. 87
Figure 26 – Instrumentation applied on the target method `build()` by OSean.EX. . 87
Figure 27 – Providing serialized objects through individual method declarations. . . 88
Figure 28 – A test suite generated by the original version of Randoop. 89

Figure 29 – A test suite generated by Randoop Clean. 89
Figure 30 – Test conflicts: empirical study setup. We start with selecting Java projects on GitHub and then filtering merge scenarios with changes on mutual class elements. Next, we compose our sample by generating three different binary file versions, followed by calls to the unit test generation tools generating test suites. Finally, we execute the generated test suites to detect test conflicts and assess further metrics. Besides that, we also perform a manual analysis to verify false positives and negatives in our sample, but that is not illustrated in the figure. 94
Figure 31 – The generation process of binary files for merge scenario commits. For each merge scenario of our sample, we generate binary files, which are given as inputs for the unit test tools. Initially, we generate binary files for the 85 cases of our sample using the original code and applying Testability Transformations. Next, for a subsample of 20 cases, we generate binary files with Testability Transformations and Serialization. 96
Figure 32 – Generation and execution of test suites. For each case of our sample, we generate test suites based on both merge scenario parent's commits. Next, we execute three times each generated test suites against all merge scenario commits in order to calculate our metrics. 98
Figure 33 – Results based on merge scenarios conflict detection and occurrence using all four unit test tools with different binary file versions. Distribution of changes (on same class element declaration), their classification, and whether a conflict is detected or not. Note: *Refactor.* abbreviates *Refactoring*, *No Interfer.* abbreviates *No Interference*. 104
Figure 34 – Test case generated by EvoSuite detecting test conflict. 105
Figure 35 – Test conflict caused by changes that update same variable. 106
Figure 36 – False-negative test conflict caused by changes on the same variable. . . 109
Figure 37 – False-negative test conflict caused by changes on local variable. 110
Figure 38 – Slightly adaptation of generated test case. 111
Figure 39 – False positive test conflict caused by unrelated parent conflicting contributions. 112
Figure 40 – General behavior change caused by changes of the Right commit. . . . 116
Figure 41 – Generated test case detecting behavior change. 116

LIST OF TABLES

Table 1 – Summary of merge scenarios with build conflicts 40
Table 2 – Build error messages related to broken builds during the build process on Travis. 42
Table 3 – Catalogue of build conflicts 50
Table 4 – Catalogue of resolution patterns adopted to fix build conflicts. 56
Table 5 – Distribution of changed mutual class elements 95
Table 6 – Distribution of detected conflicts by unit test tools and binary files. Down arrows stand for conflicts detected by the current cell, but not detected by the next cell below. Up arrows stand for conflicts detected by the current cell, but not detected by the next cell above. Left arrows stand for conflicts detected by the current cell, but not detected by the next left cell. Right arrows stand for conflicts detected by the current cell, but not detected by the next right cell. Numbers between brackets represent false positives reported by the tools. *Pr.*, *Re.*, and *Ac.* stands for *precision*, *recall*, and *accuracy*, respectively. The values of *recall*, *precision* and *accuracy* are calculated based on the 28 conflicts with ground-truth present in our sample. 107
Table 7 – Distribution of behavior changes by unit test tools and binary files. Down arrows stand for behavior changes detected by the current cell, but not detected by the next cell below. Up arrows stand for behavior changes detected by the current cell, but not detected by the next cell above. Left arrows stand for behavior changes detected by the current cell, but not detected by the next left cell. Right arrows stand for behavior changes detected by the current cell, but not detected by the next right cell. The *Total* row stands for the union of behavior changes detected by all tools for each binary file version. 115

CONTENTS

1 INTRODUCTION 17

2 BACKGROUND 22
2.1 VERSION CONTROL SYSTEMS 22
2.2 SEMANTIC CONFLICTS IN PRACTICE 24
2.2.1 Integration Conflict Types 24
2.2.2 Build Conflicts in Practice 26
2.2.3 Test Conflicts in Practice 29
2.3 UNIT TEST GENERATION 31
2.3.1 Generating Test Suites with Unit Test Tools 32

3 BUILD CONFLICTS IN THE WILD 35
3.1 EMPIRICAL STUDY 35
3.1.1 Research Questions 36
3.1.2 Empirical Evaluation 38
3.1.2.1 Selecting Candidate Merge Scenarios 39
3.1.2.2 Removing Merge Scenarios Broken by Build System Issues 41
3.1.2.3 Classifying Conflicting Contributions and Resolution Patterns 42
3.2 RESULTS 47
3.2.1 RQ1: How frequently do build conflicts occur? 47
3.2.2 RQ2: What are the structures of the changes that cause build conflicts? 50
3.2.3 RQ3: Which resolution patterns are adopted to fix build conflicts? . 56
3.3 DISCUSSION 60
3.3.1 Findings 60
3.3.2 Implications 63
3.3.2.1 Guidelines for Developers 64
3.3.3 Build Conflict Repair Prototype 66
3.3.3.1 Fault Localization 66
3.3.3.2 Patch Creation 67
3.3.3.3 Patch Validation 68
3.4 THREATS TO VALIDITY 69
3.4.1 Construct Validity 69
3.4.2 Internal Validity 70
3.4.3 External Validity 72

4 TEST CONFLICTS IN THE WILD . . . 73
4.1 DETECTING SEMANTIC CONFLICTS . . . 74
4.1.1 SAM: SemAntic Merge tool based on Unit Test Generation . . . 74
4.1.1.1 Starting Point . . . 75
4.1.1.2 Selecting Mutual Changes on Same Class Elements . . . 76
4.1.1.3 Generating Binary Files . . . 77
4.1.1.4 Generating Test Suites . . . 77
4.1.1.5 Conflict Detection based on Merge Scenario Commits' Behavior Heuristic . . . 78
4.1.1.6 Report of Semantic Conflict Occurrene . . . 83
4.1.2 Testability Transformations . . . 83
4.1.3 Serialization . . . 85
4.1.4 Randoop Clean . . . 88
4.2 EMPIRICAL STUDY . . . 92
4.2.1 Research Questions . . . 92
4.2.2 Empirical Evaluation . . . 93
4.2.2.1 Mining and Selecting Merge Scenarios . . . 94
4.2.2.2 Building the Projects . . . 96
4.2.2.3 Generating and Executing Tests . . . 98
4.2.2.4 Detecting Interference . . . 100
4.2.2.5 Evaluating Improvements for Semantic Conflict Detection . . . 100
4.2.2.6 Analyzing the Scenarios and Establishing the Ground Truth . . . 102
4.3 RESULTS . . . 103
4.3.1 Cases with conflicts . . . 104
4.3.1.1 False Negatives . . . 109
4.3.2 Cases Without Conflicts . . . 111
4.3.2.1 False Positives . . . 111
4.3.2.2 True Negatives . . . 113
4.3.3 Further Evaluation of Tools and Related Test Suites . . . 114
4.3.3.1 Behavior Change Detection . . . 114
4.3.3.2 Test Tools' Relevance . . . 117
4.3.3.3 Target Code Reachability . . . 118
4.3.3.4 Object Diversity . . . 119
4.3.3.5 Code Coverage . . . 120
4.4 DISCUSSION . . . 121
4.4.1 Semantic Conflict Detection . . . 121
4.4.2 Improvements for SAM . . . 123
4.4.3 Improvements for Unit Test Generation . . . 124
4.5 THREATS TO VALIDITY . . . 126

5 RELATED WORK . . . 128

5.1 BUILD CONFLICTS 128
5.2 TEST CONFLICTS 133

6 CONCLUSIONS 139
6.1 CONTRIBUTIONS 140
6.2 FUTURE WORK 141

REFERENCES 143

1 INTRODUCTION

During collaborative software development, developers usually adopt branching and merging practices when making their contributions. These practices increase developer productivity by fostering teamwork, allowing developers to independently contribute to different software projects. Despite such benefits, branching and merging comes with costs, including the need of resolving conflicts that are detected by merge tools when integrating code changes. Depending on project characteristics (OWHADI-KARESHK; NADI; RUBIN, 2019; DIAS; BORBA; BARRETO, 2020), such *merge* conflicts might often occur (MENS, 2002; BIRD; ZIMMERMANN, 2012; KASI; SARMA, 2013; BRUN et al., 2013; MAHMOOD et al., 2020), even when using more advanced merge tools (APEL et al., 2011; APEL; LESSENICH; LENGAUER, 2012; CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY; BORBA; CAVALCANTI, 2018; CAVALCANTI et al., 2019; TAVARES et al., 2019; SHEN et al., 2019) that explore language syntax and static semantics to avoid spurious conflicts. In fact, high degrees of parallel changes and integration conflicts have been observed in a number of industrial and open-source projects that use different kinds of version control systems (PERRY; SIY; VOTTA, 2001; ZIMMERMANN, 2007a; BRUN et al., 2011; KASI; SARMA, 2013).

While many merge conflicts are easy to fix, some of them can only be fixed with significant effort and knowledge of the code changes to be merged. This can negatively affect development productivity, and even compromise software quality in case developers incorrectly fix conflicts (SARMA; REDMILES; HOEK, 2012; BIRD; ZIMMERMANN, 2012; MCKEE et al., 2017). Partially motivated by the need to reduce merge conflicts, or at least avoid large conflicts, development teams have been adopting risky practices, such as rushing to finish changes first (GRINTER, 1996; SARMA; REDMILES; HOEK, 2012), partial check-ins (SOUZA; REDMILES; DOURISH, 2003), trunk-based development (ADAMS; MCINTOSH, 2016; POTVIN; LEVENBERG, 2016; HENDERSON, 2017) and feature toggles (BASS; WEBER; ZHU, 2016; ADAMS; MCINTOSH, 2016; FOWLER, 2010; HODGSON, 2017), like adding extra conditionals and behavior variations leading to integration delays, which are important to support actual continuous integration (CI) (FOWLER, Sep 2009), but might lead to extra code complexity (HODGSON, Oct 2017).

Although these practices might reduce the occurrence of merge conflicts, there is no evidence that they are effective for resolving or even detecting other types of conflicts, like *semantic conflicts* commonly known as *build* and *test* conflicts, that are conflicts revealed by failures when building and testing integrated code, respectively (KASI; SARMA, 2013; BRUN et al., 2013).[1] While build conflicts involve syntactic and static semantic failures

[1] This relates two conflict terminologies; one based on the development phase in which a conflict is detected, and the other based on the language aspect that causes a conflict. We use *merge* conflict and *textual* conflict as synonyms. *Build* conflict refers to *syntactic* and *static semantic* conflicts. *Behavioral semantic* conflict refers to *test* and *production* conflicts (and undetected ones). For brevity, hereafter we omit the

usually detected during the build process, test conflicts involve behavioral semantic failures relying their detection on the quality and coverage of project test suites. Due to this limitation, semantic conflicts can be escaped for the system in production leading to *production* conflicts. These conflicts are harder to detect as they are only observed when using the system in production. As such, semantic conflicts are more serious, because they reveal build and software failures. In fact, some of the practices mentioned above might even aggravate the costs of semantic conflicts.

Resolving merge conflicts is often simpler, because it mostly involves reconciling incompatible independent *textual* changes in the same area of a file. Semantic conflicts are harder, especially when resolution occurs long after conflict introduction, because they involve reconciling *static* and *behavioral semantic* incompatibilities—as when the changes made by one developer affect a state element that is accessed by code contributed by another developer, who assumed a state invariant that no longer holds after merging. Although evidence in the literature is mostly limited to merge conflicts (ZIMMERMANN, 2007a; APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015; ACCIOLY; BORBA; CAVALCANTI, 2018) and other kinds of build errors (HASSAN; ZHANG, 2006; VASSALLO et al., 2017; RAUSCH et al., 2017), previous studies investigate the frequency of semantic conflicts (BRUN et al., 2011; KASI; SARMA, 2013). As these two studies observe conflicts in a small number of projects, three and four, respectively, it is important to observe conflict *frequency* in a larger context, and analyze and classify unexplored aspects such as conflict *structure* and adopted *resolution patterns*, when possible. Better understanding these aspects might help us to derive guidelines for avoiding semantic conflicts, improve awareness tools to better assess conflict risk, and develop new tools for automatically fixing conflicts. So in this thesis, we investigate the occurrence of semantic conflicts, their causes, and how our results could support developers when facing these conflicts during software development.

In this way, to fill the gap in the literature regarding semantic conflicts, we deeply investigate merge scenario integrations exploring the changes performed by developers' contributions. As a result, we could understand how these individual contributions conflict with each other leading to semantic conflicts. To address build conflicts, we perform an empirical study investigating the frequency, causes and resolution patterns for these conflicts. In summary, our results bring evidence about build conflict frequency complementing findings from previous studies. Additionally, we report catalogues of causes and resolution patterns that can be used to improve assistive tools as also to propose an automatic repair tool to automatically resolve these conflicts, as we propose here. In order to evaluate the occurrence of test conflicts, we adopt a different approach as these conflicts involve the semantics of a program. Consequently, they can not be detected during the compilation phase of the build process. This way, initially, we perform a second empirical study investigating the detection of these conflicts using unit test regression. Our initial

"behavioral" term in spite of focusing only on *behavioral semantic* conflicts in this work.

results bring evidence about the potential of using unit test generation tools to detect conflicts, using a new approach based on unit tests as partial specifications. Additionally, we report a list of improvements that must benefit unit test tools during the test suite generation. As a final contribution, we present SAM, our semantic merge tool based on unit test generation, which warns developers abour proeminent test conflict on ongoing merge scenarios. Next, we present in more details our just mentioned results.

Regarding build conflicts, we investigate the frequency, structure, and adopted resolution patterns. For that, we analyze merge scenarios from 451 GitHub open-source Java projects that adopt Travis CI. To collect build conflicts, we select merge scenarios from the git repositories and check through Travis services whether the merge commit build is broken (*errored* status). To make sure the breakage is caused by the integrated changes, we parse the Travis logs generated when building the commits in a scenario, and automatically check whether the logged error messages are related to the merged changes; or our scripts confirm conflict occurrence by observing whether the parent commits present superior status (*failed* or *passed*) to the merge commit, and whether this commit contains only the integrated changes.

We find and classify 239 build conflicts, deriving a catalogue of build conflict causes spread in six categories. For conflicts with associated fixes, we analyze how developers fixed them, deriving a catalogue of common conflict resolution patterns. The frequencies in our catalogue of build conflicts show that most build breakages are caused by missing declarations removed or renamed by one developer, but referenced by the changes of another developer. Moreover, the conflicts caused by renaming are often resolved by updating the dangling reference, whereas the conflicts caused by deletion are often resolved by reintroducing declarations. To resolve build conflicts developers often fix the integrated code, instead of completely discarding changes and conservatively restoring project state to a previous commit.

Based on the catalogue of build conflict causes derived from our study, awareness tools could alert developers about the risk of a number of conflict situations they currently do not support. Program repair tools could benefit from the derived catalogue of build conflict resolution patterns, and the study infrastructure, to automatically fix conflicts. We illustrate that with a proof of concept implementation that recommends fixes for build conflicts caused by a developer deleting the declaration of a variable that is referenced by the changes of another developer.

Next, we investigate test conflicts by conducting another empirical study when we assess the detection of conflicts using unit test generation. The core idea we propose and assess is to use generated tests as partial specifications of the parent contributions. As a result, we could also assess to what extent a semantic merge tool could *rely on unit test generation to reveal interference* between developers' changes, say commits in branches, that should be merged. To evaluate the potential of unit test generation to reveal interference, we

apply widely recognized test generation tools (EvoSuite (ALMASI et al., 2017; FRASER, 2018) and Randoop (PACHECO et al., 2007)) to a sample of 51 merge scenarios of Java projects, in which integrate changes to the same method, constructor or field declarations, as when two developers independently change the same methods and later integrate the changes. For each merge scenario, we invoke EvoSuite (the standard and the differential version), Randoop, and Randoop Clean (our new proposed unit test tool). Each unit test tool is called with different binary file versions, like the original, slightly transformed and serialized versions (Serialization with Testability Transformations). Then, we check their potential in detecting conflicts based on our test-based criteria. These transformations are changes directly applied in the original source code aiming to increase its testability during the generation process of test suites, while the adoption of serialization aims to support tools when creating complex required objects.

Our results show that, by first applying the transformations, the test generation tools can detect interference in nine out of 28 changes on same declarations (in three merge scenarios) that in fact suffer from interference between the integrated changes. Although this results in a small true positive rate, the generated tests lead to only a few cases of false positives according to our interference criteria. This suggests that semantic merge tools based on unit test generation, as we propose, can help developers to detect semantic conflicts early, that would otherwise reach end users as failures. The Testability Transformations improved testability in three of the nine detected interference cases, suggesting that they might be useful for interference detection. Our results show that the best approach to detect conflicts involves combining the tools Differential EvoSuite and EvoSuite applied with Testability Transformations, as together they detect all reported conflicts. This way, we propose and present SAM, our semantic merge tool based on unit test generation tools using tests as partial specifications.

Manually analyzing test suites for false-negative cases, we identify and categorize improvements that could benefit unit test generation tools. The identified improvements reflect three main problems highlighting the need for creating relevant objects required for the declarations holding the conflict, and relevant assertions exploring the *propagated* interference. Regarding the detection of General Behavior Changes, EvoSuite was the most successful tool detecting 53% of all reported changes. Although the adoption of Serialization did not lead to detect new semantic conflicts, that approach benefits the detection of behavior changes, not detected by the tools when applied on original and transformed binary files.

In summary, in this thesis, we aim to fill a gap in the literature regarding semantic conflict on software development. To achieve our goal, we deeply investigate merge scenario integrations to understand how these contributions conflict with each other leading to semantic conflicts. This way, we expect that our findings may benefit researchers and software practitioners when facing semantic conflicts in practice. We believe our findings

are a basis for further research exploring other fields on semantic conflicts like identifying factors associated with their occurrence, measuring their impact on software development, and development of assistive tools. Regarding build conflicts, we believe our catalogues of causes and resolution patterns can be used as basis for the development or improvements of assistive tools to detect and resolve these conflicts. In this way, we provide an automatic repair tool prototype that suggests fixes for build conflicts. From our initial results of test conflicts, we believe they represent an initial step to understand and detect this kind of conflict. Further research must be done exploring other ways that test conflicts may occur. However, we believe our initial results detecting conflicts using unit test generation and unit test as partial specifications can be used as the basis for the development of a semantic merge tool, as we propose here.

The results and mentioned studies presented in this thesis are partially first reported as research papers. In this way, each study is presented in a chapter that includes the corresponding article manuscript format and content. The remainder of this document is organized as follows:

- In Chapter 2, we present the fundamental concepts used throughout this work;
- In Chapter 3, we present our first empirical study investigating the frequency, causes, and resolution patterns of build conflicts. Based on our results, we propose an automatic repair tool to resolve build conflicts that we illustrate with an initial prototype. This chapter is published Da Silva *et al.* (2022), with the co-authoring of Paulo Borba, who reviewed and guided the work, and Arthur Pires, who helped with implementing an automatic repair tool prototype.
- In Chapter 4, we present our second empirical study assessing the detection of test conflicts using regression testing and unit test generation. A previous version of this chapter is published in Da Silva *et al.* (2020) in collaboration with Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. Paulo Borba reviewed, guided and also helped with the extensive manual inspections necessary in this work. Wardah Mahmood and Thorsten Berger reviewed and guided this work, while João Moisakis helped with the instrumentation and the testability transformation scripts. The new features added on Randoop, as also the implementation of the tools Randoop Clean and OSean.EX was done in collaboration with Antonio Maciel. The scripts responsible for collecting source code coverage were implemented in collaboration with Vinícius Leite. Finally, SAM was implemented in collaboration with Aldiberg Gomes.
- In Chapter 5, we summarize our findings considering previous studies when we discuss related work.
- In Chapter 6, we sum up our concluding remarks, contributions, and future work.

2 BACKGROUND

In this chapter, we present the main concepts used when doing this thesis. Initially, in Section 2.1, the fundamentals of version control systems (VCSs) are discussed. Next, in Section 2.2, we briefly present how different contributions are integrated during collaborative software development, and then, in the Sections 2.2.2 and 2.2.3, we discuss the fundamentals and the different types of semantic conflicts that we investigate here, *build* and *test* conflicts, respectively. Finally, in Section 2.3, we briefly discuss initial concepts regarding unit test generation.

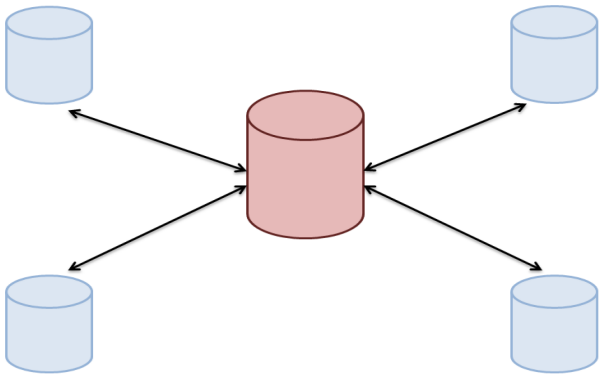
2.1 VERSION CONTROL SYSTEMS

During collaborative software development, it is expected that different developers contribute to the same project by working on individual copies of shared project files. Although this practice is commonly adopted, it is only possible thanks to software configuration management (SCM), and consequently, version control systems (VCSs), as they provide the required mechanisms to manage the evolution of large and complex systems (TICHY, 1988). In this way, motivated by the need to keep the project updated and track the changes performed over time, SCMs provide this support to software development with tools and techniques to deal with the artifacts' evolution, especially when this evolution includes parallel work of several developers (CONRADI; WESTFECHTEL, 1998). In practice, VCSs allow developers to download and change files on your private repository copies and then synchronize it with the main version of the repository.

Different paradigms of VCSs may be adopted, that are classified based on the way information can be accessed and delivered for all involved in a project. In Centralized Version Control Systems (CVCS) paradigm, one central repository represents the unique server holding the information and getting the associated change history (Figure 1). Considering the software development context using a CVCS repository, at the beginning of tasks, it is expected that developers synchronize their private copies from the central repository to get the most updated state of the project and then start making their contributions. After finishing their tasks, they are back integrated into the central repository aiming to keep the current development state stable and accessible for all involved. Examples of services implementing this paradigm include Subversion (SUBVERSION, 2020) and CVS (CVS, 2020).

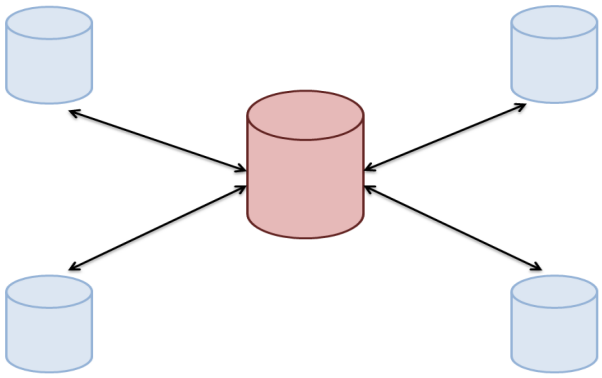
In Decentralized Version Control Systems (DVCS), developers use no central repository to synchronize changes from and to their private copies. All private repository copies are sources of information and associated change history for any other contributor (Figure 2). Considering the software development context again, but this time using a DVCS

Figure 1 – Centralised version control paradigm. The arrows among the individual (blue) and central (red) repositories indicate that only the central repository can accept others' changes. Individual repositories use the central one to update themselves with new information (synchronization process). Changes from individual repositories are only accessible if the central repository already holds them.



Source: The author (2022).

Figure 2 – Decentralised version control paradigm. The arrows among the individual repositories (blue) indicate each repository can accept changes and update themselves from the others. Individual repositories are accessed directly without any intermediary repository.



Source: The author (2022).

repository, contributions performed by a developer can be accessed and shared with others directly without an intermediary communication to another repository. This paradigm has become popular in software development, especially in the open-source community, because of its simplicity in propagating information in many directions and directly among developers (BRINDESCU et al., 2014; GOUSIOS; PINZGER; DEURSEN, 2014). Git (GIT, 2020) and Mercurial (MERCURIAL, 2020) are DVCS examples highly used in practice. Particularly for git, different services support it and are available, such as Bitbucket and GitHub.

In both paradigms, the evolution of software artifacts is achieved by integrating different contributions performed into a project over time. Different kinds of conflicts may arise during such integration, like *merge*, *build*, and *test*, directly impacting team productivity and software quality. Investigating and understanding these conflicts should benefit software practitioners as they might provide insights and improvements for assistive tools. For example, new causes of build conflicts can reflect new features for tools like Palantír (SARMA; REDMILES; HOEK, 2012). In the same way, merge tools could benefit from these findings, bringing insights into how to treat build and test conflict during contribution integration.[1] For instance, improved merge tools (APEL et al., 2011; CAVALCANTI et al., 2019) are already able to detect *duplicated method declarations* in a class that, if not detected during the merge scenario, could lead the build process to fail.

2.2 SEMANTIC CONFLICTS IN PRACTICE

Before directly getting into the kinds of conflicts we investigate in this thesis, it is important to discuss some fundamental concepts related to the integration process. So in this section, we illustrate this integration process exploring the different conflict types that may arise.

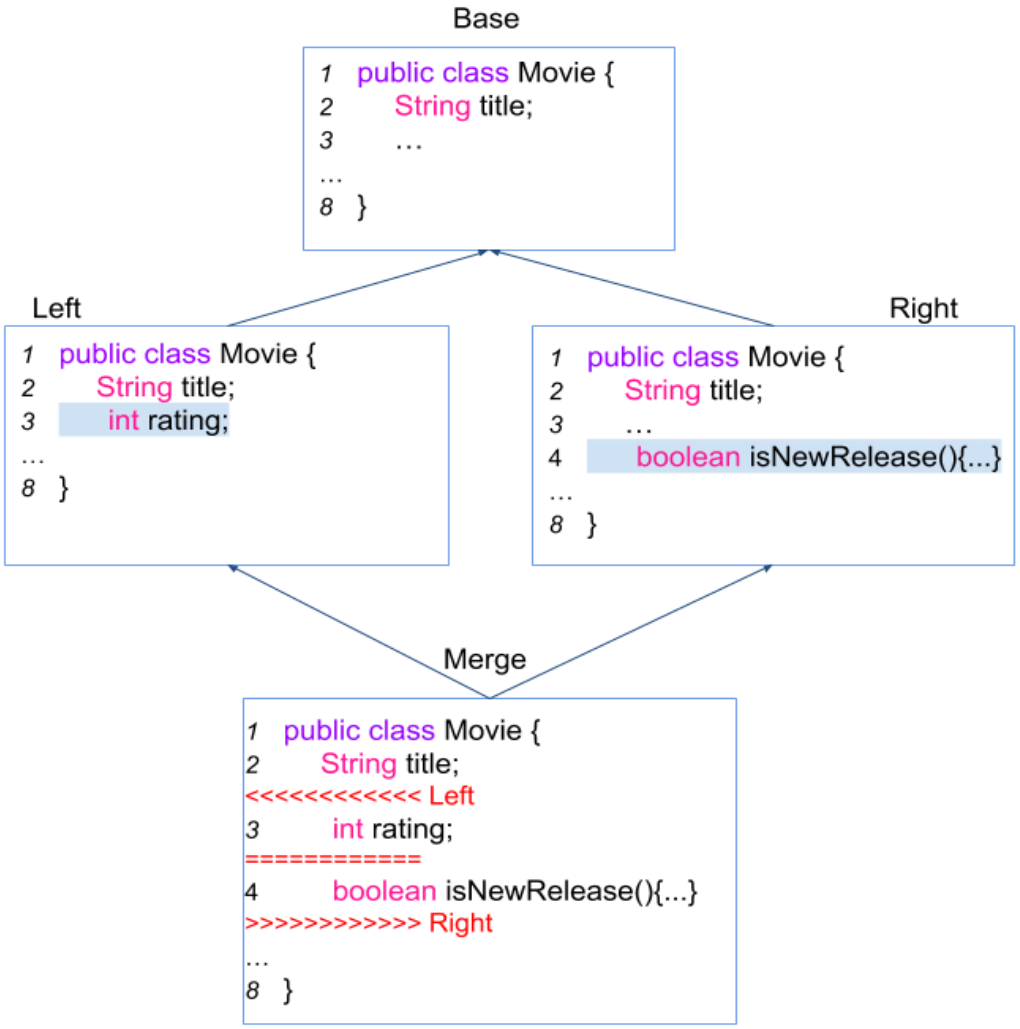
2.2.1 Integration Conflict Types

As previously motivated in Section 2.1, during collaborative software development it is expected that different developers simultaneously contribute to the same project. As a result, individual contributions are performed, and in the end, they are required to be back integrated in the project, process known as a *merge scenario*. Usually, a merge scenario is represented as a triple formed by two parent commits, *Left* and *Right*, integrated into a new commit, *Merge*. These parent commits result from individual contributions performed based on a common ancestor commit, *Base*, represented as the point where the parent commits start to diverge with each other.

An example of a merge scenario can be seen in Figure 3. In this scenario, the developer *Left* adds the new attribute `rating` for the class Movie, while the other developer *Right* introduces the new method declaration `isNewRelease` in the same class. Although the contributions are not performed on the same but consecutive lines, lines 3 and 4, respectively, *merge* conflicts arise, also know as *textual* conflicts, and the developer doing the merge must resolve the conflicts to continue the integration process. As we previously discussed, resolving conflicts might be time consuming and is an error-prone activity (SARMA; REDMILES; HOEK, 2012; BIRD; ZIMMERMANN, 2012; MCKEE et al., 2017). An option to avoid manually dealing with merge conflicts is the adoption of improved advanced merge tools.

[1] The terminology *test conflicts* stands for changes on class files that lead the system to unexpected behavior. So there is no relation with conflicts, textual or other types, involving test class files.

Figure 3 – Merge scenario with merge conflict.



Source: The author (2022).

For example, the S3M tool resolves the discussed merge conflict without requiring the developer intervention (CAVALCANTI; BORBA; ACCIOLY, 2017). However, merge conflicts are not the only kind of conflict that may happen during a merge scenario.

Conflicts may arise in different phases of the merge scenario and not only at the syntactic but also semantic level known as *semantic* conflicts. For example, after the integration of contributions, the attempt to build the integrated code may fail caused by *syntactic* or *static semantic* problems revealing a *build* conflict. In another case, during the testing phase, the integrated code presents an unexpected behavior caused by *behavioral semantic* problems leading the project test cases to fail revealing *test* conflicts (KASI; SARMA, 2013). In the next sections, we explain the nature of semantic conflicts that we investigate here illustrating how they occur.

2.2.2 Build Conflicts in Practice

To illustrate a build conflict occurrence, consider an adapted example from the Quickml project.[2] Hypothetical developers Lucas and Rachel are assigned different, but related, tasks. They start working on their private repositories, which are updated with respect to the main project repository (illustrated in Figure 4). Assuming the latest commit in the repository is *C0*, Rachel finishes her work creating commit *C1*. At this point, she successfully builds and tests her version of the project (build process) and immediately sends her contribution to the main repository.

Later, Lucas finishes his work creating commit *C2*. He also makes sure that his changes do not break the build, successfully building and testing the project at state *C2*. Nevertheless, before sending his contribution to the main repository, Lucas notices Rachel's updates in *C1*. By quickly inspecting that, he is relieved because Rachel changed a disjoint set of files and, consequently, he will not need to fix merge conflicts. He then rushes to send his contribution to the main repository, creating a merge commit *C3* (upstream).

Before starting his new task, Lucas updates his private repository, checks the new commit *C3*, and decides to run the system to have a look at the new functionalities added by Rachel. He is then worried to see error messages after trying to build the project, and realizes that the project main repository is in an inconsistent state.

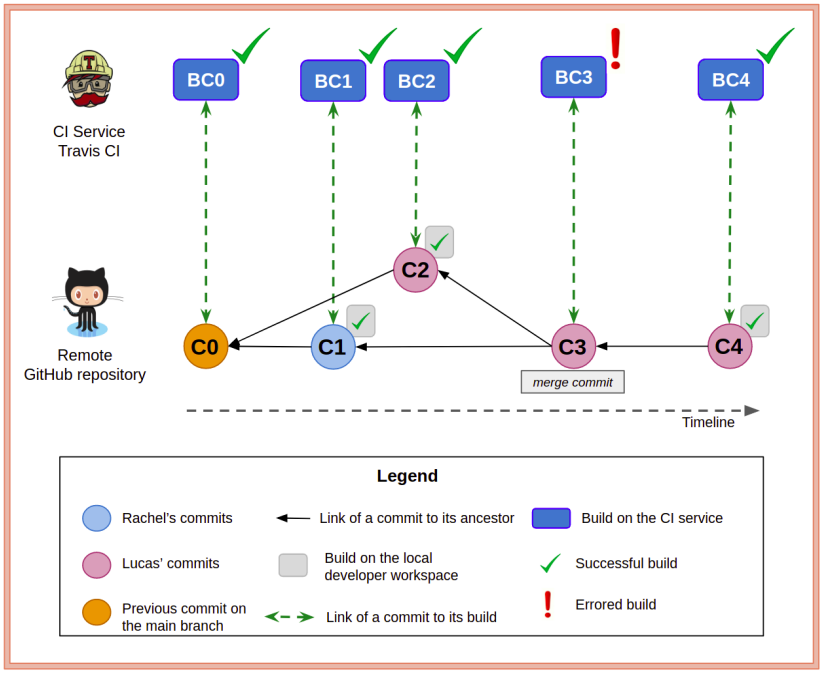
By talking to Rachel and confirming that the build process failure (*errored status*) observed for *C3* was not directly inherited from a defect in *C1* or *C2*— builds were fine for both commits—, the developers suspect the changes from one of them unintendedly interfere (CAVALCANTI; BORBA; ACCIOLY, 2017; SHAO; KHURSHID; PERRY, 2007) with the changes of the other. Trying to confirm the interference, Lucas checks the broken build log. He observes that a method call for `ignoreAttributeAtNodeProbability`, in the `StaticBuilders` class, is the source of a compilation error because its declaration is missing in the `TreeBuilder` class. Investigating the `TreeBuilder` class, he confirms the method is not declared. However, he is sure this method declaration was available when he was working on his task and added the now problematic method call. Consulting the project history, Lucas notices that Rachel, unaware of Lucas' task, renamed the `ignoreAttributeAtNodeProbability` method in *C1*. He then realizes that the changes interfere, causing a *build conflict*, that is, a build breakage in a merge commit, caused by interaction among integrated changes.

> **Build Conflict** refers to a syntactic or static semantic problem causing a build breakage detected during the compilation process of a merge commit.

To fix this build conflict, Lucas changes the method call using the new method name

[2] Build ID sanity/quickml/53571613, Merge Commit 62a2190.

Figure 4 – Build conflicts in practice. On the bottom, blue and pink circles represent commits done by Rachel and Lucas, respectively. In contrast, the orange circle represents the previous commit in the main branch before the described merge scenario. Above some developers' commits, gray squares represent the build processes done locally on the developers' workspace (✓, for successful builds). Black arrows link commits to their ancestors. Finally, on the top, green arrows associate commits from the remote repository with their build processes (blue boxes) on the Continuous Integration (CI) service. Above each build, we present its status (✓ and !, for successful and errored builds, respectively). For simplicity, we do not present commits done on developers' private repositories.



Source: The author (2022).

`attributeIgnoringStrategy`. He now successfully creates an executable build restoring the main repository consistency by sending a conflict fix (commit *C4*). A more experienced developer would maybe not have rushed as Lucas did, first pulling Rachel's changes to her private repository, merging them, and making sure she can successfully build and test the integrated code. Adopting this strategy would have avoided leaving the main repository in an inconsistent state, but would not avoid the conflict nor the effort to resolve it. The developer would have to stop her work, understand the problem, and find and fix the conflict. Similarly, a more prudent team would maybe have a continuous integration service running on the main repository, and adopt a pull-request based contribution model. This contribution model would avoid the inconsistent state, and help to earlier detect the conflict, but would not necessarily avoid it. In the discussed merge scenario, we illustrate the occurrence of a single build conflict. However, a single merge scenario may have several conflicts caused by different types and changes, requiring specific treatment for each one.

Our motivating example presents a specific context in which build conflicts may occur.

However, there are other possible contexts and associated consequences, in practice. For example, conflicts may affect productivity and limit the access to resources shared by the development team. Consider a build process that takes a long time to complete, and the build breaks caused because of a build conflict. Despite the time spent by the developer for finding and fixing the conflict, the failed build process held resources that might be relocated for other build processes. Although the discussed build conflict appears when different branches of a single project are integrated, these conflicts are not exclusive to integrations involving a single project. Sung *et al.* (2020) also report build conflicts occur when different projects are integrated. In their study, conflicts occur when developers pull changes from the main project (Chromium) into another project (Edge). Since both involved projects are in conflicting different states, build conflicts are reported.

In our motivating example, we present a build conflict caused by a merge scenario remotely performed, when the developers have the required permissions to directly update the main repository. However, a build conflict may also happen when the development team does not own these permissions. For example, during the acceptance of a *pull request* (PR) on GitHub, the current repository state may be conflicting with the PR changes. In case the repository adopts CI services like Travis CI, the broken build would be reported on the GitHub PR page, and the conflict could be easily detected. Thus, the integrator would deal with the conflict or request changes to the contributors until the problem is fixed. Otherwise, the PR could be accepted, polluting the main repository.

We believe that for projects adopting CI service and a pull-based development process, GitHub invokes Travis services and annotates the pull request with the resulting build information, right after a developer submits a pull request. In case there is a build problem, this is clearly indicated in the main pull request page at GitHub. Reviewers, noticing a build problem in the pull request, might prefer to delay revision until the build issue is solved. In this study, we do not investigate code review comments, as our dataset contains only commit information. Based on the merge commit information, we could likely retrieve pull request information from GitHub, that is, pull requests that include the merge commit under analysis. This way, we would also have to consider that not all merge commits were integrated to the main repository through pull requests. To understand the possible information we could extract from PRs discussions, we decided to randomly pick up 10 merge scenarios of our sample that were integrated through PRs. In all of them, when there is a discussion, the reviewers and PR authors discuss general ideas of the PR proposal, not specifically build breakages. We also observe that in some cases the PR build is stable (passed), but before the PR acceptance, new changes are integrated into the main repository leading the PR build to break after the integration. Reinforcing that, Zampetti *et al.* (2019) report that when a build process of PR breaks due to compilation problems, reviewers discuss very little about the related causes, as these generic errors could be fixed through private builds. Last, it is essential to highlight that build conflicts

would only be avoided by not simultaneously performing conflicting tasks. Otherwise, a conflict would be locally or remotely detected at some point.

Someone may argue that PR integration could be prioritized aiming to reduce the chances of conflicting integrations (VEEN; GOUSIOS; ZAIDMAN, 2015). However, even in such circumstances, build conflicts might be earlier detected but not avoided. Build conflicts do not exclusively occur when changes are integrated upstream, resulting in conflicts reaching the remote repositories that we investigate here. As a result, the previous mentioned breakage state would be observed when a developer locally updates her private repository with the upstream and builds the project. However, in this case, once the conflict is detected, the developer is expected to locally apply changes to fix the conflict and push them to the remote repository. If these activities of detecting and fixing conflicts frequently happen, resolving them end up being a tedious and error-prone activity (SUNG et al., 2020).

Although our motivating example has been simplified, it illustrates the kind of conflict we consider in our study, and how they might impair team productivity. However, our conflict classification is programming language type-specific since a single conflict cause may arise at different situations relying on the language types. Whereas our discussion makes sense for a Java or C++ project, in a Ruby or Python project, for example, the illustrated conflict would not be revealed during build time. Due to Ruby dynamic features, there is no pre-execution check about the existence of method declarations. The conflict would be revealed only during testing or system execution in production.

2.2.3 Test Conflicts in Practice

To illustrate the notion of behavioral test conflict we also explore in this thesis, consider the example in Figure 5. The illustrated class `Text` results from a merge that integrates the change in green (Line 6 was added, say from the *Left* commit) with the change in red (Line 8 was added, say from the *Right* commit). This example is inspired by a real merge commit from the project Jsoup.[3] The other code lines originate from the *Base* commit, that is, the most recent common ancestor of *Left* and *Right* commits.[4] As the method call in Line 7 separates the two changes to be integrated, there is no syntactic merge conflict in this case, and we cleanly obtain the syntactically valid class in the figure. We can then compile, build, and execute it.

The intention of the developer who created the *Left* commit, say developer Rachel, was to extend the method `cleanText()`, which does some string cleaning through side effects, to also remove duplicated whitespace in the text by adding the method call `normalizeWhitespace()`, since the goal of her development task was to make sure that the resulting text had no such duplications. The intention of Lucas, who created the *Right* commit, was

[3] https://github.com/jhy/jsoup/commit/a44e18a

[4] For simplicity, we assume a single most recent common ancestor. With so-called criss-cross merge situations in git, there could be more than one.

to clean the text by removing consecutive duplicated words as in the string "the␣the␣dog." Lucas, however, was not aware of the task allocated to Rachel and implemented the removal of duplicated words without eliminating whitespace between them, resulting in "the␣ ␣dog" for our string example.

This shows that, although Rachel's implementation is correct (it conforms to the implicit specification it is supposed to satisfy), the resulting method `cleanText()` we obtain after the merge does not fully eliminate duplicated whitespace from the text, which will certainly surprise Rachel. In fact, the implicit specification "resulting text has no duplicated whitespace" that is individually satisfied by the *Left* commit is not satisfied by the *Merge* commit that we illustrate in Figure 5. So, we say that *Left* and *Right* commit changes *semantically conflict*—or *interfere* in an unintended way—with respect to the *Base* commit. Lucas's implementation is also correct, assuming he was not required to eliminate whitespace between words, but ends up unexpectedly interfering with Rachel's implementation. Notably, we do not observe *interference* in the opposite direction, as the resulting merged code fully eliminates duplicated words—the implicit specification implemented by Lucas holds in *Right* and *Merge* commits.

As current merge tools cannot detect such *behavioral semantic conflicts*, it is often difficult and expensive to detect and resolve them. In case the project offers a test suite with high coverage and quality, it is possible a test case detects the unexpected behavior revealing the *test* conflict in this example.

> **Test Conflict** refers to a *behavioral semantic* problem causing unexpected software behavior detected by reported failed tests during the project test suite execution of a merge commit.

Figure 5 – A merge of two changes (each parent added one of the highlighted lines) that are semantically conflicting.

```
1   class Text {

3     public String text;

5     void cleanText() {
+       this.normalizeWhitespace();
7       this.removeComments();
+       this.removeDuplicatedWords();
9     }
    }
```

Source: The author (2022).

In fact, unless a project adopts careful code review practices and, as just previously mentioned, has strong test suites, most semantic conflicts are expected to escape to users. In such cases, these semantic conflicts are classified as *production conflicts* as they might be detected with the system in production when a user of the system reports the unexpected

behavior. However, it is also possible that behavioral semantic conflicts may never be revealed once the user never uses the feature holding the conflict.

2.3 UNIT TEST GENERATION

Test generation is a technique used to automatically generate tests for a specific program. As a result, it is expected the generated tests be able to detect unexpected behavior when executing them on modified versions of the input program (KOREL; AL-YAMI, 1998). Different tools have implemented this idea providing support for many programming languages. For example, for Java, to name a few, Randoop (PACHECO; ERNST, 2007) and EvoSuite (FRASER; ARCURI, 2012) are widely recognized tools.

The process required to generate these tests is divided into two steps: *test setup* and *asserts*. While the first step is responsible for creating, initializing, and exercising objects required for the test, the second step is responsible for verifying the test's expected results.

Regarding the first step, Randoop does it by randomly generating *sequences* of calls, and selecting the methods and constructors in the class under test. The arguments for such operations are also randomly selected from a pool of primitive type values and objects previously created in a sequence. A sequence can group one or more *statements* represented as method calls or variable declarations in a Java program. Each statement has three elements: a call for a method or constructor, a value returned by each call, and call inputs that are references for previous generated statements.

To generate sequences, Randoop provides three options to do it.[5] The first option is by *extending* a previous sequence that consists of adding a new operation at the end of the current sequence. Another option is by *concatenating* a set of prior sequences generating a new sequence grouping all input sequences. Finally, the last option is by *parsing* a String received as input and converting it into a new sequence.

EvoSuite also starts from randomly generated test suites, as in Randoop, but relies on genetic algorithms to evolve the test suites in order to optimize a specific goal, such as higher code coverage. This way, each test suite is evaluated based on a fitness criteria, and next, the fittest suites suffer mutations or crossovers generating new tests. This process is repeated until EvoSuite finds a test suite that satisfies the target fitness criteria or the time budget is over.

Concerning the generation of asserts, Randoop generates *checks*, which are pieces of code responsible for verifying an expected property returned by executing the method sequences. In a Java program, for example, a check is an *assertion* in a test case. Checks can be adopted in different ways to detect unexpected behavior. For example, a check can verify whether a mandatory expected or unexpected exception is thrown during the test case execution. Another way is by checking specific states hold by an object. As a

[5] https://randoop.github.io/randoop/manual/dev.html

check is always associated with a specific index of a sequence, it is mandatory that the check be executed just after the associated sequence. So if there is a check at the index i, this check must be performed just after the end of the i-th sequence. EvoSuite adopts a similar approach exploring the values returned by executing method sequences. EvoSuite can further calculate a reduced set of the generated assertions, whereas Randoop can also generate assertions that check basic and general contracts. A contract expresses invariant properties that hold both at entry and exit from a call; it checks whether the resulting call values conform with its specification.

2.3.1 Generating Test Suites with Unit Test Tools

Aiming to present a practical use of a unit test tool, consider a test suite generated with Randoop. As previously mentioned, the first step when calling these tools is to provide the source code used during the generation process. In this context, source code stands for the Java bytecode of the program under analysis (.class files) or an executable (jar file), and all dependencies required to run the program. Next, the tools may ask for a specific target code, which will drive the generation of tests; for example, a full class name, and in some cases, a related method signature. Regardless the mentioned details of required dependencies, consider the class `Text` presented in Figure 6 as the given class name and its method `cleanText()` (line 4).

Figure 6 – Java class given as input for unit test generation tool.

```
class Text {
2     public String text;

4     public void cleanText(){
         this.normalizeWhitespace();
6        this.removeComments();
         this.removeDuplicatedWords();
8     }

10    private void normalizeWhitespace(){...}

12    private void removeComments(){...}

14    private void removeDuplicatedWords(){...}

16    public String updateCodeFormat(){...}

18    public boolean noDuplicateWhiteSpace(){...}
  }
```

Source: The author (2022).

Once Randoop owns the required information, the tool collects all public methods of the target class and saves them in an object of available methods called *pool*. In our example, this pool will be fill up by the methods `cleanText()`, `updateCodeFormat()`, and `noDuplicateWhiteSpace`. Since the method `cleanText()` was given as input, Randoop will try to generate tests that cover the required method, though there is no guarantee the

tool will achieve this goal. In contrast, the method `updateCodeFormat()` will be covered by test cases if that method is selected during the generation, which is a random process.

Next, Randoop starts to generate *sequences* of calls and randomly selects methods from the `pool`, as previously mentioned. So consider the method `cleanText()` is selected by Randoop. First, the tool checks whether there is a valid previously generated object of type `Text`. If so, this object will be used to call the method `cleanText()`; otherwise, a new object of type `Text` is required to be created. In the same way, if the selected method requires parameters to be used in its call, the same actions are performed. However, the target method does not require any parameter in this particular case, so that the method call can be performed without further checks.

Once the method is selected and required objects are available, the method call can be executed. After the method call execution, Randoop evaluates whether the call results in a successful call. If so, the method call can be used as a valid test case on test suites; otherwise, the method call is discarded. A successful sequence defined by Randoop stands for no *contract violation*. For example, if the target method call receives as a parameter a `String`, a contract violation, in this case, would be receiving an `Integer` instead of the required type. As a result, the method call would not be properly executed due to errors caused by the generation process. An example of a valid generated test class is presented in Figure 7.

Figure 7 – Test class generated by Randoop based on given target class and method name.

```
1  class TextTestSuite {

3    public void test1() throws Throwable {
       Text t = new Text();
5      t.text = "the house is blue";
       t.cleanText();
7      assertTrue(t.noDuplicateWhiteSpace());
     }
9
     public void test2() throws Throwable {
11     Text t = new Text();
       assertEquals("UTF-8", t.updateCodeFormat());
13   }

15 }
```

Source: The author (2022).

The test case `test1` in Figure 7 presents a direct call to the target method `cleanText()` (line 6), while it also presents a call for `noDuplicateWhitespace()` (line 7). Calling and combining different methods in the same test case is the expected behavior of Randoop. Since Randoop randomly selects methods from a *pool* of available methods, different combinations can be achieved. As the last method call is not `void` returning a `boolean` object, Randoop uses its value to generate an assertion. This assertion informs that during the generation process a `true` value was returned after calling `noDuplicateWhitespace()`;

so that is the reason for the use of an assertion expecting a `true` value.

In the same way, the test case `test2` in Figure 7 presents a direct call to another public method, `updateCodeFormat()` (line 12). As this method returns a `String` (see Figure 6 line 16), Randoop generates an assertion that explores the observed value returned by the method call. So based on the object types handled by the tool during the generation process, different assertion types can be generated to explore them.

Although Randoop can not directly call the private methods of the target class `Text`, all of its private methods were called due to the internal calls done by the method `cleanText()`. It is essential to mention that given the random behavior of Randoop, it is not expected that the generated test suites will cover all methods. Another point to consider is the complexity of the target class given as input. Since the class `Text` does not have complex objects as *fields* or *parameters*, Randoop was able to generate objects of the required types and directly explore all of its public methods.

3 BUILD CONFLICTS IN THE WILD

In this chapter, we initially present our investigation of the occurrence, causes and resolution patterns of build conflicts, and then our proposal of automatic repair tool to resolve these conflicts. As previously discussed, when collaborating, developers often create and change software artifacts without being fully aware of other team members' work. While such independence is essential for increasing development productivity, it might also result in conflicts when integrating developers code contributions. To better understand some of these conflicts— the ones revealed by failures when building integrated code— we investigate their frequency, structure, and adopted resolution patterns in 451 open-source Java projects. To detect such build conflicts, we select merge scenarios from git repositories, parse the Travis logs generated when building the commits, and check whether the logged build error messages are related to the merged changes. We find and classify 239 build conflicts and their resolution patterns. Most build conflicts are caused by missing declarations removed or renamed by one developer but referenced by another developer. Conflicts caused by renaming are often resolved by updating the missing reference, whereas removed declarations are often reintroduced. Most fix commits are authored by one of the contributors involved in the merge scenario. We also detect and analyze build failures caused by immediate post integration changes, which are often performed with the aim of fixing merge conflicts but end up leading to build issues. Finally, based on our catalogue of build conflict causes, awareness tools could alert developers about the risk of conflict situations. Program repair tools could benefit from our catalogue of build conflict resolution patterns to automatically fix conflicts; we illustrate that with a proof of concept implementation of a tool that recommends fixes for conflicts.

The rest of this Chapter is organized as follows: Section 3.1 explains our study setup and the approach we use for detecting build conflicts. In Section 3.2, we present the results, which are further discussed in Section 3.3. Threats to the validity of our study are discussed in Section 3.4.

3.1 EMPIRICAL STUDY

In this section, we present our empirical study to investigate the occurrence, causes and resolution patterns of build conflicts. Initially, we introduce our research questions and explain our study setup and the approach we use for detecting build conflicts. Next, we present and discuss our results and their implications, as also the threats to the validity of our study.

3.1.1 Research Questions

As previously motivated in Section 2.2.2, resolving integration conflicts, like build conflicts, it is a time consuming and an error-prone activity negatively impacting development productivity. While previous studies have investigated the occurrence of these integration conflicts, for build conflicts, they focus only on the frequency of these conflicts (BRUN et al., 2011; KASI; SARMA, 2013). Kasi and Sarma (2013) report that build conflict rates substantially vary across projects, ranging from 2% to 15%. Brun *et al.* (2011) find a similar range: 1% to 10%. However, this is not the main focus of the two mentioned papers; the authors observe build conflicts in a small number of projects— three in one study and four in the other. It is then important to observe conflict frequency in a larger context. In this study, we consider a substantially larger sample, with 451 projects and 20 times more merge scenarios (57065) than the aggregated sample of the two studies, to answer the following research question.

RQ1 – *Frequency:* How frequently do build conflicts occur?

To identify build conflict occurrences, we look for merge scenarios with broken merge commit builds (*errored* build status) and an error message not related to Travis environment issues (like timeout of the build process). To make sure the *errored* status was caused by the integrated changes, and it is actually a conflict, we further parse the Travis logs generated when building the commits in a scenario and automatically check whether the logged error messages are related to the changes. Suppose our scripts are not able to check that due to the limited set of patterns they recognize. In that case, they confirm conflict occurrence by observing whether the merge commit parents both present superior status (*failed* or *passed*), and the merge commit contains only the integrated changes. We conservatively do not classify the breakage as a conflict if it is caused by post-integration changes, which are often applied to fix merge conflicts but could also cause build breakage.

Although useful as initial evidence of build conflict occurrence, previous work of Kasi and Sarma (2013) and Brun *et al.* (2011) do not bring information about conflict causes. They do not investigate the *structure* of changes that cause build conflicts. We go further by investigating and classifying these causes. Seo *et al.* (2014) present technical causes for broken builds in general, but do not study build conflicts in particular. They mostly explore individual developers' changes that break builds. They do not study individual changes that do not break builds, but that interact unexpectedly and lead to broken builds when integrated. Understanding the structure of parallel changes that lead to build conflicts might help us to derive guidelines for avoiding such conflicts, and to improve awareness tools to better assess conflict risk. Hence, our second research question, which has not been explored before, is the following.

RQ2 – *Causes:* What are the structures of the changes that cause build conflicts?

Based on the results of RQ1, we further analyze build logs and the source code of merge scenarios, classifying and quantifying the kinds of changes we observe. As a result, we derive a catalogue of build conflict causes, and identify the most common causes.

Identifying conflict causes might help developers to reduce conflict numbers, but most likely will not eliminate them (KERZAZI; KHOMH; ADAMS, 2014). Fixing merge conflicts might be a demanding and tedious task (BIRD; ZIMMERMANN, 2012; ZIMMERMANN, 2007b). Fixing build conflicts might be even harder and risky, since semantic aspects must be taken into account. Build conflicts may have different causes requiring different treatment to detect and also deal with them. As a result, developers should spend more time fixing these conflicts, negatively impacting team productivity and software quality as fixing conflicts is an error-prone activity. So, to reduce build conflict resolution effort, it is important to better understand which resolution patterns are adopted, and maybe automate some of them. It is also important to understand whether conflicts are fixed by both contributors and integrators, especially because integrators might have a harder time fixing them. When we say contributor changes, we mean changes performed before integration, while integrator changes are done during the integration.[1] It is also possible that contributors play the role of integrators when integrating their contributions. Thus, our third research question, and a complementary related question, is the following:

RQ3 – *Resolution Patterns:* Which resolution patterns are adopted to fix build conflicts?

RQ3.1 - *Fixer:* Who does fix build conflicts?

To answer these questions, which have also not been explored before by previous work of Kasi and Sarma (2013) and Brun *et al.* (2011), our scripts perform an automated analysis between the broken merge scenario and the closest commit that fixes the conflict. Analyzing these fix changes allows us, at least partially, with a focus on technical terms, to derive some understanding about build conflict resolution, as we report a catalogue with the adopted resolution patterns. However, we do not investigate the motivation/discussion among developers when deciding which fix is adopted for a conflict. As we mentioned before, Zampetti *et al.* (2019) inform that when compilation errors arise in PRs, for example, they are very little discussed by reviewers, as these generic errors could be easily fixed through private builds. Finally, we chose to keep RQ2 and RQ3 separately, as it helps one to contrast our work with related work. For example, Seo *et al.* (2014) investigate

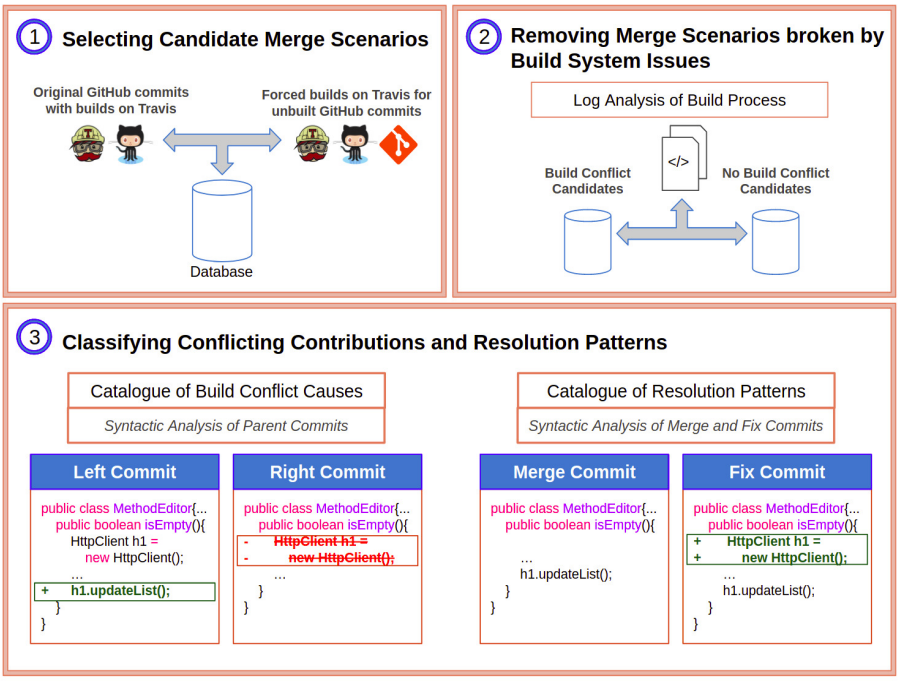
[1] This is related but not strictly the same as an alternative terminology (GOUSIOS et al., 2015) that considers contributors as commit authors, and integrators as project members that inspect and integrate changes into the project's main development line. In this terminology, contributors might also play the role of integrators when integrating their contributions.

RQ2, but not RQ3, while Sung *et al.* (2020) mainly focus on RQ3, but not RQ2. Chapter 5 discusses these differences in detail.

3.1.2 Empirical Evaluation

To answer the just discussed research questions, we structured our experiment in three main steps, as represented in Figure 8. For automating this process, we implemented scripts that perform the analyses required for the experiment. Although most of this process is automatically done using these scripts, we perform manual analysis to detect conflicts for exceptional cases not supported by our scripts due to limitations. We explain later in this section when this manual analysis is performed. In the first step, our scripts mined project repositories in GitHub and Travis to get the information required for the experiment, including the source code and build status of commits that are part of merge scenarios. As explained in Section 3.1.1, we collected only merge scenarios with broken merge commit builds (*errored* build status). In the second step, our scripts parsed the Travis build logs of the collected scenarios to filter out scenarios with build breakages caused by non integration related causes such as execution timeout of the Travis services. Finally, in the last step, our scripts confirmed conflict occurrence, and classified conflicts and the adopted resolution patterns. The scripts we used to automate these steps are available online (Appendix 1, 2022) to support replications.

Figure 8 – Build conflicts: empirical study setup. The first step (Section 3.1.2.1) mines project repositories. Step two (Section 3.1.2.2) filters merge scenarios and yield build conflict candidates. Finally, step three (Section 3.1.2.3) classifies build conflict causes and resolution patterns.



Source: The author (2022).

3.1.2.1 Selecting Candidate Merge Scenarios

To explain in more detail how we select candidate merge scenarios for our experiment, we first discuss our project sample. Then we explain the criteria we use to select scenarios from our project sample.

Project Sample

As our experiment relies on the analysis of source code and build status information, we chose for GitHub projects that use Travis CI for continuous integration for service popularity reasons. As a significant part of our automated analysis is programming language-dependent, we consider only Java projects. Similarly, as Travis build log parsing depends on the underlying build automation infrastructure, we analyze only Maven (MAVEN, 2019) projects; its log reports are very informative compared to the reports of other dependency managers. Considering more languages, build systems and CI services would demand more implementation effort.

We start with the projects used in the studies of Munaiah *et al.* (2017) and Beller *et al.* (2017b), which include a large number of selected and active open-source projects covering different languages and domains. For each project, the datasets also inform whether the project adopts any continuous integration service. We then select Java projects that satisfy the following criteria: (i) presence of the *.travis.yml* file in the root directory of the latest revision[2] of the project (this indicates that the project is configured to use the Travis service); (ii) presence of at least a build process in the Travis service, and confirmation of its active status, which indicates that the project has actually used the service; (iii) presence of the *pom.xml* file in the root directory of the latest revision of the project (this indicates the use of Maven), and absence of Gradle (Gradle, 2019) configuration files, which could represent early use of Maven, but later migration to Gradle, demanding a different build log parser; and finally, (iv) the project should have at least one merge scenario considering the history interval we were analyzing. We discuss the threats regarding our choices in Section 3.4. This way, we ensure only Java projects that use Maven and Travis were selected (HILTON et al., 2016; VASILESCU et al., 2014). As a result, we select a sample of 451 projects. Some projects of our sample are also evaluated by related work. However, all these projects adopted Travis after 2015, which is the initial year we consider to mine merge scenarios. Statistical information and associated discussion about our sample can be found online (Appendix 1, 2022).

Our sample has projects from different domains, such as APIs, platforms, and Network protocols, varying in size and number of developers. The Truth project has approximately 31.2 KLOC, while Jackson Databind has more than 113 KLOC. Moreover, the Web Magic project has 45 collaborators, while OkHttp has 195. We believe larger projects were filtered

[2] As in August 2018.

Table 1 – Summary of merge scenarios with build conflicts

Number of projects	451
Number of Merge Scenarios (MS)	57,065
Errored MS	4,252
MS with Build Conflicts	65
Number of Build Conflicts	239

Source: The author (2022).

out of our sample because of two main criteria. First, some of these projects do not adopt public/free CI practices such as Travis CI. For example, the project Liferay Portal does not indicate the adoption of an online CI service. However, it has configuration files that suggest the adoption of a private CI option. Second, other projects have adopt Travis only after we run our related studies. For example, the project Apache Commons Lang adopted Travis in 2017. The complete list of the analyzed projects can be found online (Appendix 1, 2022).

Merge Scenario Candidates

For each selected project, our scripts locally clone the project and select[3] merge commits created after the project adopted Travis CI, that is, the first build appeared in Travis. Since Travis CI is a relatively recent technology, most projects adopted it later in their history. Our filter then makes sure we select merge scenarios that are Travis ready. We also avoid selecting *fast-forward* situations (CHACON; STRAUB, 2014, p. 69), as they do not correspond to a code integration scenario. Applying this filter to our project sample, we obtained 57065 merge commits. By collecting each merge commit with its parents (the associated changes it integrates), we obtain a sample with the same number of merge scenarios (second row in Table 1).

For each merge scenario in our sample, we try to identify the Travis builds associated with the merge commit and its two parents. As not all commits originally have an associated Travis build, we force the creation of builds for these unbuilt commits. Our scripts use GitHub's API[4] to fork the corresponding project, and then reset the fork head to the unbuilt commits, triggering Travis service to start the build process. As a result, these commits are finally built. With builds for the commits in a merge scenario, we filter our sample by selecting only the scenarios with broken merge commit builds. That is, scenarios having merge commits with *errored* build status. We select or discard a merge scenario based on its final build status. A build can group different jobs that are build processes with a specific list of steps. Developers may use different jobs to build the same code under

[3] We use the `git log --merges` command with an extra parameter specifying the initial date to drive the search.

[4] <https://developer.github.com/v3/>

different environment options, like, different operating systems, platforms, and language versions. This way, each job may require access to different resources and break due to different reasons. If one of the jobs breaks due to an unavailable external resource, while the others are successful, the final build status is broken as it considers the job with the minimal status. However, when a compilation problem occurs, we observe that all jobs present the same result. So, if a build is composed of many *jobs* and not all jobs have the same final status, we do not check individual job status to reach a decision. We consider the final status reported by Travis that takes into consideration all valid build jobs. To avoid bias, we also discard duplicate merge scenarios, which involve different merge commit hashes but have the same parents. Otherwise, the same merge scenario would be evaluated twice and introduce noise in our results as we would report duplicate conflicts caused by the same changes. As a result, 4252 merge scenarios are classified (third row in Table 1).

3.1.2.2 Removing Merge Scenarios Broken by Build System Issues

To ensure the merge scenarios selected so far are associated with conflicts, and to better understand what caused the *errored* status, we further investigate the build status and associated logs of the merge commits. This investigation is needed because the broken build status might have been caused by a number of reasons not related to the merged contributions. In particular, the breakage might have been caused by build system issues such as execution timeout of the Travis services, or unresolvable dependencies no longer supported by Travis. Another example of broken builds is caused by changes performed in the environment or build script files, not on source code changed by both contributions in a merge scenario. As previously motivated, in our study, we only consider as build conflicts those cases caused by contributions performed on source code files. Eventual conflicts involving no source code files are not targeted in our study. As previously discussed (see Section 3.1.1), previous studies investigate build conflicts, but they do not explore aspects like the causes and resolution patterns adopted, as we do here. So we further analyze, for each merge commit build, its Travis log report.

Our scripts parse each log and search for the specific error messages listed in the third column of Table 2. This list was incrementally derived by observing the error messages of broken merge builds in our sample. Initially, whenever our script could not parse a log, we would extend it to consider the new kind of message that appeared in that log and run it all over again. As a result, we have a list of the most common error messages in our sample, being able to classify most (99%) broken builds we found. The remaining 1% of the cases are not classified, as their logs were empty or without information that let us classify the cause; it may happen because old Maven versions do not report logs with complete information. Our scripts do this analysis based on matchings between the expected and the observed string in the build file logs; if our scripts fail to perform these checks due to limitations, we miss these conflicts so that we may have false negatives. To

Table 2 – Build error messages related to broken builds during the build process on Travis.

Error Category	**Error**	**Build Error Message**
Static Semantic	Unimplemented Method	does not override method
	Duplicated Declaration	is already defined in
	Unavailable Symbol	cannot find class cannot find method cannot find variable
	Incompatible Method Signature	no suitable method found for cannot be aplied to
	Incompatible Types	error: incompatible types cannot be converted
Other Static Analysis	Project Rules	some files do not have the expected license header
Environment Resource	Remote problems	The job exceeded... No output received... Your test run exceeded
Dependency	Environment configuration	could not (find OR transfer) artifact

Source: The author (2022).

measure these false negatives, we would need to build merge commits, probably updating the adopted Maven version, and expect to have access to a log with complete information. However, our scripts present a high precision when selecting or removing merge scenarios during this analysis. The third column of Table 2 highlights how we group such error messages into conflict causes expressed in terms of the structure of the changes that lead to the conflicts. These causes are then grouped in the categories that appear in the second column.

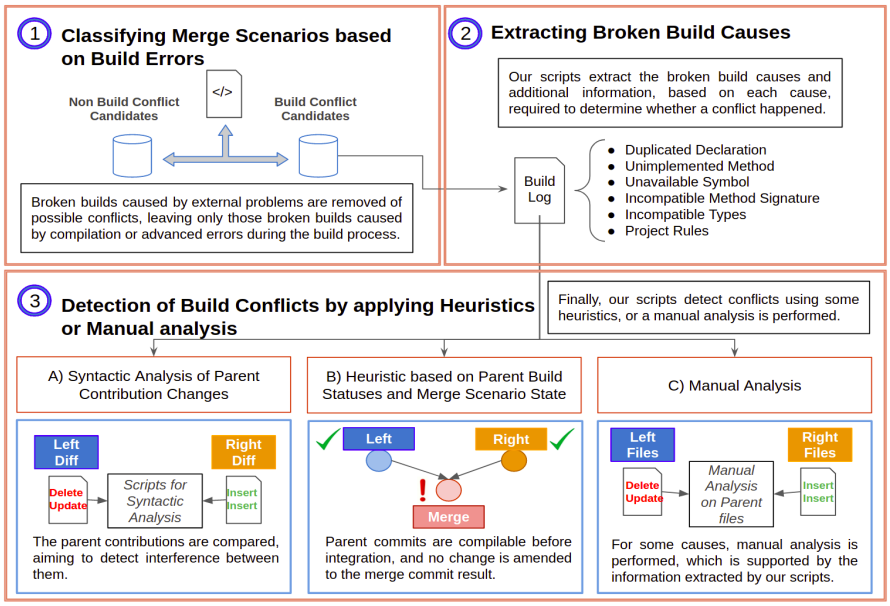
To filter out scenarios with build breakages caused by build system issues, we discard scenarios with broken merge commit builds whose logs have one of the error messages listed in rows *Environment Resource* and *Dependency* of Table 2. For these cases, we have no evidence that the corresponding build breakages are caused by the integrated changes, or are related to build conflicts.

3.1.2.3 Classifying Conflicting Contributions and Resolution Patterns

As presented in Table 2, *errored* builds might have many causes, and these are associated with specific error messages. However, even after discarding breakages caused by build system issues, these messages' occurrences do not imply conflict occurrences since the integrated changes might not have caused the error. For example, the breakage might have been inherited from one of the parent commits instead of conflicting contributions. We now discuss the extra checks needed to confirm conflict occurrence and classify conflicts

according to the mentioned causes. We structured all adopted steps for the detection of conflicts in three main steps, as represented in Figure 9. Additional information about this process can be found online (Appendix 1, 2022).

Figure 9 – All adopted steps for the detection of build conflicts. In the first step, merge scenarios are classified based on their main associated build breakage types (related or not with environmental issues). In the second step, for each merge scenario with a possible conflict, our scripts extract the associated build breakage cause. Finally, in step three, conflicts are detected based on syntactic analysis of parent contributions' changes, a new heuristic involving parent build statuses and merge scenario state, or manual analysis.



Source: The author (2022).

Checking Build Conflicts

To make sure an *errored* merge commit build in one of the selected scenarios corresponds to a build conflict, we have to check whether the integrated changes caused the build breakage (see the second step in Figure 9). So, we parse the build log and check whether the logged error messages are related to the changes integrated into the merge commit. (see the third step in Figure 9). To capture the integrated changes, we use GumTree (FALLERI et al., 2014) to compute syntactic *diffs* among the *merge commit*, its *parents*, and the associated *base*[5] commit. Conceptually, the spectrum of build conflict causes can be defined by pairs of source code changes that affect each other in the sense that, when merged, they result in an invalid program (compilation problem) or an error during the build process. As we adopt a conservative and empirical approach to detect conflicts, our results partially cover the spectrum of causes.

[5] The latest common ancestor to the parent commits.

When our scripts, due to the limited set of recognized patterns supported by them, are not able to check that error messages are related to the changes, they confirm conflict occurrence by observing whether the merge commit parents present superior status (*failed* or *passed*), and the merge commit contains only the integrated changes (see the third step in Figure 9). For that, our scripts locally replicate the merge scenario, merging the parent contributions into a new commit. They then check whether this new replicated commit has any difference when compared with the original merge commit. If no difference is observed, we assume the broken build is caused by the parent contributions revealing a build conflict as the parents present no *errored* build status. Changes performed after the merge commit creation, by `git amend` commands, for example, could be part of the amended merge commit, leading to false positives in our results. For example, suppose a developer applies fixes for textual merge conflicts during integration. In that case, we might not report the build conflict occurrence exclusively based on the superior parent build statuses, as the extra changes might be responsible for the breakage. This way, we adopt a more conservative approach to detect conflicts, ensuring that the merge scenario holds only the integrated changes, while the parent commits present superior build statuses. Our heuristic is based on the following assumptions:

- As the parent commits present superior build statuses, we may conclude that no compilation problems occur during their build processes. If any compilation problems occur after integration, that is, the creation of the merge commit, these problems were caused by the integration of parent contributions or extra changes.

- A merge scenario holding only the integrated changes, and associated with compilation problems, indicates the parent contributions cause such a problem. A compilation problem would not be motivated by unavailable external remote services, for example.

For confirming the *Unavailable Symbol* conflict, for example, we initially look for declarations removed or renamed by the changes of one developer, but referenced by the changes of the other developer.[6] So, after confirming the presence of a "cannot find..." error message (see Table 2, third column, fourth row) in the log, our scripts further parse the log to extract the following information: *the missing symbol* and the *involved classes* (one references the symbol, the other should have declared the symbol). With this information, to confirm the conflict, our scripts check whether one of the parents remove the declaration of the *missing symbol* from one of the *involved classes*, while the other parent adds a reference to the *missing symbol* in the other *involved class* (see Algorithm 1). These checks confirm the parent contributions are responsible for the error reported by the broken build. The scripts also consider the case when the removed declaration and

[6] Conflicts might even happen when integrating changes a single developer made to different branches or repositories. For simplicity, in the explanation, we consider just the most common case of conflicts caused by integrating different developers changes.

added reference are in the same class. The scripts used to perform the analysis for all build conflict causes are based on matchings expected and observed strings in the GumTree logs associated with the conflicting files. These logs are generated based on the parent commits (holding the changes performed by each parent during their contributions) and the base commit (the common ancestor). In case our scripts check the expected matching, a build conflict is detected. Although we report *Unavailable Symbol* as a cause for build conflicts, our scripts do not cover all situations in which this cause could occur. For example, consider a developer that updates an import declaration from `java.util.*` to `java.lang.List`, while another developer adds a new reference for `java.lang.ArrayList` in a different version of the same file. After integration, a build conflict is expected to be reported caused by an Unavailable Symbol, but the current implementation of our scripts do not detect them.

Algorithm 1 Checking conflict occurrence for *Unavailable Symbol*

Data: $diffChangedFilesByLeft, diffChangedFilesByRight, missingSymbol,$
$classWithTheMissingSymbol, classCallingMissingSymbol$
Result: A boolean representing whether a conflict occured

```
1 if diffChangedFilesByLeft[classWithTheMissingSymbol].
   match(/Delete (Simple|Qualified)+Name missingSymbol/) then
2     if diffChangedFilesByRight[classCallingMissingSymbol].
       match(/Insert (Simple|Qualified)+Name missingSymbol/) then
3         return True
4 return False
```

Under the presence of the "cannot find..." error message (see Table 2, third column, fourth rows), or in pathological cases of build logs that do not conform to the common expected format parsed by our scripts, we manually analyze the merge scenario to confirm conflict occurrences (see the third step in Figure 9). We check whether the integrated changes either removed and added references as just explained, or removed a build dependency that declares the missing symbol, causing a build dependency issue. In both cases, we rely on the file name information in the build log. One single author performs this manual analysis. Although the scripts could not detect the conflict in these cases, they inform the classes The author should look at to mitigate errors or mistakes during this analysis. We chose for not automating these cases because they rarely occur in our sample; for example, in our study, a removed build dependency causes a conflict only in one scenario. We actually adopt an on-demand approach for evolving the scripts, with a few cycles of first extending script functionality as needed, followed by rerunning the experiment, and then identifying and analyzing common cases not captured by the scripts. Build conflicts associated with *Incompatible Types* and *Project Rules* are manually checked. Automatically detecting conflicts caused by *Incompatible Types* using GumTree diffs might introduce false positives in our results, as the diffs treat object types as strings. Suppose a

project has two classes with the same name but on different packages, GumTree treats them as one single type. So our scripts could not differentiate them. Besides that, for cases caused by external dependencies, we could not compare different jar files using GumTree. In the same way, to automatically detect conflicts caused by *Project Rules*, we would need to work with XML and Java files as this cause involves style changes. However, we can not track style changes on GumTree diffs. We discuss the threats related to this manual analysis in Section 3.4.

In case the logged error messages are not related to the changes integrated into the merge commit, we further investigate the merge scenario. We confirm that such build failures are caused by immediate post-integration changes, often performed with the aim of fixing merge conflicts. In this case, the integrator changes, not the contributors' changes, cause the problem.[7] So we do not consider that a build conflict, but a broken build caused by changes applied to fix a merge conflict, or amend a merge commit for other reasons. Nevertheless, as these cases might also benefit from a number of applications of our results for conflicts, our scripts also analyze them. For confirming *Unavailable Symbol* breakages caused by post-integration changes, our scripts check whether both parents' changes keep the *missing symbol* declaration. If so, it implies that the integrator removed or renamed the symbol declaration. In case we cannot check this, our scripts apply a similar approach used to identify build conflicts. Our scripts now check again whether the merge commit parents present a superior status (*failed* or *passed*), and if the merge commit does not contain only the integrated changes. We check this replicating the merge scenario again and comparing the new merge commit with the original merge commit.

We follow a similar approach for confirming the other conflict causes. For brevity, they appear only online (Appendix 1, 2022).

Build Conflict Fixes

For each build conflict identified in the previous steps, we analyze the adopted conflict resolution pattern and who (integrator or contributor) was responsible for applying the fix. So we first look for fix commits performed in the main branch. For a merge commit with a build conflict, and consequently, a broken build, the fix commit is the first commit that follows the merge commit and has a superior build status. For build conflicts, we consider *failed* and *passed* as superior status for a fix; both statuses indicate that the source code could at least be compiled. Our approach considers the commit fix statuses as the starting point for our analysis. Suppose a possible fix commit under analysis does not present a superior build status. In that case, we move for the next commit performed after the previous commit under analysis until we find a proper commit with a superior build status. We adopt this strategy because a developer might have created a few conflicts before she

[7] The same developer might be playing both roles in the same merge scenario, but that is not necessarily the case.

is actually successful in fixing the conflict. If the fix is associated with a commit that introduces new conflicts or errors motivated by the commit changes, the build process fails as expected, leading our scripts to move to the next commit, instead of already extract the resolution pattern.

Our scripts automatically analyze the fix commits, extracting common resolution patterns that appear in the third column of Table 4. Similar to the scripts that identify conflict causes (see Section 3.1.2.3), we adopted an on-demand and relevance based approach for evolving the scripts that identify conflict resolution patterns. The scripts used to detect the resolution patterns also follow the same approach adopted for the detection of build conflict causes. The scripts work based on matching expected and observed strings in the GumTree logs associated with the broken merge and fix commits' files. In case our scripts perform this matching of strings, a resolution pattern is detected. Uncommon and harder to automate cases of build conflict resolution patterns are manually handled. Based on the conflict type and files involved, we analyze the fix commits looking for changes in those files. Similar to the automated analysis, we use a syntactic diff between the merge and fix commit, getting all syntactic differences. The list of resolution patterns in Table 4 covers all fixes we are able to identify in our sample.

3.2 RESULTS

Following the empirical study design presented in the previous section, we analyze merge scenarios in 451 GitHub Java projects to investigate the frequency, causes, structure, and resolution patterns adopted for build conflicts. This section details our results, answering our research questions. As explained in previous sections, the first question has been explored before but in a significantly more restricted scope. The other questions are originally explored here.

3.2.1 RQ1: How frequently do build conflicts occur?

To answer RQ1, we follow the steps in Section 3.1.2, select merge scenarios in our sample, discard non-conflicting scenarios, and count conflict numbers in the remaining scenarios.[8] We then find 239 build conflicts in 65 scenarios (see Table 1, fifth row). These conflicts were automatically (174 cases) and manually (65 cases) identified.

To better contextualize that, we observe in Table 1 that roughly 7.5% of the merge commit builds are broken in our sample (4252 out of 57065 merge scenarios). This maybe surprisingly high rate of broken merge builds is due to a number of causes: build environment (Travis timeouts, unavailability of external services such as package manager servers, bugs in build scripts, and configuration problems) issues; integration conflicts;

[8] Each scenario might have a number of conflicts caused by different changes and associated with different error messages in the build log of the scenario merge commit.

defects in post-integration changes; or defects, and consequently breakages, inherited from the parents. Most breakages are due to the first cause.

Part of the build conflicts, 51%, have parents with superior build status; the build breakage just arises after the merge scenario integration; no error is inherited from the parent commits. The remaining cases, besides build conflicts, may also present additional errors not caused by conflicting contributions but inherited from their parent commits. Other non inherited breakages occur when a merge commit and at least one of its parents have builds with environment issues, but these are less interesting for our discussion.

We believe the low numbers and frequencies of build conflicts in our sample are highly influenced by the use of continuous integration practices and services in the projects we analyze. With automated build and testing processes in these projects, developers can easily build their contributions before sending them to the main repository (FOWLER, Sep 2009). All projects should have guidelines informing required steps and how the build process should be done. However, some of these steps must be challenging or expensive to perform on an individual developer's workspace, leading developers to use the CI service to build/test their contributions (ZAMPETTI et al., 2019). For that reason, we believe an automated process is a distinct factor. Developers are often, by project guidelines, required to locally integrate their contributions into the main repository contributions before submission for approval or final integration. It has an impact on build conflicts but also on other conflicts like test ones (SILVA et al., 2020). For this particular type, the developer may ensure his contribution does not change previously validated behavior. This way, we assume most build conflicts are actually detected and resolved locally, in contributors' private repositories. Our results indicate that build conflicts occur during software development supporting the findings of previous studies. However, many conflicts do not reach the remote repositories as developers fix them before sending their contributions. That is an assumption motivated by our personal experience and based on observations in previous work on the literature (ACCIOLY; BORBA; CAVALCANTI, 2018; HILTON et al., 2016; ZHAO et al., 2017). For example, Accioly *et al.* (2018) observe that when applying improved merge tools on merge scenarios, merge conflicts are reported involving duplicated method declarations. So traditional tools would merge the code without reporting a merge conflict, but only during the build process, the build conflict would arise. These findings bring evidence that developers locally face build conflicts, but solve them before sending their contributions to the remote repository.

The adoption of CI does not avoid or prevent build conflicts, as they would still occur when developers integrate code on their private local repositories. On the other hand, some companies may not apply CI on their projects to all integration scenarios due to high implementation or execution costs (long build times, large number of builds, etc.) and culture change. Hence, these projects might directly benefit from our findings, as they provide insights to deal with them based on their common causes and resolution

patterns. In the same way, the adoption of CI should not be taken as responsible for avoiding or overcoming conflicts. As previously discussed, related work on literature has shown evidence of build conflicts occuring on private local repositories of developers. Hence, the context of software development with CI narrows the conflicts that reach public repositories. In the same way, Sung *et al.* (2020) investigate the occurrence of conflicts but in a third different context, when different forks of the same project are integrated with each other.

As our study analyses only public repositories, we do not have access to problematic code integration scenarios that were locally amended before reaching the main repository. Consequently, our numbers reflect the number of build conflicts that reached public repositories, not the actual number of conflicts that happened and had to be resolved. This justifies the high frequency of merge scenarios with successful build processes, and also of non integration related breakages. As these two aspects widely vary across the analyzed projects, we miss existing integration conflicts. Sung *et al.* (2020) also investigate build conflicts, but when changes of different projects are integrated; developers face build conflicts when pulling changes from a project into another. Analyzing a project during three months, they report the occurrence of 398 build conflicts. Their results bring evidence that build conflicts occur when integrating code across different projects, when these conflicts do not reach the main development line.

Regarding the interval of commits analyzed in our study, knowing that our selection sample criteria rely on the adoption of Travis CI, we did not analyze the whole projects' history. So we might have missed conflicts that occur before Travis' adoption. However, we believe the causes reported here might still cover possible previous commits' cases. On average, our sample projects have been adopting Travis for more than three years, although there are projects that have used it for more than six years. Additional details about the Travis adoption by each project are available online (Appendix 1, 2022).

In the end, 37 projects of our sample present build conflicts. Regarding the distribution of conflicts by project, we observe that five projects hold half of all reported conflicts. For example, Pinpoint and Ontop's projects present 33 and 25 conflicts, respectively. Although other projects do not report conflicts with the same frequency, their occurrence shows that conflicts are expected to occur during a project life cycle. As presented in Section 3.1.2, our analysis considers only the main development line. So we are not able to identify conflicts that occur on local developers' workspaces. Despite the low frequency, when compared to related work (KASI; SARMA, 2013; BRUN et al., 2011), this frequency is 9 and 12 times, respectively, bigger. Furthermore, these projects cover different domains showing that build conflicts are not restricted to a specific domain (see Section 3.1.2.1). Despite the low number of projects with conflicts, we do not observe exceptional characteristics that could justify the conflict occurrence in these projects.

As we comment in Chapter 2, our results show that different conflict causes may coexist

Table 3 – Catalogue of build conflicts

Build Conflict Category	**Cause**	#	(%)
Static Semantic	Unimplemented Method (method from super type or interface)	12	5.02
	Duplicated Declaration (elements with the same identifier)	5	2.09
	Unavailable Symbol (reference for a missing symbol)	157	65.70
	Incompatible Method Signature (unmatched method reference)	26	10.88
	Incompatible Types (type mismatch between expected and received parameters)	17	7.11
Other Static Analysis	Project Rules (unfollowed project guidelines)	22	9.20
Total		**239**	**100**

Source: The author (2022).

in a merge scenario. While previous studies consider the build process breakage as one single conflict occurrence, we go further and detail the different causes as the conflict causes occur independently (see Section 3.1.2). Analyzing the distribution of conflicts based on merge scenarios, we identify 24 out of 65 merge scenarios present more than one single conflict. Besides the effort and time spent to fix these conflicts, different conflict causes also require different approaches to handle them.

We report a catalogue of 239 build conflicts in 65 merge scenarios. Additionaly, we observe that more than one build conflict may occur in a merge scenario as the causes for these conflicts are independent. We believe this frequency is motivated by the characteristics of our sample, like the use of CI for all merge scenario commits, allowing developers to detect conflict in their private workspaces.

3.2.2 RQ2: What are the structures of the changes that cause build conflicts?

Based on the conflict occurrence results, we proceed with further analysis to answer RQ2 by investigating conflict causes. As a result, we observe six build conflict causes used to define our catalogue of build conflicts. Table 3 shows these six causes, their descriptions and frequencies are shown grouped by cause categories (second and third columns).

Most build conflicts are caused by Unavailable Symbol

We find that 65% of all build conflicts are caused by a reference for a missing declaration (third column, fourth row in Table 3). The most recurrent missing symbols are classes (112 occurrences), corresponding to 73% of all *Unavailable Symbol* occurrences. Missing methods (22) and variables or parameters (20) come next, with the missing declaration and the dangling reference possibly associated with the same class. The other six causes occur less frequently (each in less than 35% of the cases).

Concerning the distribution of *Unavailable Symbol* cause, we observe 39 out of 65 merge scenarios present this conflict cause. Looking at its distribution on our project sample, we observe these 39 scenarios belong to 27 out of 37 projects with conflicts. These numbers show how recurrent conflicts of this type occur and reinforce the need for an approach to deal with them. Especially because the effort to fix each conflict directly depends on the type of the missing element, and each type requires specific attention; we explain it in detail in Section 3.3.3. Furthermore, even for conflicts that require small and straightforward changes, fixing them must be a tedious task, negatively impacting team productivity. For conflicts that occur in dynamic languages, we might consider a more negative impact as the conflicts would arise as test conflicts, or even worse at runtime. This way, end-users would experience the system on production falling (unavailability), while the team would need to rush to fix the problem and put the system back on service.

Unplanned dependencies also cause build conflicts

We have observed a number of *Unimplemented Method* conflicts, which often occur when one developer adds a method to an interface while another developer adds, to an existing class, an implements clause referencing the same interface. The build then breaks because the existing class does not declare the method introduced to the interface; the class developer is not aware of that method, even though there is a direct dependence between the class and the interface. So the changes introduce an *unplanned direct dependency* causing the conflict. Nevertheless, we have also observed similar problems when developers change files or program elements that are not directly related.

For example, in the Ontop project (see Figure 10),[9] one developer adds the new interface `Var2VarSubstitution` (line 2), which extends another interface (`Substitution`, line 12), and its implementing class `Var2VarSubstitutionImpl` (line 7). The other parent, unaware of the previous changes, adds the new method `composeFunctions` to the `Substitution` interface (line 14). Once all contributions are integrated, the build process breaks as the `Var2VarSubstitutionImpl` class does not implement the method `composeFunctions` added to the interface.

Another example occurs in the project PAC4J (see Figure 11).[10] While one devel-

[9] Build ID: ontop/ontop/59371438 – Merge Commit: c626206
[10] Build ID: pac4j/pac4j/291027337 – Merge Commit: e18dd85

Figure 10 – Unimplemented Method conflict caused by new method added on old interface.

```
//New interface added by Left
public interface Var2VarSubstitution extends Substitution {
    {...}
}

//New class added by Left
public class Var2VarSubstitutionImpl implements Var2VarSubstitution {
    {...}
}

//New method declaration added by Right
public interface Substitution {
    {...}
+   boolean composeFunctions(Function term1, Function term2);
}
```

Source: The author (2022).

oper adds the class `DigestAuthExtractor` implementing interface `Extractor` (line 2), the other developer changes the location of the interface class (lines 7 and 8). Besides the expected build conflict occurrence, caused by the dangling reference to interface `Extractor` (*Unavailable Symbol*), the `override` anontations in `DigestAuthExtractor` class also causes build conflicts. In this context, there is no super class or interface associated with `DigestAuthExtractor`. So it is not possible to override a method. Different from the previous example, when the missing method implementation causes the conflict, here the missing method declaration in the interface or super class is the reason for the conflict occurrence.

Figure 11 – Unimplemented Method conflict caused by interface class location update.

```
//New class added by Left
public class DigestAuthExtractor extends Extractor {
    {...}
}

// Interface class location updated by Right
// old location: org.pac4j.http.credentials.extractor
// new location: org.pac4j.core.credentials.extractor
public interface Extractor {
    {...}
}
```

Source: The author (2022).

In the same way, a related case happens in the Ontop project (see Figure 12).[11] The build conflict occurs due to the class `MonetDBSQLDialectAdapter` (line 9), which presents a method implementation for `strconcat` (line 4) that was not declared in the interface `SQLDialectAdapter` (line 2). So the annotation `override` above the method declaration could not be resolved. This case occurs because one parent updates the method name from `strconcat` to `strConcat` (line 5), while Right adds the class `MonetDBSQLDialectAdapter`.

[11] Build ID: ontop/ontop/83689234 – Merge Commit: 0f62121

Figure 12 – Unimplemented Method conflict caused by method name update.

```
   //Method name updated by Left
 2 public interface SQLDialectAdapter {
       {...}
 4 -   public String strconcat(String[] strings);
   +   public String strConcat(String[] strings);
 6 }

 8 // New class added by Right
   public class MonetDBSQLDialectAdapter {
10     {...}
   }
```

Source: The author (2022).

Build conflicts are caused by copy/paste actions involving different branches

The reported build conflicts in this category are caused by the addition of methods with the same signature in the same class, or the same variable added in the same method (third column, third row in Table 3). For duplicated methods, each parent commit adds its method declaration in a class resulting in one single declaration. However, after the merge scenario, the class presents two methods with the same signature. Analyzing these reported cases, we observe, in all occurrences, the duplicated methods have the same implementation (method body); for example, in the Blueprints project, the same method is added in the class `GraphTestSuite`, `testRemoveNonExistentVertexCausesException`.[12] It may happen when, for example, a developer is working in two different branches and decides to use changes previously done in one branch into the other. Instead of integrating his work with the commit holding the desired method, the developer decided to copy/paste it. Accioly *et al.* (2018) also observe this behavior in their study. They bring evidence these operations are done in practice and the local occurrence of build conflicts experienced by developers.

Build conflicts also involve unmatching operations

During her contribution, when a developer adds a new reference for a specific method, it is expected this reference may be resolvable by matching the call reference for the method declaration in the class supposed to hold it. However, when these matchings are not resolvable, build conflicts may happen caused by *Incompatible Method Signature* (third column, fifth row in Table 3). For example, in the Spark project (see Figure 13), one parent adds a new method call in the class `MatcherFilter` to the method `modify` of class `GeneralError` (line 6).[13] The other parent, unaware of the changes previously done, changes the signature of method `modify` adding a new parameter (line 13). It also updates previous method calls using the old signature to the new signature. However, when the

[12] Build ID: tinkerpop/blueprints/267833702 – Merge Commit: 5a25e3a
[13] Build ID: perwendel/spark/229805514 – Merge Commit: 3fd18a9

parent commits are integrated, the first parent's new method call is not resolvable as the method signature has changed. Similar to *Unavailable Symbol* conflicts, this cause involves a method reference that could not be resolved, but the method is not removed or renamed. This cause requires more attention by the developer as there is a slightly different signature that may confuse the developer during analysis. In case this initial analysis is done locally, the developer may be supported by IDEs, which correctly indicates the mentioned problem using type checking. If this analysis is remotely done, the developer may spend more time understanding the problem.

Figure 13 – Incompatible Method Signature Conflict caused by method signature update.

```
//New method call added by Left
public class MatcherFilter {
    {...}
    public void doFilter(){
       {...}
       GeneralError.modify(httpResponse, body, requestWrapper,
          responseWrapper, generalException);
    }
}

//New parameter added on method modify by Right
public class GeneralError {
    {...}
    static void modify (HttpServletRequest httpRequest, {...});
}
```

Source: The author (2022).

In the same way, consider the occurrence of build conflicts caused by *Incompatible Types* as unbound matching between an expected and an observed type (third column, sixth row in Table 3). For example, in project Elasticsearch-SQL, one parent adds a new local variable `genderKey`, which is initialized by the method call `getKey` of an external class `Bucket` returning a `String`.[14] The other parent updates the version of the jar used in the project, that holds the class `Bucket`. Now the method `getKey` returns no longer a `String` but an `Object`. When the contributions are integrated, the variable `genderKey` can not be initialized with an `Object` type as its type is `String`. This case requires more attention as it is expected the developer to initially explore the classes involved and reported in the broken build log. When no valuable information is observed inspecting these reported classes, the developer might explore the history changes performed by the parent commits and then verify the change of an old dependency for its new version. *Incompatible Types* and *Incompatible Method Signature* causes are related to type mismatch, so there is no conceptual difference among them. However, we decide to separate them as individual causes based on the classification of prior related work (SEO et al., 2014), so that we could more easily compare results.

[14] Build ID: NLPchina/elasticsearch-sql/99402562 – Merge Commit: eb5fe5f

Figure 14 – Project Rules conflict caused by unfollowed project guideline.

```
1  //New classes added by Left
   /* Copyright 2005-2015 hdiv.org */
3  public class HTTPSessionCache {
       {...}
5  }

7  /* Copyright 2005-2015 hdiv.org */
   public class SimpleCacheKey {
9      {...}
   }
11
   //License header updated by Right
13 - Copyright 2005-2015 hdiv.org
   + Copyright 2005-2016 hdiv.org
```

Source: The author (2022).

Build conflicts are also caused by non compilation problems

Although most build conflicts are related to programming language static semantic problems (third column, 2-6th rows in Table 3), we find conflicts due to failures during the execution of static analysis tools caused by *Project Rules* (third column, seventh row in Table 3). These failures appear during the so called ASAT phase of the Travis build process.[15] The integrated source code is compilable, but the static analysis presents errors, like name and style conventions adopted by the project, breaking the build process.[16] For instance, in the project HDIV (see Figure 14),[17] one developer updates the license header file (line 14), while the other developer adds two new classes (`HTTPSessionCache` and `SimpleCacheKey`, lines 3 and 8). As the newly added classes have the old header style, the verifications identify inconsistencies caused by build conflicts. We decide to include *Project Rules'* causes in our catalogue as the conflicts break the build process and involve conflicting contributions. However, as reported in Table 3, we classify them as *Other Static Analysis* as this cause does not involve static semantic aspects like the other causes. Unlike the other conflict types caused by compilation problems, *Project Rules* arise when advanced static analyses are performed on the source code during the build process. As a result, a set of violations related to this conflict type occurs, which requires developers to fix them to continue the ongoing integration. Someone may argue that fixing these violations is less error-prone when compared to the fixes applied for other conflict types; however, it is important to have in mind that developers will waste time on these fixing tasks in both situations.

The build conflicts we report here are related to compilation problems that arise during the source code compilation (product). So compilation problems that might occur during the test code compilation are not covered by our results. However, in our previous

[15] Travis Documentation: The Build Lifecycle
[16] Broken builds caused by errors during automated static analysis are classified as *errored* by Travis.
[17] Build ID: hdiv/hdiv/126140680 – Merge Commit: c620070

Table 4 – Catalogue of resolution patterns adopted to fix build conflicts.

Conflict Category	**Cause**	**Resolution Patterns**	#	(%)
Static Analysis	Unimplemented Method	Super type class addition	5	3.38
		Method implementation	5	3.38
	Duplicated Declaration	Duplicated element removal	2	1.35
	Unavailable Symbol Class	Missing class import update	47	31.75
		Missing class reference removal	27	18.24
		Missing class reference update	4	2.70
	Unavailable Symbol Method	Missing method reference update	7	4.73
		Missing method reference removal	6	4.06
		Associated class import update	3	2.03
	Unavailable Symbol Variable	Associated class import update	12	8.11
		Missing variable reference update	4	2.70
		Missing variable reference removal	2	1.35
	Incompatible Method Signature	Required method reference update	3	2.03
		Required method reference removal	2	1.35
		JDK setup	7	4.73
	Incompatible Types	Type update	9	6.08
Other Static Analysis	Project Rules	License header update/removal	3	2.03
Total			**148**	**100**

Source: The author (2022).

studies (SILVA, 2018), some build conflicts occur during such circumstances; although they do not occur with the same frequency as in the compilation phase, their causes are covered by our catalogue of conflict causes.

Our catalogue of build conflicts groups six causes. The most recurrent cause is *Unavailable Symbol*, which can be split into sub-categories, like *Unavailable Symbol Class, Method* and *Variable*. These build conflicts are caused not only by static semantic problems but also static analysis performed after the compilation phase during the build process.

3.2.3 RQ3: Which resolution patterns are adopted to fix build conflicts?

To answer this question, we first tried to identify commit fixes for all observed conflicts. For a number of conflicts, we were not successful for the following reasons. First, the conflict was not fixed because it was potentially hard to resolve; as it occurred in an

auxiliary branch, developers ignored the changes in the branch and moved on to the main branch. Second, the fix appears only after our limiting date for analyzing project history when running our study, either because the conflict occurred not long before this date, or because the project receives only sporadic contributions (or is no longer active).[18] Third, as our scripts try to build fix commits not built yet, environment problems may occur during this attempt, breaking the build process. It was a common problem when building commits on Travis, when we force the creation of unbuilt commits in our forks. In general, these problems occur when a project adopts external dependencies under development, not a final or stable release. This way, when these dependencies become stable, dependency repositories are updated with them, and all previous related dependencies are removed. So in these cases, we are not able to assess the fix commit because the build process would continue as *errored* status. Finally, as explained before (see Section 3.1.2.1), our scripts discard duplicate merge scenarios; the fix could be associated with the scenario we discarded. So we analyze fixes for just part of the conflicts: 148 fixes out of all build conflicts. Only part of these conflict fixes is automatically analyzed by our scripts (44), while the remaining cases (104) are manually analyzed. We adopt this manual analysis as some patterns could not be checked using GumTree diffs, or rarely occurred; for example, in *Unavailable Symbol* conflicts caused by a missing class, they can be fixed by importing the whole package where the missing class is declared. However, there is no way to ensure the missing class is declared in that imported package by using GumTree diffs. We decide to not include metrics of spent time and number of broken builds until the fix be done. We believe these metrics could be negatively impacted by delays due to the unavailability of developers when doing a fix. In our sample, there are projects managed or funded by companies, like Microsoft[19], but other projects are not led or supported by private companies. So contributors are expected to contribute when they have available time, so there is no pressure to rush into dealing with eventual problems. Hence, including these metrics in our context as a proxy for effort would bias our findings and conclusions.

The identified resolution patterns, together with their frequencies, appear in the fourth column of Table 4.

Developers often fix the integrated code preserving parent contributions

We observe that build conflicts are fixed using 17 resolution patterns. Most of these fixes are done aiming to integrate the code involved in the conflict, instead of completely discarding changes and conservatively restoring the project state to a previous commit. This practice is adopted in 73% of all analyzed conflict fixes. Build conflicts caused by *Unavailable Symbols* are fixed by removing or updating the dangling reference; not only

[18] As we select our sample using previous studies' datasets, we may have selected projects that were active during the execution of these previous studies but no longer during our study. This way, we look for commits dated until August 2018, when we ran our study.

[19] https://github.com/OfficeDev/

the direct reference for the missing *unavailable symbol* itself but also the reference for the class, where the *missing element* should be available. However, roughly 62% of the fixes are done by updating the dangling reference. The most recurrent resolution pattern adopted is related to the *Update of the class import* holding the element that caused the conflict. The build conflict may be caused by a missing reference for a field, a method, or even a direct reference for the class itself. For example, in project Jackson-Core, a build conflict happens due to the location change of class `JsonFactory`.[20] While one parent adds a new reference for this class and its import in class `AsyncTokenFilterTest`, the other parent changes the location of that class. After the integration, the import could not be resolved, and the build conflict is reported. Updating the import declaration of the missing class is enough to fix the conflict. Although this resolution pattern is straightforward, redoing the same update operations or removing old imports that are no longer valid repeatedly is a tedious activity that could be automatically done.

In other cases, when a change in classes' location does not cause a conflict, the adopted resolution pattern follows the same change structure. For example, *Unavailable Symbol* conflicts are often resolved by updating the dangling reference. In the Java Driver project, the reference for the missing symbol `builderWithHighestTrackableLatencyMillis` in the request of the `Activator` class was updated to the new method signature `builder`.[21] For the cases that discard the changes instead of adjusting them, we may comment one scenario of the OkHttp project. So the reference to the *missing symbol* `deadline` (local variable), in the `GzipSource` class, was just removed.[22]

The other conflicts are also fixed in the same way aiming to preserve all parent contributions, except for build conflicts caused by *Duplicated Declaration*, whose cases are fixed by removing the duplicated method. In the Blueprints project, as previously discussed, in the `GraphTestSuite` class, both contributors added the same method `testRemoveNonExistent-VertexCausesException`.[23] The integrator fixes the conflict by removing the *duplicated* test case. In these cases, the duplicated methods share the same body, so no behavior change on the program would be observed after removing one of the duplicated methods. Regarding this conflict involving duplicated test cases, someone may argue that the developers want to test the same piece of code twice. If so, the integrator could rename one of the duplicated test cases and keep both tests. However, this observation is not applied to duplicated methods, when both developers add the same method and a specific call for that method. Renaming one of these methods and keeping both would introduce duplicated code in the class, which is not a good practice.

Conflicts caused by *Incompatible Types* and *Incompatible Method Signature* are fixed by adopting straightforward actions. For example, in project Elasticsearch-SQL, as previously

[20] Build ID: FasterXML/jackson-core/382865581 – Merge Commit: fdf1663
[21] Build ID datastax/java-driver/144600040, Fix Build ID datastax/java-driver/144856728.
[22] Build ID square/okhttp/19399475, Build Fix ID square/okhttp/19404300.
[23] Build ID tinkerpop/blueprints/267833702, Build Fix ID tinkerpop/blueprints/10120795

discussed (see Section 3.2.2), the build conflict is fixed by adjusting the return of method `getKey` for `String`. It is done by adding a call to the method `toString`, which now returns a `String` as expected by the parent contribution changes.[24] For the build conflict observed in project Spark, as previously discussed, the *Incompatible Method Signature* is fixed by adding the new required parameter in the call for the method `modify`.[25] In both cases, the project already has source code showing how the developer could fix the conflict. This source code is available in both scenarios as the parents responsible for adding the unexpected changes also updated the old source code impacted by his changes. They perform these changes before integrating their contributions in the remote repository.

In other cases, adopted resolution patterns are not made in source code but on configuration files. For example, in project Vavr, the conflict is fixed by forcing the installation of a specific JDK version when building the project (changes on `travis.yml` file).[26] As we previously discuss (see Section 3.2.2), in this work, we do not focus on conflicts caused by configuration files. However, this is a particular case caused by changes in source code that requires specific environment configurations to be settled before building the project.

Most fixes are done by parent commit authors

Once the resolution patterns are identified, we investigate who was responsible for the fixes. Most fix commits are authored by one of the contributors involved in the merge scenario (137 out of 148 fixes follow that pattern). In this context, contributors may play an integrator role by integrating their contributions and applying changes when required. We could also observe this situation in other development models; for example, in pull-based development, when integrators require contributors to perform changes. So contributors would fix the build conflicts until the pull request is stable and can be accepted. Otherwise, considering the integrator is not aware of these conflicts, he can accept the PR polluting the repository. The integrator could also apply these changes to fix conflicts or any problem affecting a PR, but in general, these changes are performed by the contributors who submitted the PR.

Our catalogue of resolution patterns is composed of 17 fixes adopted by developers to fix build conflicts. These patterns show that fixes do not only preserve all contributions from the merge scenario but also discard some of them. In most cases, these fixes are applied by one of the developers involved in the merge scenario.

[24] Build ID NLPchina/elasticsearch-sql/99402562, Build Fix ID NLPchina/elasticsearch-sql/99402657
[25] Build ID perwendel/spark/229805514, Build Fix ID perwendel/spark/403188038
[26] Build ID vavr-io/vavr/58945430, Build Fix ID vavr-io/vavr/58958884.

Availability of data and material

Our catalogues of build conflicts and scripts we used to perform this study are available online (Appendix 1, 2022). These catalogues can be consulted and exported for further analysis. We encourage study replications once our scripts are available for the community. However, exact replications directly depend on data stored in external services (GitHub and Travis CI). Additional instructions and further information can also be found online (Appendix 1, 2022).

3.3 DISCUSSION

In this section, we discuss our findings and their implications, and how they can be applied to assistive software development tools.

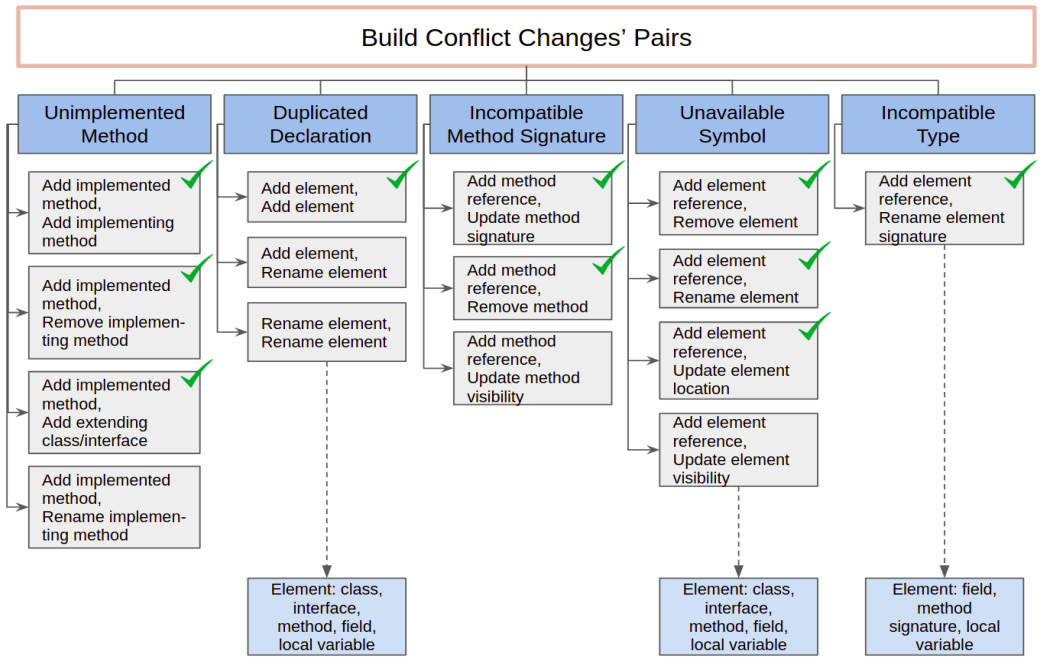
3.3.1 Findings

Comparing our RQ1 results with previous studies, we conclude that, in our sample, build conflicts occur less frequently. Kasi and Sarma (2013) show build conflict occurrence ranges from 2% to 15% of the analyzed merge scenarios across projects. Brun *et al.* (2011) find a similar range: 1% to 10%. We report significantly inferior numbers. However, our number should not be interpreted as conflict occurrence rates in general as studied in previous work, but as conflict occurrence that reaches public repositories and, therefore, are more problematic. So, given the differences in sample size and characteristics, our conflict frequencies results actually complement previous results. This number discrepancy is further justified by limitations and bias in the mentioned studies, as later explored in Chapter 5. For example, to confirm conflict occurrence, previous studies consider only the build process status of merge commits, while we additionally confirm that the changes are related to the build error message, making sure build breakages actually correspond to conflicts. So we eliminate possible threats by adopting a more conservative and well-designed approach. We must say all these conflict causes are intrinsic changes that were introduced by specific changes in the source code (RODRÍGUEZ-PÉREZ et al., 2020). Even conflicts caused by external dependencies, they occur because a developer changes the external dependency version.

In our study, we investigate build conflicts aspects not targeted by the related work of Kasi and Sarma (2013) and Brun *et al.* (2011). While previous studies focus on the frequency of build conflicts, we go further analyzing and classifying the *structure* of changes that cause build conflicts and their *resolution patterns*. In the next section, we discuss in detail the implication of these new contributions. Regarding the spectrum of build conflict causes, specifically within the context of the Java programming language, there is a limited list of changes' pairs that might result in build conflicts. Based on our knowledge and related work, Figure 15 presents a list of possible build conflict changes' pairs. Our results

cover 10 out of 15 possible changes' pairs; the five uncovered pairs of changes could appear in a larger dataset. For example, it is reasonable to expect *Unavailable Symbol* conflicts caused by changes applied to element visibility. Besides these, we omitted changes' pairs for conflicts caused by *Project Rules*. As this conflict category is caused by potentially arbitrary project style guidelines, we understand there are multiple combinations of changes that might result in conflicts. Essentially we would have to consider any kind of addition, renaming, and removing actions.

Figure 15 – Build conflict theory for Java programming language. Changes' pairs for the Java programming language that might result in build conflict when performed in parallel. We report 15 changes' pairs, which 10 of them are covered by our results.



Source: The author (2022).

In our study, *Unavailable Symbol* is the most frequent build conflict cause. Although Seo *et al.* (2014) discuss build errors in general, not relating them to integration conflicts, they report a related finding: *Unavailable Symbol* is the most frequent cause of build error in their sample. It reveals the common mistake of removing or renaming declarations, but possibly not updating all references to the new identifiers. Assistive tools like Palantír (SARMA; REDMILES; HOEK, 2012) could be applied to anticipate the emerging conflicts, or even treat them directly. In this way, developers could be aware during the tasks' development that a new reference for a symbol will fail during integration (in case of build conflicts).

Build conflicts caused by *Duplicated Declaration* have also been observed as semistructured merge conflicts in previous studies of Cavalcanti *et al.* (2017) and Accioly *et al.* (2018). It supports our assumption that build conflicts frequently occur in private repositories,

but not so much in public repositories of projects with automated build processes and continuous integration.

Although our catalogue of resolution patterns owns straightforward fixes for build conflicts, which do not involve too complex changes to fix them, we believe recurrently fixing conflicts like this would negatively impact the productivity of developers. This way, the just mentioned assistive tools might assume the responsibility of applying these changes, in a way developers might not even realize when these fixes are performed.

We believe some build conflicts might be avoided if their associated conflicting contributions are not performed in parallel; for example, a better assignment of activities based on the chance of interference among contributions. Rocha *et al.* (2019) propose a tool for predicting the files that could be changed during a task. If two tasks are predicted to change the same set of files, these tasks should be assigned for developers at a different time, not in parallel. As a result, potential merge conflicts (textual) would be avoided as developers would change different files. However, there are no guarantee build conflicts would not arise; for example, our results show that build conflicts occur in the same and dependent files. So we still believe in some situations, conflicting contributions will be required to be executed simultaneously, and consequently, build conflicts will arise, impacting the team productivity and the software quality. Van der Veen *et al.* (2015) propose prioritizing pull request (PR) integration aiming to reduce the chances of conflicting integrations. However, prioritizing PR integration at this point would not avoid the occurrence of build conflicts as the contributions are already implemented. So the conflicts could be earlier reported but not avoided.

Although most source code related to build breakages we observe in merge scenarios are caused by build conflicts (239), we also find evidence of build breakages caused by immediate post-integration changes (485), often performed with the aim of fixing merge conflicts. Such post-integration changes occur when integrators change the merge integration result before committing it. For example, in project Traccar, both parents change overlapping lines of the class `WebServer`.[27] While left parent adds the new method `initRestApi`, right also adds the method `initConsole`. During the integration attempt, the merge tool reports a merge conflict treated by the integrator. As a result, the method declaration `initRestApi` is removed, and consequently, a reference for this method could not be satisfied, breaking the build process.

This kind of breakage is consistent with our experience that fixing merge conflicts is occasionally challenging and error-prone, possibly introducing other kinds of conflicts that are harder to detect and resolve. Teams adopting improved tools like semi-structured merge (CAVALCANTI; BORBA; ACCIOLY, 2017) would reduce the number of spurious merge conflicts, and possibly reduce the risks of build breakages caused by immediate post-integration changes.

[27] Build ID traccar/traccar/232042524, Merge Commit ca06e8d.

In other cases, even the parent contributions not conflicting with each other, the integrator's changes during the integration are responsible for the broken build. For example, in project DSpace, the parent contributions do not conflict in Java files.[28] Even though, the broken build is caused by an *Unavailable Symbol* (`DSpaceSetSpecFilter` class). The class `DSpaceItemRepository` presents a dangling reference that may not be resolved for class `DSpaceSetSpecFilter`. The right parent is the only one changing class `DSpaceItemRepository`. The class file location is changed, and a new code is added. Despite the changes in class `DSpaceItemRepository` introduced by the right parent, the integrator is responsible for removing class `DSpaceSetSpecFilter` breaking the build.

As our build classification criteria are programming language-dependent, studying, for instance, Ruby projects could lead to rather different build conflicts frequencies. For example, a missing reference for a method leads to a build conflict in Java since the code cannot be compiled and built. However, in Ruby, this would likely be classified as a test conflict,[29] and only if there is a test case that exercises such method reference. Due to Ruby dynamic features, there is no pre-execution check about the existence of method declarations. Regarding the causes of conflicts, we believe our current catalogue of causes covers most cases of build conflict occurrences. For example, Sung *et al.* (2020) raise a list of possible causes that might result in conflicts based on their investigation of the fixes adopted for build conflicts in a C++ project. Although they do not directly investigate build conflict causes, the reported causes are covered by our results in this study.

3.3.2 Implications

Based on the catalogue of build conflict causes derived from our study, awareness tools such as Palantír (SARMA; REDMILES; HOEK, 2012) could alert developers about the risk of some conflict situations they currently do not support. For example, suppose a developer renames a method while another developer adds a new reference for it. In this case, an awareness tool would alert the second developer about the renaming performed by the first.

Our results are also supporting evidence for some of the warnings currently supported by Palantír. In another scenario, suppose two developers simultaneously add to the same class two methods with the same signature. Palantír could alert the second developer that another developer has introduced that method signature. Our catalogue of build conflict causes could support assistive tools that consider developers' workspace as source information for predicting conflicts. That tool should be aware of changes in dependent classes independently of when changes are performed.

Program repair tools (GOUES; FORREST; WEIMER, 2013) could benefit from the derived catalogue of build conflict resolution patterns, and the study infrastructure, to automatically

[28] Build ID DSpace/DSpace/263379307, Merge Commit 049eb50.
[29] Failures revealed after testing the integrated code.

fix conflicts. For example, part of the *Unavailable Symbol* conflicts could be automatically fixed as follows. Suppose the left parent renames a method, while the right parent adds a call using the old name. The attempt to build the resulting integrated code will break because of the call for an *Unavailable Symbol* (missing method). A program repair tool could rename the newly added call, fixing the build conflict. Our scripts could support such a tool as they automatically identify the new method name by comparing the base and left commits. The same approach could be adopted if the symbol was deleted by one parent. In this case, a repair tool would reintroduce the deleted declaration that caused the build conflict. *Unavailable Symbol* related to missing class declarations should be more carefully handled, as they could have been moved. Instead of always reintroducing the class, which would introduce code duplication and possibly inconsistencies, after further analysis and confirmation, a program repair tool could simply update an import declaration.

In the same way, a program repair tool could also fix build conflicts caused by *Incompatible Method Signature*. This error may happen in two contexts. First, when deleting parameters from a method signature. The tool could then update the broken call adapting it to the new method signature by removing arguments. Second when deleting the declaration of an overloaded method that shares the name, but not the signature, with others in the same class. To fix this error, the tool could reintroduce the missing method declaration.

Most fix commits are authored by one of the contributors involved in the merge scenario. Our assumption is that build conflicts are harder to fix than merge conflicts, requiring one of the contributors to be involved in the fix. This assumption might also reflect often adopted practices of requiring contributors to successfully integrate code before sending contributions to the main repository; for example, in a pull-based development model, it is expected that integrators require changes to contributors until the pull request is stable and can be integrated. Once a developer submits a PR to be integrated, the PR code owns the current project state and the new developer contributions. If the resulting PR code has conflicting changes, build conflicts will be reported at that moment, as previously described. Reviewers may request changes, and developers will update the PR. However, during this process, new errors can be introduced, breaking the build process again. Unless other developers update the project state through acceptance of other PRs, these new errors are exclusively caused by the developer who initially submitted the PR. Another motivation for this behavior is related to build processes of some projects that are hard to set up the CI pipeline (ZAMPETTI et al., 2019), or they are expensive to run it on an individual developer's workspace, leading developers to use the CI service to build/test their contributions.

3.3.2.1 Guidelines for Developers

As we previously discuss, build conflicts can not be avoided unless conflicting contributions are not performed in parallel. Our results reinforce previous studies on the literature that provide a list of guidelines that could be adopted for developers during collaborative software development. Below, we discuss how these guidelines may be used to support developers when dealing with future conflicts.

Pull changes from upstream regularly

Dias *et al.* (2020) investigate the effect of contribution timing on textual merge conflict occurrence. As a result, the authors report that the higher the contribution duration, the higher the chances of textual merge conflict occurrence during a merge scenario. Although they do not investigate build conflicts, we believe their findings are also valid for our context, as the contribution time may influence build conflict occurrence. For example, consider an incoming merge scenario, where a build conflict might occur caused by *Unavailable Symbol*. In this case, a developer must add new calls to a specific method, while another developer must update that method name. If the changes performed by the second developer are already available on the main repository, the first developer pulls these new changes before starting her new task, the upcoming conflict might be avoided. These recent incoming changes would update her local private repository with the most current project state. This way, the first developer would perform her changes with all updated classes, methods, and new changes.

On the other hand, if the first developer pulls these changes, while performing her new task, she may not avoid conflicts, as her current developer workspace is different from the remote repository. However, she might deal with build conflicts, or even less of them, before finishing her task. For example, as she must add multiple calls to the updated method, pulling the remote changes after adding one call would result in one single conflict, preventing the remaining incoming calls from resulting in conflicts.

Push contributions upstream regularly

Complementary to the previous guideline, previous studies report about the benefits of keeping the main repository updated (ZAMPETTI et al., 2020; VASILESCU et al., 2015). This way, they recommend that developers regularly update the remote repository with their new contributions. Once the remote repository owns developers' new contributions, developers may keep their local repositories updated with the latest current project state and benefit from the previous guideline.

Talk to colleagues, that are working on related tasks

Knowing that a colleague is working on a related task or is changing files that may impact another colleague's task may be a warning for starting a conversation and sharing some information. Brun *et al.* (2013) present Crystal, a tool that speculatively merges local developers' repositories and builds and tests the results, aiming to early detect and warn developers about potential textual and behavioral conflicts. In the same way, Sarma *et al.* (2012) present Palantír, a tool that monitors ongoing changes in personal developers' workspaces and continuously shares information about those changes that might result in conflicts. This way, we expect developers to discuss the best way to deal with incoming conflicting contributions, like holding back some parts of a task until the other developer finishes and submits her changes.

Adopt CI practices

Similar to Hilton *et al.* (2016) and Zhao *et al.* (2017), we recommend adopting CI to keep the remote repository consistent and free of polluting changes. This guideline directly impacts the first and second guidelines, as developers would pull and push their changes to the remote repositories.

3.3.3 Build Conflict Repair Prototype

For better assessing this proposal of using our catalogue and study infrastructure to implement a program repair tool that fixes build conflicts, we implemented a prototype that identifies and recommends fixes for build conflicts. We evaluated our prototype on some build conflicts identified in this study. We plan to evolve this prototype and evaluate it with developers in future work.

We structure our prototype in terms of three main execution steps: fault localization, which identifies the *Unavailable Symbol* build conflict and its cause; patch creation, which proposes changes that solve the conflict; and patch validation, which confirms with developers that the proposed solution is valid. To detail these steps, we use as example a broken build in project Swagger Core [30], reflecting an *Unavailable Symbol* build conflict caused by a missing reference for the `op` local variable. One of the developers renames the variable to `apiOperation`, while the other developer adds a new reference for the old variable name.

The main goal of this proposed tool is to handle build conflicts. As verified in this study and previous ones, this kind of problem occurs only in merge scenarios. As the merge commit owns the changes performed during a merge scenario, merge commits are required as input for our proposed tool. In the following sections, we explain how the particularities of build conflicts are reflected in the way the tool works.

[30] Build ID swagger-api/swagger-core/65086450, Merge Commit 657b64b.

3.3.3.1 Fault Localization

For a given merge scenario, the tool first uses our script to check whether the provided scenario has build conflicts. If there is at least one build conflict, the tool uses another script for classifying the conflicts according to the conflict categories in our catalogue. In case of a build conflict supported by the tool, we move on to the next step carrying on the information yielded by the classification script. In the illustrated case, a build conflict caused by *Unavailable Symbol* due to a missing variable. So the tool parses the build logs and finds out that a missing reference for the variable `op` causes the conflict.

The tool adopts our scripts analyzing syntactic information of parent contributions to ensure the missing variable is an actual build conflict. In our example, the local variable `op` was renamed as `apiOperation` by one of the developers. Analyzing just the contributions of the left parent would only observe the variable was renamed. Similarly, analyzing only the right parent would just inform that a reference for the missing variable was added. Considering each analysis separately, none of them present enough information to ensure that updating the variable name would fix the problem. First, it is necessary to be aware of the scope where the variable was renamed; for instance, if two methods present the same local variable declaration, one method's changes do not impact the other method. When analyzing both parents together, the conflict can be confirmed, and then a fix can be recommended.

A more robust tool would perform exactly these substeps, but relying on local build information (instead of depending only on Travis CI) to check conflict occurrence. This tool could also monitor the local Git repository to trigger the fault localization process for each new merge commit.

3.3.3.2 Patch Creation

The tool further analyses the integrated code contributions to better understand the changes that lead to the missing variable declaration issue and create a patch. During this process, the tool must be aware that different changes to the variable declaration demand different resolution strategies. In our example, the local variable `op` was renamed as `apiOperation` by one of the developers. As a solution, the tool proposes to update the dangling variable references to use the new variable name. In case of deleting the missing variable declaration, the tool proposes a solution that consists of reintroducing the deleted declaration.

The identification that the variable `op` is renamed, it is carried on by one of our study scripts, as explained before (see Section 3.1.2.3). For that, the scripts analyze the syntactic diff between the base and each parent commits (see the top of Figure 16). As the tool now must know the new name of the missing variable, we extend the previously mentioned scripts to obtain the required extra information (see bottom of Figure 16).

Figure 16 – Syntactic diff used to identify the new variable name

Regular log used to identify build conflicts
Update SimpleName: op(1540) to ...

New regular log used to identify the new variable name
Update SimpleName: op(1540) to apiOperation on Method read

Source: The author (2022).

3.3.3.3 Patch Validation

In this last step, the tool applies the required changes as proposed in the last step and compiles the resulting code to check if any compilation problems remain. If the proposed solution fixes the conflict without causing any new compilation problem, then the tool informs the build conflict cause, presents the proposed fix, and asks the developer if she accepts the recommended fix. Once the developer accepts the recommendation, the tool creates a new associated commit. Otherwise, the proposed changes are discarded, and no change is added to the project.

Our current implementation can also fix build conflicts caused by *Unimplemented Method*, *Duplicated Declaration* and *Unavailable Symbol Method*. Additional information about our prototype, and its source code, are available online (Appendix 1, 2022). Although the current implementation does not cover all conflict causes reported in this study, based on the conflict types already supported, that are of types *Unavailable Symbol*, *Unimplemented Method* and *Duplicated Declaration*, we believe the tool could have automatically fixed 21% of the build conflicts in our sample if used by the respective development teams. All of them are not among the most effort-prone to fix. The fix applied to the *Unavailable Symbol* of variables is the least effort-prone, as they involve updating a variable name in a specific context. Fixes for *Duplicated Declarations* require more attention, as duplicated methods may have different implementations. Hence, keeping only one of them might introduce unexpected behavior. However, this was not the case in our study, as all duplicated methods were identical. Last, the most effort-prone fixes involve *Unimplemented Methods* as they may involve reference import, which may cause other compilation errors. If an error is reported for whatever reason at any point during this process, the tool undoes all changes, implementing no fix. Regarding future fixes for other cases, the most effort-prone fixes are associated with conflicts caused by *Incompatible Types* and *Incompatible Method Signature*. For these conflicts, the fix should handle reference imports, properly selecting and positioning required variables based on their types; otherwise, errors would arise during the build process. In this way, prior studies on API migration domain may be used to support our prototype tool when dealing with these conflict fixes. For example, Xu *et*

al. (2019) present Meditor, an approach to extract and apply the necessary edits together with API replacement changes; this technique might be extended to support evolutionary changes done during a merge scenario. The most challenging fixes to automate are conflicts caused by *Project Rules*. The fix in these cases involves detecting, understanding, and applying specific guidelines made for a project.

All cases of the currently supported conflict types are similar, so the way the tool deals with them is satisfactory. We do not observe false positives in our results, but we may discuss some potential scenarios. For example, on fixes for *Duplicated Declaration*, if the duplicated methods are not identical, keeping only one might introduce unexpected behavior in the program. However, this was not the case in our study, as in all cases of *Duplicated Declaration* involving methods, these methods share the same method signature and body. So removing any of these duplicated methods fixes the conflict without semantically impacting the resulting code. In the same way, *Unimplemented Method* fixes might also introduce unexpected behavior motivated by adopting an inappropriate method implementation. Last, fixes for *Unavailable Symbol* of variables may introduce false positives if the variable name update is applied on a different scope. For example, consider an attribute and a local variable with the same name but different types; updating one instead of the other would result in a type mismatch resulting in a compilation error. Next, we present how the tool handles build conflicts caused by *Unavailable Symbol* of variables.

3.4 THREATS TO VALIDITY

Our empirical study leaves a set of validity threats that we now explain.

3.4.1 Construct Validity

As explained in earlier sections, given the nature of our sample, the conflict frequencies we observe should be interpreted in the limited scope of conflict occurrences that reach public repositories. As these occurrences might have a greater impact due to their wider visibility, it is worth studying them even if they do not frequently occur in projects that use continuous integration. Studying private developers' repositories would, nevertheless, be also important. Zhao *et al.* (2017) have also experienced that. Their study observes a decreasing trend in builds with errors caused by missing classes and dependencies after projects adopt Travis.

We manually analyzed some reported build conflicts concerning the accuracy of our scripts, like the examples we discuss in this study, confirming they are true positives. Although we did not analyze all reported conflicts, they are detected using the same process. To detect the reported conflicts, we adopt a more conservative approach than related work. For build conflicts in merge scenarios, we extract the causes responsible for the broken build. Next, we extract syntactic information of each parent contribution and observe

conflict occurrence, checking if the error messages are caused by parent contributions. In case our scripts confirm the interference, we compute a new build conflict. As our scripts were created based on manually analyzed cases, some interferences among contributions may not be verified. Instead of assuming these cases as build conflicts and introducing possible false positives in our results, we conservatively categorize them as broken builds caused by post-integration changes.

Regarding how our prototype works, it may adopt different solutions based on each conflict cause. Errors caused by *Unavailable Symbol* of variables are related to missing variable declarations in a specific scope. Depending on the circumstances that caused the missing declaration, the tool may adopt different fixes. If a variable declaration is removed, the tool recommends the reintroduction of the missing declaration. In the case of variable rename, our tool recommends the update of the dangling variable to its new name. Even if a developer changes the variable declaration location in a specific scope, while the other adds a new variable reference, this action does not represent a conflict of *Unavailable Symbol*. The reference is still available. The error fixed by the prototype is related to the use of a variable in a specific scope without its declaration. However, if a developer changes the object type instead of the object declaration, a build conflict of *Incompatible Type* could arise. For this case, other fix patterns should be adopted considering the particularity of this conflict type that is still not handled by our prototype.

3.4.2 Internal Validity

We miss conflicts that appear in merge scenarios that we could not build on Travis. Projects might specify a list of branches for creating builds on Travis. As all builds we create come from the main branch, if it is not in the list, no build process is started, and we have to discard the potentially conflicting scenario. As in related work (KASI; SARMA, 2013; BRUN et al., 2011; LESSENICH et al., 2018; CAVALCANTI; BORBA; ACCIOLY, 2017), we might have missed code integration scenarios, and conflicts, that reach public repositories but do not result in merge commits. This limitation might occur when using git commands such as *rebase*, *squash*, *cherry-pick*, or under smart kinds of fast-forward performed by git merge. It might also happen when stashing changes, pulling from the main repository, and then applying the stashed changes. Since we cannot track integrations on git repositories performed with these advanced options, we might have missed these cases in our study. Thus, our results are actually a lower bound for build conflicts.

As we analyze projects that adopt good practices of software development, like the use of version control system (git), build manager (Maven), and CI service (Travis CI), we may consider that developers perform better merge integrations under these conditions. This way, as we mentioned before when discussing our results, even if conflicts happen during integration, developers might fix them locally and change the original merge integration result by amending the new conflict fix code.

Some of our scripts are limited in the sense that they search for conflicting contributions with partial information and resolution patterns. As discussed in Section 3.1.2.3, for some scenarios, this could bring imprecisions in the conflict confirmation process. So we manually analyzed the risky cases and confirmed that no imprecision occurred. At most, a conflict was classified as a post-integration change that led to a broken build. We adopt an approach based on the commit fix statuses as the starting point for our analysis. If a fix commit fixes the conflict but new errors arise, we move to the next commit instead of already extract the resolution pattern. For resolution patterns, we analyze the syntactic differences applied to the fix commit. In both cases, we perform manual analyses to classify conflicts and resolution patterns. From all conflicts, 27% of them demanded manual analysis, while resolution patterns required manual analysis in 77% of all cases. Both analyses were rather simple and syntactic, and our scripts supported some steps as they inform the files to be analyzed. As we adopt an objective notion and classification of conflicts, we believe the manual analysis performed by one researcher would have the same result if performed by different researchers. Our notion of conflict is based on a broken build (compilation problem during the build process, that is, the presence of an error in a build log) caused by conflicting contributions after a merge scenario. Our classification of conflict causes is based on extracting the cause of the broken build and identifying the changes that caused the breakage across the parent contribution of the merge scenario. As explained in Section 3.1.2.2, our scripts can parse a list of common build error messages. In case a build log presents an error message, not in this list, the log is discarded. It occurred in rare cases when the build log was empty. So our results actually represent a lower bound of build conflicts.

We use GumTree diff to analyze the contributions performed by developers and integrators. However, our approach has some limitations. When we look for a specific class method in the diff, our search uses the method name instead of its signature. It represents a threat for *Duplicated Declaration* and *Unimplemented Method* since a class can have two methods with the same name but different signatures. When we try to classify build conflicts caused by contributor changes, we observe conflicting contributions looking for the method names in the syntactic diffs, which can lead us to a wrong conclusion. Suppose both parents introduce two methods with the same name but different signatures, and in the merge commit, there are two methods with the same signature. In that case, the conflict is motivated by the integrator changes and not by contributors. To dimension the impact of this threat, we manually evaluate all cases of *Duplicated Declaration* methods revealing all cases were well classified.

During the study, some manual analysis was performed by one single person. Although another researcher verified the procedures, such analysis can introduce bias in the results as a false judgment would lead to inconsistencies. As previously explained, our notion and classification of conflict are clear and objective, leaving no room for subjective interpretation

by the evaluator. Furthermore, all manual analysis was supported by information from our scripts, eliminating the chances of errors being done. As explained in Section 3.1.2.2, our scripts can parse a list of common build error messages. Regarding the accuracy of our scripts, they can parse a list of common build error messages. In case a build log presents an error message, not in this list, the log is discarded. However, our scripts were able to classify 99% of all build commits.

3.4.3 External Validity

Our results are specific to the context of open-source GitHub Java projects that use Maven and Travis CI. Although our sample groups different projects regarding the domain, size, and the number of collaborators, most of them are small or medium-sized projects, having received a few or sporadic commits in the last months. In earlier sections, we motivate our choices. We also explain that results could be very different for dynamic languages; most build conflicts we discuss here would actually appear as test conflicts. We analyze build logs looking for specific message patterns associated with problems that cause the build breakage. After a pattern is identified, we perform additional analysis as explained in Section 3.1.2, to ensure a conflict causes the problem. For a new sample, different patterns can arise, demanding script adaptation to handle them. Build systems with less informative log reports, CI services, and tools with less accessible information could hinder replication. Considering only projects adopting Maven as build manager may have filtered out some recent and valid projects. We decide to follow this approach based on the information we could extract from the build process logs generated by build managers. We observe that prior build logs generated by Gradle were shallow or without full complete information. Besides that, we are not aware of how our choices could affect the results.

4 TEST CONFLICTS IN THE WILD

In this chapter, we investigate the detection of test conflicts using unit test generation. As previously discussed, branching and merging are common practices in collaborative software development. They increase developer productivity by fostering teamwork, allowing developers to independently contribute to a software project. Despite such benefits, branching and merging come at a cost— the need to merge software and resolve merge conflicts, which often occur in practice. While modern merge techniques, such as 3-way or structured merge, can resolve many such conflicts automatically, they do not support conflicts when they arise not at the syntactic but the semantic level. Detecting such conflicts requires understanding the behavior of the software changes, which is beyond the capabilities of most existing merge tools. As such, test conflicts can only be identified and fixed with significant effort and knowledge of the changes to be merged. While semantic merge tools have been proposed, they are usually heavyweight, based on static analysis. Here we take a different route and explore the automated creation of unit tests as partial specifications to detect unwanted behavior changes (conflicts) when merging software.

In order to address these limitations, we present SAM, a semantic merge tool exploring the detection of test conflicts through unit-test generation. To evaluate our tool, we perform an empirical study relying on a ground-truth dataset of more than 80 mutually integrated changes' pairs on class elements (constructors, methods, and fields) from 51 software merge scenarios; we manually analyzed all changes and investigated whether test conflicts exist. Next, we systematically explore the detection of conflicts through unit-test generation tools using EvoSuite (the standard and the differential version), Randoop, and Randoop Clean, our modified version of Randoop presented in Section 4.1.1. Additionally, we also propose the adoption of Testability Transformations and Serialization. These transformations are changes directly applied in the original source code aiming to increase its testability during the generation process of test suites. In contrast, the adoption of serialization aims to support tools when creating complex required objects. As a final remark, since some conflicts might be challenging to be detected, we also aim to assess whether the generated test suites can detect general behavior changes and related metrics. This way, we may evaluate how far the tools are to detect conflicts considering some behavior changes might involve conflicting contributions from merge scenarios. Our results show that the best approach to detect conflicts involves combining Differential EvoSuite and EvoSuite applied against programs with Testability Transformations (nine detected conflicts out of 28). Our results bring evidence of the potential of using test-case generation to detect test conflicts as a method that is versatile and requires only limited deployment effort in practice, as well as it does not require explicit behavior specifications. Regarding the detection of Behavior Changes between commit pairs, though EvoSuite was the most

successful tool detecting 53% of all reported changes, there is no combination of tools to detect all reported behavior changes. Finally, we observe that larger budgets —time used for tools to generate test suites— benefit Randoop Clean when generating tests with more diverse objects and multiple target method calls.

The rest of this Chapter is organized as follows: Section 4.1 presents the solutions we propose to detect conflicts. First, we present our semantic merge tool SAM. Next, we present the approaches we use to support the detection of conflicts, like Testability Transformations and Serialization. Then, we present Randoop Clean, our modified version of Randoop. Section 4.2 presents our study setup assessing the potential of detecting conflicts using our proposed solutions. In Section 4.3, we present the results, which are further discussed in Section 4.4. Threats to the validity of our study are discussed in Section 4.5.

4.1 DETECTING SEMANTIC CONFLICTS

This section presents the solutions and approaches we propose to detect semantic conflicts. Initially, we motivate and introduce SAM, our semantic merge tool based on unit testing. Next, we present our different approaches that might benefit the potential of detecting conflicts, like the use of Testability Transformations and Serialization. Finally, we present Randoop Clean, our modified version of Randoop.

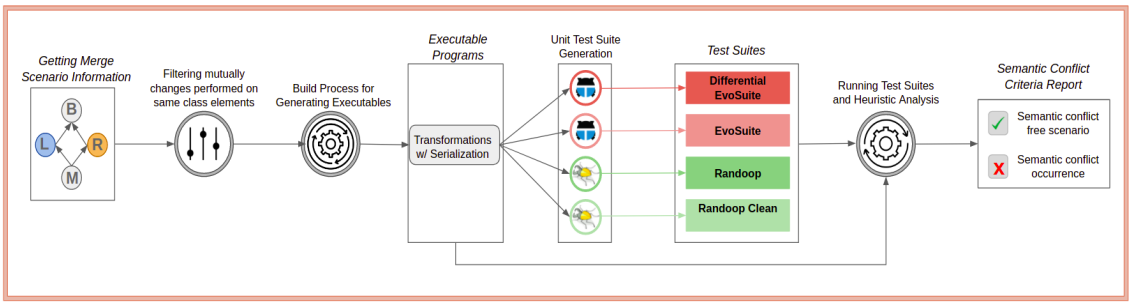
4.1.1 SAM: SemAntic Merge tool based on Unit Test Generation

Detecting semantic conflicts, as previously motivated, is a complex task, and current code integration merge tools do not support this kind of conflict (see our motivating example in Section 2.2.3). As a result, their detection exclusively relies on developers, who should have a good knowledge of the system under development. Alternatively, regression test suites associated with a project could be used to detect conflicts since they might capture unexpected behavior changes during merge scenario integrations. However, even in such cases, it is expected that conflicts escape to production, leading the end users to experience and report bugs later. Aiming to support developers in actively detecting these conflicts during merge scenarios, we present SAM, our semantic merge tool based on unit test generation. The essence of SAM is to generate and execute tests when merge scenarios are performed. These tests are executed over the different commits of a merge scenario, and after interpreting their results, the tools report whether a semantic conflict is detected.

SAM receives as input a merge scenario and, based on the changes performed by developers, invokes unit test generation tools to generate test suites exploring the ongoing changes. Next, based on some heuristics, SAM checks whether test conflicts occur by comparing the test suites' results against the different merge scenario commits. For example,

consider a test case generated by a unit test tool based on the left commit of a merge scenario. When running this test case on the left commit, the test passes as expected, while running it on the base commit, it fails considering that left changes lead the program to a new behavior not held by the base commit. However, if this test case also fails on the merge commit, we may conclude that the changes applied by the right are in conflict with left changes causing the failure. We explain in detail our heuristics later in Section 4.1.1.5. If a conflict is detected, the tool warns developers about its occurrence, informing the class and its element involved in the conflict. Next, we present in detail how SAM works and its different steps (see Figure 17).

Figure 17 – Overview of steps performed by SAM and its workflow.



Source: The author (2022).

4.1.1.1 Starting Point

SAM is a semantic merge tool called when a merge scenario integration is under progress. This way, SAM is called when the `post-merge` *hook* is actived.[1] This hook is responsible for performing additional and specific checks after a successful merge commit is created. If during a merge scenario, no merge (textual) conflict occurs, SAM is called to verify the occurrence of semantic conflicts involving the ongoing parent commits' contributions. If merge conflicts are detected, SAM is not called since these conflicts would require manual fixes by the integrator, eventually leading to new changes not related to the original parent commits. For merge scenarios classified as *fast-forwards*, SAM does not take any action, leaving the default merge tool to lead the integration process. In these circumstances, although the merge commit presents two parent commits, only one parent commit, let us say left, has effectively changed the code under integration, while the other parent, let us say right, does not perform any change.[2] This way, left and merge commits hold the same program behavior. So there is no way the merge commit presents a different behavior compared to the left after integration.

Once SAM is called, the first action is to collect the commits involved in the merge scenario. For that, the tool gets the *merge commit hash* from the head of the current

[1] Git's feature to execute custom scripts from client and server-side.
[2] In this scenario, right and base commits represent the same commit.

branch, while the left and *right commit hashes* are taken calling a git command that informs the parent commits of a given commit; in this case, the previous merge commit.[3] Finally, when the tool gets the parent commit hashes, it uses another git command to extract the base commit hash.[4] After collecting this merge scenario information, the tool advances for the next step, when the parent commits' contributions are explored and mined.

4.1.1.2 Selecting Mutual Changes on Same Class Elements

In this step, the tool explores the changes performed by the parent commits, aiming to collect mutually changed elements (methods, constructors, and field declarations). We adopt this approach as an initial starting point since we are aware that during collaborative development, developers are expected to change different classes of a project based on the goals associated with each task. As a result, these changes might introduce new features to the system (new behavior), which conforms with their individual tasks' goals. However, when different tasks are integrated, they might impact each other's goals leading the system to a state of unexpected behavior. Considering the spectrum of changes that different contributions can do, it is hard to determine whether a pair of changes are conflicting based only on the changes performed without considering their semantics. For example, conflicting contributions may or may not occur when changes are simultaneously applied to the same method. In the same way, a conflicting situation may or may not occur when one developer changes a method `m()`, which is called by the new changes done by another developer from a method `n()`, and so on.

This way, in order to restrict the spectrum of changes that might result in conflicting contributions currently covered by the tool, SAM initially seeks for parallel changes applied to the same class elements (fields, constructors, and methods). We consider that the chances one developer's contributions unexpectedly affect others might be higher under such situations, as these changes might modify the same local variables, for example, changing the method behavior. So the second step of SAM is to mine the mutual elements changed by both parent commits during a merge scenario. For that, SAM collects the set of changed Java class elements by each contribution using DiffJ[5] and then performs an intersection between these elements; if the intersection is not empty, the tool moves to the next step. Otherwise, the merge operation continues with no change in its usual workflow.

On the other hand, in case a conflict occurs when dependent methods are changed in parallel, SAM is currently missing these cases. However, we believe SAM would still work in cases like this, considering that the generation of test suites might be performed based on the classes holding the target changed and impacted method. This way, SAM

[3] git cat-file -p commit_id
[4] git merge-base left-hash right-hash
[5] https://github.com/jpace/diffj

might explore a new approach to determine and identify dependent methods changed and impacted in parallel.

4.1.1.3 Generating Binary Files

In this step, our tool focuses on getting the required inputs in order to generate the test suites in future steps. Since SAM uses unit test tools to generate the test cases, we must feed the tool with binary file versions of the target code we want to assess.[6] For that, at this point, SAM generates one binary file of the system program for each commit of the merge scenario (base, left, right, and merge commits). This information is given as output from the previous step when the tool collects the four commit hashes involved in a merge scenario. Binary files of all versions of the system corresponding to the commits are required because our tool detects conflicts by comparing the test results against the different commits. Later, in this section, we explain our heuristic to detect conflicts.

Next, the tool performs a sequence of checkouts for each commit hash generating the associated binary file. For that, SAM locates and instruments the *pom.xml* file of a given commit, adding a new plugin called *Maven Assembly Plugin*,[7] which holds the required information to generate the binary file with all related dependencies. This plugin presents different options to describe the future generated binary file; however, since SAM only needs the source code and associated dependencies, a default configuration is used in this context. Once this instrumentation is performed, SAM calls the build manager to compile the code while also invoking the just added plugin. As a result, a jar file is created and released on the *target* directory of the project. As a final step, SAM collects the generated binary file, while discarding all previous changes applied on the *pom.xml* file.

Aiming to increase the testability of the target code under analysis, SAM adopts extra approaches like the adoption of Testability Transformations and Serialization. These transformations are changes directly applied in the original source code, while the adoption of serialization is based on the generation of serialized objects resulting from executing the original project test suite. We explain in detail these two approaches in Sections 4.2.2.2 and 4.1.3, respectively.

4.1.1.4 Generating Test Suites

With the set of elements mutually changed by the parent commits and the associated merge scenario commit binary files, SAM starts the next step responsible for generating and executing test suites. The generation of test suites is driven by the class where the mutually changed elements are located. When applied, the generation is also driven by the constructor/method changed by the parent commits, aiming to directly explore the code where the conflict takes place. Our current implementation generates test suites for

[6] A binary file means a jar file with all classes associated with the project and required external dependencies.
[7] https://maven.apache.org/plugins/maven-assembly-plugin/

both parents' commits as these commits are responsible for introducing the changes that could be conflicting in the merge commit. As we previously mentioned, SAM currently starts to work when detecting elements mutually changed by both parent commits. In the case of mutually changed methods, SAM might drive the generation of test suites to directly explore these methods. Otherwise, the generation of test suites is driven by the target class holding the mutually changed element.

Since SAM may consider different unit test tools to generate test suites, we propose here our tool called Randoop Clean, a modified version of Randoop. In Section 4.1.4, we present this tool highlighting its changes compared to Randoop and its workflow.

4.1.1.5 Conflict Detection based on Merge Scenario Commits' Behavior Heuristic

After generating the test suites, giving the left and right binary files of the system for the test generation tools, SAM executes the generated tests against the merge scenario commits. The current implementation executes each test suite three times against each of the four commits, aiming to reduce the chances of relying on the results of flaky tests. When the executions are finished, SAM collects the results achieved by each commit for a given test suite and checks whether any test case satisfies one of our conflict criteria. Next, we present our conflict criteria, their motivation and our notion of partial specifications.

Conflict Detection Criteria

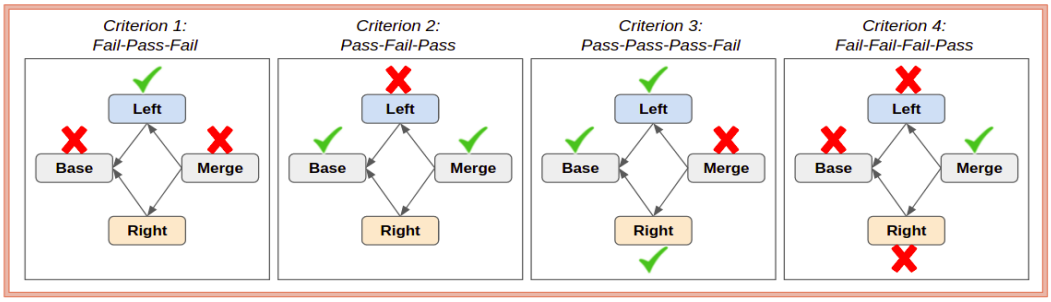
In the previous section, we present the generation of test suites adopted by SAM. Once these suites are generated, we use a set of conflict criteria to detect and report the occurrence of conflicts. Figure 18 represents the four conflict criteria we consider for our tool. We believe these criteria cover common situations of conflicts involved by different contributions.

As we discuss in Chapter 2, current merge tools do not support developers on the detection of semantic conflicts. To reduce this discussed difficulty and the costs associated with test conflict detection and resolution by original project test suites, we might think about other approaches that could reveal the kind of interference we investigate here.

This way, suppose a regression test generation tool —such as Randoop (PACHECO et al., 2007) or EvoSuite (ALMASI et al., 2017; FRASER, 2018)— generates the test in Figure 19 when given the left commit as input (see Figure 5 of our motivating example in Chapter 2 for additional details).

That test passes when executed against the left commit, which leads to a call to the `normalizeWhitespace()` method when executing `cleanText()` in Line 6 of the test. With the input illustrated in `test1`, when reaching Line 6, `t.text` stores the test input string in Line 5, except for the extra space character right before `dog`. Consequently, the assert successfully evaluates. The same test breaks when executed against the base commit, since this revision involves no call to `normalizeWhitespace()`, and so the assert throws

Figure 18 – Heuristics for conflict detection criteria. Each conflict criterion is based on the output of a given test considering the analyzed versions of the system corresponding to the merge scenario commits. The green checkmark stands for a test with a passed output, while the red X stands for a failing output.



Source: The author (2022).

Figure 19 – A test case revealing an interference from the example presented in Figure 5.

```
class TextTestSuite {

  public void test1() throws Throwable {
    Text t = new Text();
    t.text = "the␣the␣ ␣dog";
    t.cleanText();
    assertTrue(t.noDuplicateWhiteSpace());
  }
}
```

Source: The author (2022).

an exception. For this reason, breaking in the base commit and passing in the left commit, we say that `test1` partially reveals the intention of the commit left based on the base commit. We can see `test1` as a partial specification of the changes of the left commit.

Now note that `test1` fails when executed against the merge commit in Figure 5. The `normalizeWhitespace()` ends up being called when executing `cleanText()`, but the call to `removeDuplicatedWords()` leads to a new duplicated whitespace in the text, as explained before. This way, `test1`, the partial specification of left commit, is not satisfied in the merged version, revealing that the changes done by right commit interfere with the left commit (with respect to the base commit). Applying this idea for software development in practice, if we find a test that satisfies the just mentioned criteria, a *test conflict* might be detected (first two criteria on Figure 18).

Now, consider a slightly different scenario, where the parent commits aim to eliminate a current redundancy state of the method `cleanText()` (see Figure 20). The illustrated class `Text` results from a merge that integrates the change in green (Line 8, say from a revision left) with the change in red (Line 13 was added, say from a revision right). Just like our previous example, as the source code in the range of lines 9 and 12 separate the two changes to be integrated, there is no syntactic merge conflict in this case, and we cleanly obtain the syntactically valid class in the figure.

Figure 20 – A merge of two changes (each parent removed one of the highlighted lines) that are semantically conflicting.

```
1   class Text {
3     public String text;
5     void cleanText() {
        this.removeDuplicatedWords();
7       this.removeComments();
-       this.normalizeWhitespace();
9     }
11    void removeDuplicatedWords() {
          ...
13  -     this.normalizeWhitespace();
      }
15
      void normalizeWhitespace() {
17        ...
      }
19
  }
```

Source: The author (2022).

The primary purpose of the method `cleanText()` is still the same of the previous example, that is to apply some string cleaning through side effects. For that, it calls additional methods to remove duplicated words (Line 6), comments (Line 7), and normalize whitespace (Line 8). The mentioned method calls were added in a previous merge scenario when the involved developers, during their revisions, added independent calls to the method `normalizeWhitespace()` causing the current redundancy state. As a result, when these revisions were integrated, the developers observed redundant and unnecessary calls to the discussed method. Someone may argue that these redundant calls could have been avoided by establishing a good communication channel between the involved developers. However, considering the current collaborative software development, when developers can be in different countries and time zones, and the assigned tasks have different goals and no intersection, it is plausible to understand the no active communication.

Aiming to eliminate the current redundancy state, the developers, without communicating with each other, decided to, partially or entirely, undo their changes by eliminating their duplicated method calls. While Green removes her method call in Line 8, Red removes such a method call in Line 13 (see Figure 5). As a result, after integrating these revisions, there is no call left to `normalizeWhitespace()`, though this method is still declared in the `Text` class (Lines 16-18 in Figure 20). Notably, we observe *interference* involving the revisions left and right. The resulting merged code (revision merge) does not have treatment to normalize whitespace, a behavior expected to be preserved as all previous revisions held it.

Different of the previous merge scenario, this time, as the parent commits perform refactorings, even if we use unit test tools to generate tests exploring the changes performed

by the parents, the tests would present the same behavior in base and parent commits. However, we are aware that a test conflict happens in this scenario. This way, although our previously introduced criteria detect conflicts, some of them might escape requiring new criteria, which could be used to detect the conflict of the current example. Although we need new conflict criteria, the test case presented in Figure 19 can still be used to detect the conflict in this case.

That test passes when executed against revision left, which leads to a call to `normalizeWhitespace()` when executing `removeDuplicatedWords()` (Line 6 in `Text` class). At that time, the parent revision right has not removed his duplicated method call yet (Line 13). With the input illustrated in `test1`, when reaching Line 6, `t.text` stores the test input string in Line 5, except for the extra space character right before `dog`. Consequently, the assert successfully evaluates. Executing this test case against revision right, we also observe the test passes as there is a call to `normalizeWhitespace()` (Line 8); at that time, the revision left has not removed its duplicated method call yet. Finally, the same test case also passes when executed against revision base since that revision has two calls to `normalizeWhitespace()` (Lines 8 and 13, respectively). For this reason, passing in base and both parent revisions right and left, we say that test1 partially reveals the behavior of method `cleanText()` was preserved by each revision left and right, and therefore we expect such behavior to be preserved in the merge too.

Now note that `test1` fails when executed against revision merge in Figure 20. The method `normalizeWhitespace()` ends up not being called when executing `cleanText()` as both parent revisions removed their calls to that method, as explained before. This way, `test1`, the partial specification of left, is not satisfied in the merged version, revealing that the changes in right interfere with left (with respect to base). If we could find a test that passes in revisions base, left, and right, and the same test fails in revision merge and vice-versa, we would similarly say that parent revisions left and right might interfere with each other (last two criteria on Figure 18).

Since our conflict criteria rely on the final statuses of test cases executed in different commits, we must further comment about statuses different than *passed* and *fail*. For test cases that present *error* statuses, we opt to not consider them when reporting conflicts. We believe that considering these cases might introduce false positives in our results. For example, by executing test suites in different versions of a program, if a specific version has a call to an external service and no answer is received, the execution would break due to unexpected trhown exceptions or other unexpected behavior. The *error* status might be caused by external dependencies, and not changes performed by the parent commits of a merge scenario.

As a final remark, it is important to discuss that our conflict criteria are valid to detect conflicts if the associated test cases explore the conflicting changes integrated during a merge scenario. Otherwise, false positives might be reported. For example, consider a test

case that satisfies our first conflict criterion (see Figure 18), when it passes on left and fails on base and merge commits. In case this test case fails on base or merge commits due to unrelated changes to the conflicting contributions, this test case should be discarded.

Although we previously advocated the potential of adopting Randoop or other tools to detect conflicts, there is no guarantee that they always generate a relevant test considering its randomness. For example, tools may generate tests without covering all methods of a class. However, particularly in our motivating example, we know a conflict occurs on a specific method. So we expect the tools to cover that method and preferably call it many times, increasing the chances of detecting the conflict.

The detection of conflicts we adopt for SAM is based on the idea of *partial specifications*, as we explain in detail as follows.

Test Cases as Partial Specifications

Although we motivate the occurrence of conflicting parent contributions, we may not ensure these contributions will result in conflicts, as we do not have access to their formal specifications. Even if we had access to these formal specifications, we believe they still might miss some scenarios of conflicting contributions, considering the multiple tasks performed in parallel. However, we have access to the code under integration, which represents the intention and goal of developers when performing a task. With that in mind, we propose and explore here the concept of *partial specification*.

This concept is related to the ability to generate test suites for the merge scenario parent commits' changes, assuming these tests as partial specifications of the changes performed by each parent commit. This way, as we generate test suites that might explore the changes performed by one parent, we may assume that test cases can be used as specifications exploring the current behavior of the changed method. For a *full specification*, the generated test cases should explore all changes performed by both parents of the same target method. So based on our first conflict criterion (see Figure. 18), if a test generated based on the left commit passes on its execution, it shows the changes implemented by the left satisfy the behavior explored by the test. On the other hand, if the same test fails on the base and merge commits, respectively, it shows these commits present different behaviors that do not satisfy the test case under analysis. The test case failing in the base commit means the intention of the changes done by the left was achieved by changing the program behavior related to the base. In contrast, the test case failing in the merge commit means the program behavior proposed by the left parent is no longer available, as its changes conflict with the other parent's contributions.

4.1.1.6 Report of Semantic Conflict Occurrene

Once a test case satisfies one of our conflict criteria, our tool warns the developer about a potential conflict occurrence by informing the element where the conflict takes place, as

also the test that reveals the conflict. Then, the developer might evaluate whether the reported conflict represents an actual conflict or not. If so, she may apply changes in order to fix the conflict and change the current merge commit; otherwise, she skips the warning leading the merge commit without applying any change.

4.1.2 Testability Transformations

Previous studies report that unit test tools face problems when directly calling specific methods of a target class (SILVA; ALVES; ANDRADE, 2017). Regarding the attempt to reach `private` methods, in our previous study (SILVA et al., 2020), we observe tools can only reach these methods by calling `public` methods, which have direct calls to these target `private` methods. Aiming to address these issues, we propose here the adoption of three Testability Transformations aiming to increase the testability of the code under analysis by the unit test tools. This way, the tools might direct call and access elements of a given class. The first proposed transformation is responsible for replacing `no-public` access modifiers of classes and their elements for `public`; so, this transformation might be applied to classes, methods, and fields declarations. Next, the second transformation aims to add a new empty constructor in a given class if no empty constructor is already available. Finally, the last transformation is responsible for extracting inner classes from their enclosing classes. The first two transformations are supported by an automated tool, while the last transformation is manually applied.

Figure 21 – A conflict occurs and is propagated trhough a private field.

```
public class SlimTableFactory {
  private final Map<String, Class<? extends SlimTable>> tableTypes;

  public SlimTableFactory(){
    tableTypes = new HashMap<String, Class<? extends SlimTable>>();
    tableTypes.put("dt:", DecisionTable.class);
    tableTypes.put("decision:", DecisionTable.class);
+   addTableType("ddt:", DynamicDecisionTable.class);
+   addTableType("dynamic decision:", DynamicDecisionTable.class);
    {...}
+   addTableType("script:", ScriptTable.class);
   }

   {...}
  }

}
```

Source: The author (2022).

The application of these Testability Transformations is motivated by preliminary experiments we performed using the unit test generation tools with toy examples and a small subsample of the scenarios we consider here. As guidelines, we used four scenarios of our sample. For example, consider the example presented in Figure 21, when the class

constructor is changed by both parent commits.[8] While the left parent adds two new elements in the field `tableTypes` (lines 8 and 9), the right commit adds a new element (line 11). Although the *interference* state is reached, it was *propagated* through the private class *field*. As a result, the generated tests could not detect the interference because they would not directly access the involved private field. The tests also did not invoke additional methods that had access to such attributes. So the interference could be *reached*, the *infection* could occur, but the *propagation* could hardly be observed. For being able to detect interference in such situations more easily, we present our first transformation by applying source code transformations to increase testability, with a possible impact on correctness, as assessed in the last step of our study. In this case, the transformation replaces `no-public` access modifiers with `public` access ones from the class that contains the method or field declaration that two developers independently changed. We apply this transformation for classes and their methods, constructors, and fields.

By replacing all non-public elements for public ones, the transformations aim to increase the testability of the code under analysis, considering the number of elements possible to be called by test suites might increase. As a result, the tools would use part of the generation budget for directly calling elements not involved in the test conflict. This way, someone may argue we should apply these transformations only in the element involved in the semantic conflict in order to maximize its use by the test suites. We are aware of this issue; however, we believe calling non-related elements involved in the conflict is beneficial since these calls might change object states leading to conflict detection.

Regarding our second proposed transformation, we observed that the unit test generation tools could not even reach the interference locations during our initial experiments. This occurred because the tools could not create objects of the class that should be exercised to reveal the interference of classes that appeared as parameters of methods in the generated test. Aiming to address this issue, we propose a transformation to add an empty constructor to the class under analysis if a non-empty constructor was unavailable. An empty constructor is relevant when the tools cannot create complex objects with internal and external dependencies. If a class extends another one (superclass), no empty constructor is added in the first class, as this transformation would require adding another constructor in the superclass. This way, when the transformations are applied, we inform if we want to fully or partially apply them. Finally, for our last transformation, for scenarios where the independently changed declarations occur inside inner classes, we extracted them to the outer level, as the test generation tools could not directly exercise inner classes.

As a result, we present these transformations as an executable jar file, which covers the two first presented transformations. This file receives the local path of the target class under analysis as input. A semantic merge tool could transparently apply such

[8] Fitnesse - merge commit: 4d9ba9d

transformations by adding this jar file as a dependency. We expect no major negative implications for users of the tool. Applying the transformations is computationally not expensive compared to generating and executing tests.

4.1.3 Serialization

Figure 22 – Class with complex required objects, given as input for unit test tools.

```
1  public class BuildRequest {

3      private Converter converter;
       private RestMethodInfo methodInfo;
5      private String apiUrl;

7      public RequestBuilder(Converter converter){...}

9      private Request build() throws UnsupportedEncodingException {
           {...}
11         String apiUrl = this.apiUrl;
           StringBuilder url = new StringBuilder(apiUrl);
13         if (this.methodInfo.hasQueryParams) {
               boolean first = true;
15         }
           {...}
17         return new Request(this.methodInfo.requestMethod, url.toString(),
               headers, buildBody());
       }
19
   }
```

Source: The author (2022).

Based on our previous findings (SILVA et al., 2020), test generation tools face problems when creating complex objects required to be used by test suites. For example, consider the class `BuildRequest` is given as input for a unit test tool (see Figure 22) aiming to explore the method `build()` (line 9).[9] In order to call the target method, the unit test tool must generate a `BuildRequest` object, which might be later used to perform the call. Although the class `RequestBuilder` has a public constructor, it requires as input a parameter (`Converter`, line 7), which is provided by external dependencies. So in order to generate an object for the class `BuildRequest`, the tool must also generate all further required objects. Similarly, the target method `build()` must access the field `methodInfo` (line 13), which is a local class in the project but with multiple dependencies.

However, the unit test tool is not able to generate valid objects of the required types (`Converter` and `RestMethodInfo`) due to the complexity associated with these objects. Alternatively, the tool generates a test case with the required objects by assigning to them `null` values (see Figure 23). When running this test case, consider that during its execution, no call to the `null` object `Converter` is done. So after establishing the fixture, the target method `build()` is invoked (line 6 in Figure 23). However, once the `if`

[9] Retrofit - merge commit: 71f622c

statement is reached (line 13 in Figure 22), a `NullPointerException` is trown, as the field `methodInfo` is `null` and has no field `hasQueryParams`.

Figure 23 – Irrelevant test case generated by unit test tools due to not well-formed objects.

```
1  @Test
   public void test1() throws Throwable {
3      RequestBuilder requestBuilder0 = new RequestBuilder(null);
       requestBuilder.methodInfo = null;
5      try{
        requestBuilder0.build();
7       fail("Expected exception of type NullPointException.");
     } catch (NullPointException e) {...}
9  }
```

Source: The author (2022).

Aiming to address this limitation, consider that we could feed unit test tools with additional information, which might be used to generate relevant instances of the required objects. This way, consider a unit test tool with further support that might parse a XML file (see Figure 24), which owns the required information to create a `BuildRequest` object with valid values assigned to all of its fields. For example, the field `converter` and `methodInfo` are instances of the classes `GsonConverter` and `RestMethodInfo`, respectively (see lines 2 and 8 in Figure 24, respectively). Calling the tools again, consider that this the test case presented in Figure 25 is generated. When running this test case, we observe that no exception is expected to be thrown, as valid values are assigned to all required objects. Furthermore, the target method `build()` is called (line 4), and its returning `Request` object is explored on the generated assertion (line 5).

Figure 24 – Example of a serialized object given as input for unit test tools.

```
1  <RequestBuilder>
     <converter class="GsonConverter">
3      <gson>
         <calls><threadLocalHashCode>865977</threadLocalHashCode></calls>
5        {...}
       </gson>
7    </converter>
     <methodInfo>
9      <hasQueryParams>false</hasQueryParams>
       {...}
11   </methodInfo>
     <apiUrl>"http://example.com"</apiUrl>
13   {...}
   </RequestBuilder>
```

Source: The author (2022).

Based on this example, we propose here the generation of binary files with serialized objects and Testability Transformations (previously discussed) and feeding the tools with these new binary file versions. This way, the tools would access well-formed objects created by project tests by deserializing and directly using them on test suites. As a result, the tools

Figure 25 – Relevant generated test case using serialized objects as input.

```
@Test
public void test1() throws Throwable {
    RequestBuilder requestBuilder0 = new RequestBuilder(null);
    Request request0 = requestBuilder0.build();
    assertEquals("http://example.com", request0.getApiUrl());
}
```

Source: The author (2022).

might generate test cases reaching the method or field (holding the semantic conflict) with valid values, without throwing irrelevant exceptions because of not well-formed objects.

To create those new binary files, we implement OSean.EX, a tool responsible for generating serialized objects based on a specific target method. First, OSean.EX instruments the target method, previously given as input, by adding a method call for a new assistant class method (`ObjectSerializerAssistant.serialize(Object)`), previously added in the current project. For example, for the method `build()` previously introduced (see Figure 22), its resulting instrumented code is presented in Figure 26 (line 4). This method `serialize` is responsible for serializing the current class object and eventual parameters received by the target method. Next, the tool runs the original project test suite for a specific amount of time given as input, creating new unique serialized objects each time the target method is reached; eventually, duplicated objects are discarded.

Figure 26 – Instrumentation applied on the target method `build()` by OSean.EX.

```
public class BuildRequest {
    {...}
    public Request build() throws UnsupportedEncodingException {
        ObjectSerializerAssistant.serialize(this);
        {...}
    }

}
```

Source: The author (2022).

When the project test suite execution is finished, a new class is added to the target project (`SerializedObjectAssistant`); this class holds a list of methods, each one returning a specific previously serialized object. Running OSean.EX for our example presented in Figure 22, the resulting `SerializedObjectAssistant` class would provide four different types of objects: the class holding the target method (`RequestBuild`) and its fields (`GsonConverter`, `MethodInfo`, and `String`, see Figure 27). Additionally, a new `SerializedObjectAssistant` field on the target class is added, aiming to provide access to EvoSuite and related tools to these serialized objects, as they do not support multiple classes as input.

OSean.Ex accepts a list of commits from a project as input. This way, the serialized objects are exclusively generated based on the first commit of this list. Posteriorly, the other commits reuse these serialized objects, deserializing them based on their versions.

Thus, for these remaining commits, OSean.Ex only performs the last step generating the binary file. The costs to use this approach are based on the amount of time used to execute the project test suites. Since developers can inform the amount of time the tool may spend running the project test suites, they may decide the approach that better fits their needs.

Figure 27 – Providing serialized objects through individual method declarations.

```
public class SerializedObjectAssistant {

    public RequestBuilder deserializeRequestBuilder1() {...}
    public RequestBuilder deserializeRequestBuilder2() {...}
    {...}
    public Converter deserializeGsonConverter1() {...}
    public Converter deserializeGsonConverter2() {...}
    {...}
    public MethodInfo deserializeRestMethodInfo1() {...}
    public MethodInfo deserializeRestMethodInfo2() {...}
    {...}
    public String deserializeApiUrl1() {...}
    public String deserializeApiUrl2() {...}
    {...}
}
```

Source: The author (2022).

4.1.4 Randoop Clean

Based on weaknesses reported by our previous study and how Randoop works (SILVA et al., 2020), we decide to apply improvements to address some of these issues, particularly for this study. For example, recall the semantic conflict that takes place in the method `cleanText()` as previously presented in Section 2.2.3. Invoking Randoop to generate tests for this associated program, consider the test cases presented in Figure 28 are generated. Although the generated tests directly call the target method `cleanText()` (lines 5 and 16), we believe many calls as possible to this method might increase the chances to detect the conflict. To address this weakness, SAM forces additional calls to the target method given as input. Figure 29 presents an expected test class generated for this scenario highlighting the increase of calls to the method `cleanText()`; in this case, one new call to the target method compared to Randoop.

Still, regarding the generated tests by Randoop in Figure 28, we observe that the same input is used when `Text` objects are created (lines 4 and 17). This happens because Randoop reuses previously created objects and their declarations based on the object types required when calling a specific method. However, calling a method with the same inputs may reduce the chances of a conflict being detected. Aiming to address this issue, Randoop Clean forces the creation of new objects instead of constantly reusing them (lines 4, 11, and 19 in Figure 29). This way, we believe that the chances to detect a conflict increase considering the target method `cleanText()` is called with different inputs.

Figure 28 – A test suite generated by the original version of Randoop.

```
1  @Test
   public void test1() throws Throwable {
3    {...}
     Text t = new Text("the house is blue");
5    t.cleanText();
   }
7
   @Test
9  public void test2() throws Throwable {
     {...}
11   t.updateCodeFormat();
   }
13
   @Test
15 public void test3() throws Throwable {
     {...}
17   Text t = new Text("the house is blue");
     t.cleanText();
19 }

21 @Test
   public void test4() throws Throwable {
23   {...}
     t.noDuplicateWhiteSpace();
25 }
```

Source: The author (2022).

Figure 29 – A test suite generated by Randoop Clean.

```
1  @Test
   public void test1() throws Throwable {
3    {...}
     Text t = new Text("the house is blue");
5    t.cleanText();
   }
7
   @Test
9  public void test2() throws Throwable {
     {...}
11   Text t = new Text("the car is blue");
     t.cleanText();
13
   }
15
   @Test
17 public void test3() throws Throwable {
     {...}
19   Text t = new Text("the house is red");
     t.cleanText();
21 }

23 @Test
   public void test4() throws Throwable {
25   {...}
     t.noDuplicateWhiteSpace();
27 }
```

Source: The author (2022).

The Algorithm 2 shows how our proposed tool Randoop Clean works. Since we preserve most behavior of the original Randoop, we highlight with blue lines here only the changes we implement on Randoop Clean. First, before Randoop starts to generate tests, it selects all public methods and constructors from a given list of classes given as input and puts them in a pool. Next, Randoop uses this pool to randomly select an element and generate sequences, which are used to create test cases (line 7 in Algorithm 2). In Randoop, sequences represent inputs for the tests, for example, object creation and method calls. If a particular method is expected to be covered, that method can also be given as input to the tool. However, in our context, we want to maximize the number of calls to a given method.

Algorithm 2 Generation of sequences adopted by Randoop Clean.

Data: $classes, contracts, filters, timeLimit, targetMethod$
Result: List of valid and invalid sequences
// We highlight by blue the lines that are new to Randoop Clean. The other lines are as in the original Randoop (PACHECO et al., 2007).

```
1  errorSeqs ← {} // Contract violations
2  nonErrorSeqs ← {} // No contract violations
3  requiredClasses ← selectRequiredClasses(classes)
4  numSteps = 0
5  while timeLimit.isReached() do
       // Create new Sequence
6      numSteps++
7      m(T1...Tk) ← GetPublicMethod(classes, requiredClasses, numSteps)
8      ⟨hseqs, vals⟩ ← randomSeqsAndVals(nonErrorSeqs, T1...Tk)
9      newSeq ← extend(m, seqs, vals)
10     if newSeq ∈ nonErrorSeqs ∪ errorSeqs then
11         continue
12     end
       // Execute new sequence and check contracts.
13     ⟨o⃗, violated⟩ ← execute(newSeq, contracts)
       // Classify new sequence and outputs
14     if violated = true then
15         errorSeqs ← errorSeqs ∪ {newSeq}
16     else
17         nonErrorSeqs ← nonErrorSeqs ∪ {newSeq}
18         setExtensibleFlags(newSeq, filters, o⃗)
19     end
20     if nonErrorSeqs.size()%(10 * requiredClasses.size()) == 0 then
21         newSeq ← extend(targetMethod, seqs, vals)
22     end
23 end
24 return ⟨nonErrorSeqs, errorSeqs⟩
```

To address this idea of maximizing target method calls, Randoop Clean removes some randomness by optimizing the number of calls to a target method. Therefore, the tool

internally has access to a list of valid sequences generated during the test suite creation (`nonErrorSeqs`, line 2 in Algorithm 2). Based on the number of classes given as input to the tool (`requiredClasses`, line 3 in Algorithm 2), Randoop Clean constantly checks the list length verifying whether a specific number of new sequences were created (line 20 in Algorithm 2). Each time this number is achieved, the tool adds a new call to the target method (line 21 in Algorithm 2). We adopt an interval among new target method calls as we expect other methods to be called. This way, objects required as parameters or holding the target method can be changed, allowing the target method to be executed against different objects.

To generate diverse objects required by the target method, Randoop Clean adopts a similar idea presented for the feature previously explained. Since the creation of objects occurs through classes' constructor or method calls, Randoop randomly selects calls from the pool, which holds the previously available methods and constructors. In our context, randomness might represent a weakness as a specific object could be generated and continuously used when required. A method could be called with the same objects, consequently producing the same results. Therefore, we opt to add new constructor or method calls that return objects of a specific type in order to increase the chances of generating diverse objects.

In a nutshell, Randoop Clean is expected to maximize the number of direct calls to a method and create diverse objects involved in the previous method call. We believe more calls to the target method associated with multiple objects used as parameters might increase the chances of detecting the conflict. By Randomness of Randoop, we mean the feature of randomly selecting methods and calling them. This way, Randoop Clean partially eliminates this randomness by adding additional calls to a target method, previously informed.

To address this idea of maximizing the object generation calls, Randoop Clean internally has access to the number of all attempts made to generate sequences (`numSteps`, line 4 in Algorithm 2), which may result in invalid or valid sequences (`errorSeqs` and `nonErrorSeqs`, lines 1 and 2 in Algorithm 2, respectively). Based on the number of classes given as input to the tool, Randoop Clean constantly checks whether a specific number of attempts were performed. Each time this attempt number is achieved (line 1 in Algorithm 3), the tool selects calls to the creation of objects required by the target method. For instance, creating objects of the class holding the target method and required parameters. Since multiple constructors and methods might create or return objects of a specific type, the tool randomly selects an element of the pool, generating a new object based on the required object type.

Algorithm 3 Selection of method or constructor based on the number of steps done during the generation process.

Data: $classes, requiredClasses, numSteps$
Result: A random public method or constructor from the set of target classes

```
1 if numSteps%(10 * classes.size()) < 2 * classes.size() then
2 |   return selectRandomMethod(requiredClasses)
3 else
4 |   return selectRandomMethod(classes)
```

4.2 EMPIRICAL STUDY

This section presents our empirical study assessing the occurrence and detection of test conflicts using the solutions presented in Section 4.1. Initially, we present the related research questions we investigate in this study. Next, we explain in detail our study setup.

4.2.1 Research Questions

As previously motivated in Section 2.2.3, resolving integration conflicts, like test conflicts, is time-consuming and an error-prone activity that negatively impacts development productivity. While previous studies have investigated the occurrence of these integration conflicts (KASI; SARMA, 2013; BRUN et al., 2013), for test conflicts, they focus only on the frequency of these conflicts analyzing a few Java projects (three and four, respectively). Kasi and Sarma (2013) report that test conflict rates range from 5% to 35%, while Brun *et al.* (2013) report rates ranging from 5% to 28%. However, this is not the main focus of the studies. Since they do not check the program behavior of all merge scenario commits as we do here, their results might have false positives. Additionally, the approach adopted by The author for detecting conflicts relies on the original project test suites under analysis. If the test suite has low quality and coverage, no conflict might be detected. We discuss related work in detail in Section 5.2. In this way, it is important to evaluate the occurrence of conflicts using other approaches. This study aims to assess test conflicts using unit test generation to answer the following questions.

First, we aim to assess whether test generation might be used to detect conflicts. To answer the mentioned question, we select merge scenarios and unit test generation tools for an empirical study. Next, we look for generated test cases that satisfy our conflict criteria revealing the interference. Unlike the previously mentioned studies, our detection of conflicts considers all project versions of a merge scenario; we explain it in detail in the following sections. Finally, based on the adoption of unit test tools, we also discuss the potential to detect semantic conflicts using a semantic merge tool.

As we cannot assess semantic conflict occurrence without having access to the developers' intentions or specification of the changes they make, we use the term *interference* as a proxy to conflicts (see Section 4.1.1.5). The term interference in this chapter refers to

parents' contributions that semantically interfere with each other (see Section 2.2.3). We explain it in detail over this chapter.

Second, once the generated test suites are available, we also aim to evaluate the potential of the related tests regarding the detection of general behavior changes. Answering this question, we believe we might assess how close the unit test tools are to detect conflicts, for the cases that the reported behavior changes involve the conflicting contributions. Third, as we propose a new version of Randoop, we aim to assess the quality of their generated test suites. This way, we compute some related metrics specifically for Randoop and Randoop Clean. For example, we compare the diversity of handled generated objects required by the target methods by each tool, as also the number of direct calls to these target methods and achieved code coverage.

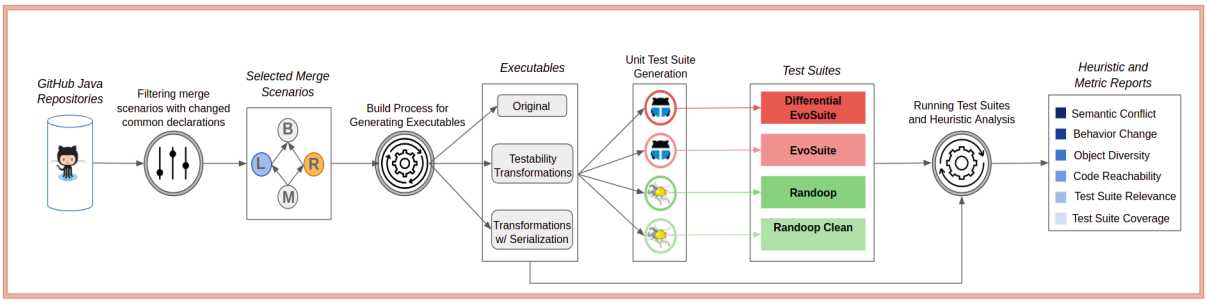
4.2.2 Empirical Evaluation

Our methodology comprises five main steps to assess the potential of unit test generation to reveal interference (Figure 30). First, we extract and select merge scenarios from Java projects hosted on GitHub, including a number of scenarios that appear in previous integration conflict studies (SOUSA; DILLIG; LAHIRI, 2018; CAVALCANTI et al., 2019; FILHO, 2017). Second, we create binary files of the programs for each selected scenario. Initially, we generate binary files using the original source code for the four software versions corresponding to the base, left, right, and merge commits (see Figure 31). Next, we generate additional four binary files (one for each commit version) but this time applying Testability Transformations we conceived (explained in Section 4.1.1.3). For some cases of our sample,[10] we generate a third binary file version with Testability Transformations and Serialization. This new version offers serialized objects as input for the unit test tools; we feed the tools with such objects to increase the chances of creating more relevant tests. Third, we apply four test generation tools to create tests for the parent commits of a merge scenario based on the binary file available (left and right from original, transformed, and serialized binary files). Next, we run our scripts to execute the tests and discard invalid tests avoiding flakiness issues.[11] Fourth, as a last automated step, we run our scripts to check the test-based interference criteria and additional related metrics regarding the quality of the generated test suites. Fifth, we manually analyze each merge scenario and the generated results to ensure that the reported conflicts are true ones. Furthermore, we investigate the reasons behind the generated tests cannot detect interference in some of the scenarios that suffer from interference.

[10] Since we rely on the quality of original project test suites to generate these objects, we could generate binary files with serialization for a few scenarios of our sample.

[11] We consider tests invalid if they present different results on different executions.

Figure 30 – Test conflicts: empirical study setup. We start with selecting Java projects on GitHub and then filtering merge scenarios with changes on mutual class elements. Next, we compose our sample by generating three different binary file versions, followed by calls to the unit test generation tools generating test suites. Finally, we execute the generated test suites to detect test conflicts and assess further metrics. Besides that, we also perform a manual analysis to verify false positives and negatives in our sample, but that is not illustrated in the figure.



Source: The author (2022).

4.2.2.1 Mining and Selecting Merge Scenarios

Our merge scenario sample consists of 85 mutually changed elements from 51 merge scenarios. Since we analyze class elements mutually changed by both parents during a merge scenario, it is expected that one single merge scenario holds more than one case of changed elements. As a result, for some merge scenarios, multiple cases of changed elements are evaluated. To establish our sample, we follow two steps, as detailed below.

As previously introduced, this study has a sample of 85 cases, of which 40 of them are originally from our previous study (SILVA et al., 2020). This way, our sample consists of three parts. The first part contains merge scenarios mined from a number of Java projects hosted on GitHub. We opt for Java projects only because the unit test generation tools are language-dependent, and some of our scripts are test tool-dependent; the tools we use in our study primarily generate test cases for Java. Most related studies also focus on Java projects. We also limit our study to GitHub projects as it is one of the most popular sources of open-source projects, and most related studies also use GitHub.

For this first part (illustrated in the third row of Table 5), we start with a list of projects from a previous study (BELLER; GOUSIOS; ZAIDMAN, 2017c; MUNAIAH et al., 2017) that focuses on Maven projects, which could help in the build creation process of our second step (see Figure 30). We then arbitrarily select a subsample of projects, available in our online Appendix (Appendix 2, 2020). Then, we use our Mining Framework to select merge scenarios that integrate changes in the same method body, constructor, or field initialization (class element). We believe this would increase the chances of collecting interference situations and make easier our manual analysis. As merge scenarios may have multiple independent changes to individual class elements, we consider each changed class

Table 5 – Distribution of changed mutual class elements

Original Sample	**Selected Changed Mutual Elements**	
	With Interference	**Without Interference**
Da Silva *et al.* (2020)	4	2
De Sousa *et al.* (2018)	3	18
Cavalcanti *et al.* (2019)	2	6
Barros Filho (2017)	12	16
Current study	7	15
Total	28	57

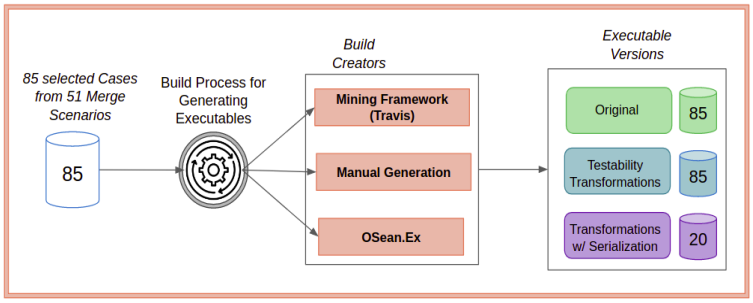
Source: The author (2022).

element a new case in our sample. In this manner, when we mention an interference case, we do not refer to a specific merge scenario. We consider only recent project histories during the mining step to reduce build creation effort in the next step, as older commits might require dependencies or resources no longer available.[12] As a result, we select six cases from five merge scenarios, four with and two without interference.

The second part of our merge scenario sample (illustrated in rows 4-6 of Table 5) contains Java merge scenarios from related studies (SOUSA; DILLIG; LAHIRI, 2018; CAVALCANTI et al., 2019; FILHO, 2017). From the first study, we select 21 cases (16 merge scenarios), three with and 18 without interference. From the second study, we select eight cases (eight merge scenarios) of independent changes to the same declaration, two with and six without interference. We also try to add more cases without interference, but we could not build them due to missing old dependencies. In a few scenarios, we do not observe changes to the same declaration, so we discard them. Finally, from the third study, we include 28 cases (22 merge scenarios), 12 with and 16 without interference.

The third part of our merge scenarios (illustrated in the seventh row of Table 5) contains cases that we originally mine from merge scenarios from previous work (SOUSA; DILLIG; LAHIRI, 2018; CAVALCANTI et al., 2019; FILHO, 2017). These studies are the same we mention when presenting the second part of our sample in the last paragraph. Although we consider some scenarios of their original studies, we observe that multiple class elements were changed at the same merge scenario. So additional cases might be mined from the related merge scenarios. This way, we use our Mining Framework (Appendix 2, 2020) to select new cases. As a result, 22 new cases were added to our sample (seven merge scenarios), seven with and 15 without interference. Table 5 summarizes all selected cases discussed here. Thus, we had a sample of 85 interference cases from 51 merge scenarios.

Figure 31 – The generation process of binary files for merge scenario commits. For each merge scenario of our sample, we generate binary files, which are given as inputs for the unit test tools. Initially, we generate binary files for the 85 cases of our sample using the original code and applying Testability Transformations. Next, for a subsample of 20 cases, we generate binary files with Testability Transformations and Serialization.



Source: The author (2022).

4.2.2.2 Building the Projects

As previously mentioned at the beginning of this section, for each interference case of our sample, selected in the previous step, we must generate binary files that are used to generate test suites later. Considering we need to execute build files with all dependencies defined for a project, we initially create these binary files versions on Travis (see Figure 31). So our Mining Framework requests Travis to create the binary file version associated with the original source code and the binary file version with the Testability Transformations. As a result, four binary files were generated for each version (original and transformed). The main advantage of this approach is to reduce the chances of broken build processes due to environmental issues. As we use the Travis infrastructure, in case of merge scenarios requiring different environment options, we would not need to deal with each one directly. Instead, we need to set up a configuration file and reuse it when applicable. When Travis fails to create the builds for lack of dependencies, support for older Java versions, or additional analysis adopted by projects, like style checking, we try to manually fix the problem on Travis updating the configuration files; otherwise, we locally create the builds. We decided to further work on these scenarios for a list of reasons. First, these scenarios satisfy our merge scenario criteria regarding parent commits changing the same class element. Second, the fixes required for the reported issues were simple, though we manually applied them, and they did not change the original behavior of the target code. Third, related studies previously evaluated these cases, so they might be relevant cases to include in our sample.

We also adopt the manual build creation process for Ant projects (one single case), as our infrastructure supports only Maven and Gradle projects. We opt to generate the binary file versions with serialized objects locally as our serializer was implemented to

[12] When we mined these scenarios, we adopt a limit date for merge scenarios not older than five years.

work that way, and it is a tool still under development. At this point, if we fail to create one of the eight builds for a scenario, we simply discard the scenario— in our experiment, we discarded five scenarios. The process and infrastructure we use to create the builds appear in our online Appendix (2020).

At the end of this step, we were left with a sample composed of 51 *merge scenarios* and 85 *potential interference cases*, knowing that some merge scenarios contain more than one independent change on the same declaration. For all 85 cases, binary files with the original source code and Testability Transformations are generated. Finally, for 20 out of these 85 cases, an additional binary file version with serialized objects is also generated.

Regarding the adoption of Testability Transformations and Serialization during the generation of the binary files, we follow some steps, as discussed below.

Testability Transformations

As previously discussed in Section 4.1.2, Testability Transformations are adopted to increase the testability of the code under analysis by unit test tools. This way, for each commit of a merge scenario of our sample, we apply these transformations when generating the binary files. For that, we call the associated tool informing the path of the class, where the transformation should be applied. In case the binary files are generated by our Mining Framework, this process is automated as internal steps of the framework. Otherwise, we manually call the tool to apply the transformations. If the supposed target class to receive the transformations extends another one (superclass), applying this transformation only on the target class might result in invalid code, considering the superclass should also receive the transformations. For these cases, we manually apply the transformations considering the elements that would not require changes on the parent classes.

Serialization

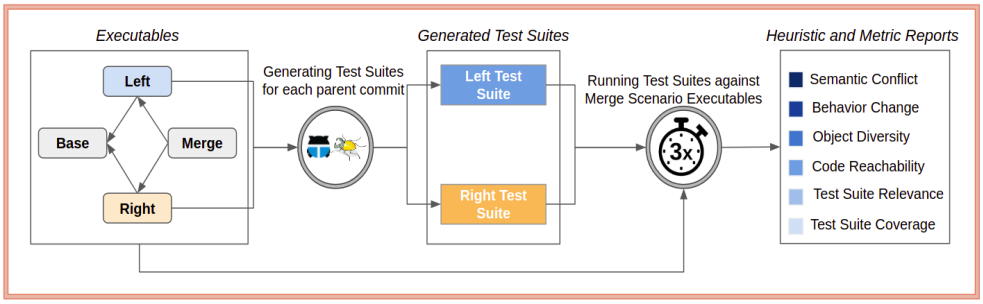
Since unit test tools face some problems when generating required object types used by test suites, Serialization might be used to provide relevant objects, which can be directly used by the test tools (see Section 4.1.3). This way, to generate these serialized objects in our study, we use our tool OSean.Ex. Initially, we wanted to provide serialized objects for all cases of our sample; however, we could not do it due to weak project test suites. For example, objects will be serialized if at least a test case exercises a target method given as input. In our study, we inform the method that potentially holds the semantic conflict as the target method for the serializer. Additionally, we opt for running the test suites for 60 seconds as most project test suites are finished by this time. Once OSean.Ex accepts a list of commits, we invoke this tool giving as input the four commits of a merge scenario. This way, the serialized objects are generated based on the first commit of the previous list; in our study, the merge commit. For the remaining commits of that list, they reuse the

serialized objects by deserializing them based on their versions. So for base and parent commits, OSean.Ex only performs the last step generating the binary files.

Since this step is expected to provide useful objects generated from the original project test suites, if a project has a strong test suite, we might expect the target code to be called multiple times during the test execution, resulting in multiple generated serialized objects. For the 20 cases of our sample supported by serialization, we observe many generated objects in most cases. For example, for project Okhttp, 357 serialized objects were generated based on calls to the method `copyWithDefaults` of the class `OkHttpClient`. In this case, the test suite called the target method multiple times and with different arguments. Cases like this are beneficial to our approach as the tools might use and combine different objects and possibly detect behavior changes and conflicts. However, for some cases, like in the project Cucumber, we observe that only eight objects were serialized, revealing the weakness of the project test suite. As a result, the tools are expected to have limited chances to detect the conflict considering the low number of objects.

4.2.2.3 Generating and Executing Tests

Figure 32 – Generation and execution of test suites. For each case of our sample, we generate test suites based on both merge scenario parent's commits. Next, we execute three times each generated test suites against all merge scenario commits in order to calculate our metrics.



Source: The author (2022).

Each merge scenario resulting from the previous step has proper binary files that tests can execute and exercise. These binary file are required by unit test generation tools that generate tests and run them against the system to be tested, discarding tests that fail or do not increase code coverage (see Figure 32). This observation is valid here for the test generation tools we evaluate: Differential EvoSuite, EvoSuite (ALMASI et al., 2017), and Randoop (PACHECO et al., 2007). We also chose these tools for their robustness and popularity. Additionally, we evaluate Randoop Clean, our modified version of Randoop.

In this step, we readily apply the unit test generation tools to create tests for four of the versions (left, right, and their transformed versions, as explained above) and for a subsample of the other two versions, when applied (serialized left and right) associated with

each merge scenario. For each version, our scripts call EvoSuite, Randoop, and Randoop Clean with the parent commit binary file as input. For Differential EvoSuite, which tries to generate tests that reveal behavior differences between two program versions, we also give the binary files of the base commit as input, used as the regression version. So, the tool will try to generate a test that passes in the parent commit and fails in the base commit.

For each tool, we use their default configuration and adopt a budget of 5 minutes.[13] We decide to adopt 5 minutes considering related work opt for different budget configurations (1, 2, or even 10 minutes). Additionally, in our previous study (SILVA et al., 2020), we opt for 2 minutes of budget. So this time, we want to give more time for the tools and verify if the budget affects the detection of conflicts. Our scripts call each tool once, obtaining two test suites, one for each version in a merge scenario associated with each case of our sample. So for each case, 16 test suites are generated for the original and transformed binary files; additionally, when applied, other eight suites for the serialized binary files.

Regarding the number of generated tests by each parent commit, the tools might generate different numbers. For example, Differential EvoSuite might generate at most one test case, while the other tools EvoSuite, Randoop, and Randoop Clean might generate multiple test cases. This way, we may not ensure the number of tests as it relies on the target code given as input for tools, like complexity, direct dependencies, and other associated factors.

For each resulting test suite, our scripts execute the contained test cases three times for each of the different versions: base, both parents, and merge (see Figure 32), resulting in 12 executions. We must execute the tests in both parents, as our last two conflict criteria must assess the test results against all binary files associated with the merge scenario commits. Finally, for each merge scenario, the 16 generated test suites are executed 12 times resulting in 192 executions; in the case of a scenario with serialized binary files, the other eight test suites are also executed 12 times resulting in additional 96 executions.

These three mentioned executions are repeated, aiming to detect test flakiness. If a test case does not yield the same result (pass or break) in the three repeated runs, we filter it out, as not doing that could compromise the accuracy of our interference detection criteria. The test suite execution results associated with each case of our sample are grouped into three sets: tests with failed status, tests with passed status, and tests that could not be executed because they do not even compile with the version under test. Such validity issues with tests might occur because the test was generated for a given revision, say left, but is executed in other revisions as well: base, right, and merge. If the left revision, for example, adds a method declaration that is called in the generated test, this test will not even compile with the base and right version. In the same way, tests with error statuses were not considered as failed tests. An errored status means an unexpected situation during the test execution, which does not involve the program behavior under test. Such invalid

[13] The versions of the tools used by SAM are informed in our online appendix.

tests are discarded as the last action in this step.

Someone may argue that we might consider using original project tests to detect test conflicts. In previous runs of our study, we explored this option by running the project test suites on merge scenario commits of four projects of our sample; however, no conflict was detected. Further investigating these cases, we observed that these projects present good test suites, but they did not cover the mutually changed element by the parent commits. Consequently, they do not execute the target code during their execution. Thus, we decided to generate new test suites using unit test tools aiming to execute these elements during test execution.

4.2.2.4 Detecting Interference

We group test suite executions into sets for each case of our sample based on the binary file versions used to generate the tests. Each set contains the executions associated with the base, parents (left and right), and merge commits for the original, transformed, and serialized versions. Next, for each execution result set, our scripts compute the test cases that satisfy one of our conflict criteria. For example, for our first two criteria (see Figure 18), we look for tests that present the same result when executed on the base and merge commits but different results on the parent commit (left or right). For our last two criteria, we look for tests that present the same result when executed on the base and parent commits (left and right) but different results on the merge commit. Finally, our scripts collect the results for further analysis and report interference if our criteria are satisfied.

4.2.2.5 Evaluating Improvements for Semantic Conflict Detection

Besides evaluating the detection of semantic conflicts, we decide to evaluate how far the tools are to detect semantic conflicts based on the different approaches and techniques we present here. For that, we consider metrics that may help us evaluate the impact of each approach (conflict detection criteria, testability transformations and serialization), and technique (different unit test tools). Next, we present the metrics that we consider for this work.

General Behavior Change Detection

Besides detecting semantic conflicts, we also evaluate whether the tests generated by the tools could detect general behavior changes. Previous work considers this metric to assess the potential of unit test tools to detect behavior changes. A behavior change represents an unexpected behavior between two versions of a program (commits) detected by a test case. For instance, consider a test case that passes and fails against the program versions *v1* and *v2*, respectively. The different outcome observed by the test case indicates

that some behavior was previously held by version *v1* but no longer by version *v2*. In our context, if a behavior change is detected due to the conflicting changes introduced by one of the parent commits, we may conclude the tools were closer to detecting semantic conflicts.

To detect those behavior changes, we use the test suites previously generated and look for test cases that report different outcomes when running them against two commits. As we want to detect the behavior changes introduced by each parent in a merge scenario, we look for detection affecting the parent and base or merge commits. So for each case of our sample, there are at most four possible behavior changes. For example, a test case might reveal a behavior change involving the parent and base commits due to the changes applied by the parent. Additionally, the same test may also reveal another behavior change involving the parent and merge commits, but this time caused by the changes applied by the other parent.

Test Suite Relevance

Considering that a parent commit might introduce multiple changes to different parts of the code, there is no guarantee that unit test tools will detect all these behavior changes due to their randomness. However, we believe that the higher the number of generated tests detecting behavior changes, the higher the chances these detections are associated with different changes. This way, our scripts aim to assess the number of test cases generated by each tool that detects a behavior change.

For our previous metric, our scripts check whether at least a test detected a behavior change between two commits. For this metric, our scripts check the number of different tests that detected a behavior change.

Object Diversity and Target Code Reachability

Based on the changes performed in our tool Randoop Clean, we aim to evaluate whether these changes are closer to detecting semantic conflicts compared to Randoop. For that, we compute two metrics that emphasize the improvements implemented on Randoop Clean.

As we advocate in Section 4.1.4, Randoop Clean aims to maximize the number of calls to the target method holding the semantic conflict and the number of object diversity by maximizing the number of calls to create objects. Based on these properties, we compute two metrics: the number of calls performed to a target method, and the number of different handled objects by a test suite. We believe the higher the calls to the target method and the diversity of handled objects, the higher the chances to detect a conflict or a behavior change.

To compute those metrics, we implement a report that for each generated test suite (i) counts the number of calls made for all possible methods of the target classes given as input to the tool, and (ii) the number of different objects handled by the test suite.

Once a test suite is generated, our report analyzes the test suite and generates the results on a CSV file. This report was previously implemented for our tool Randoop Clean, and then we extend Randoop with it. So when we refer to Randoop in our study is actually a version of Randoop extended with this metric collection functionality.

Source Code Coverage

Aiming to evaluate the improvements of Randoop Clean when compared to Randoop, we decide to compute the source code coverage achieved by the generated test suites of each tool. Based on such a metric, we may inform which tool is closer to detecting semantic conflicts. For that, we decided to compare only the coverage achieved by each tool against the merge commit. Since the merge commits own the conflicting changes, if these conflicting changes are covered by a test suite on the merge commit, it is also expected that these changes are also covered on the parents and base commits, when applicable. As Randoop does not provide source code coverage information, we implement additional scripts to compute this metric using the tool Jacoco (JACOCO, 2022).

For each generated test suite, our scripts call Jacoco to instrument the binary file previously used to generate the mentioned test suite. This instrumentation is required as Jacoco adds instructions used to inform whether a specific code instruction was covered or not during the execution of the test suite. Next, the test suite is executed against that new instrumented binary file resulting in a file with the coverage results. Jacoco provides different report formats, but our scripts adopt CSV files as the default format. With these reports, we have access to the coverage of the project, its classes, and methods. Since we want to explore the coverage of a target method, we look for the instruction coverage of a target method in each test suite. Thus, we compare the percentage achieved by each suite, and consequently, the more efficient tool.

4.2.2.6 Analyzing the Scenarios and Establishing the Ground Truth

With the results reported by our scripts, we manually analyzed each case of our sample to establish a ground truth of actual interferences to contrast it with the obtained results. In particular, we collected information on false positives (our interference criteria are satisfied, but there is actually no interference in the scenario) and false negatives (our interference criteria do not hold, but the scenario actually suffers from interference). This analysis also helps to understand the potential of unit test generation and our criteria to detect interference.

Six researchers manually analyze all cases of our sample; in pairs, the researchers individually analyze each case to check for interference and later compare the analysis with his partner. If both researchers agree with the same decision, they present the scenario and its evaluation to the remaining four researchers. However, in case of a conflicting decision, the whole group discuss the case and reach a verdict. To reduce the chances of human

error and misjudgment in this process, for each interference verdict, we manually design a test that could reveal the problem, especially when the tools are not able to find one. Similarly, each non-interference verdict has an explanation of why we could not design such a test case; for example, one of the changes is a structural refactoring, not affecting the behavior of the other integrated changes. Instructions and guidelines used during this process are organized as a document, which is available in our online Appendix (Appendix 2, 2020).

For many cases (57), the ground truth is available in previous work, but we nevertheless follow the process above and compare verdicts. For all cases, we summarize the integrated changes to help reach verdicts and using our dataset for replications and further studies. For the merge scenarios reported with interference, we analyze the associated test suites to ensure that the tests explore the conflict that we find during our manual analysis. This analysis is essential since the testability transformations could introduce false positives to our results, as some semantically change the program behavior. For that, we check whether the failed test case assertions explore the side effects of the elements involved in each conflict.

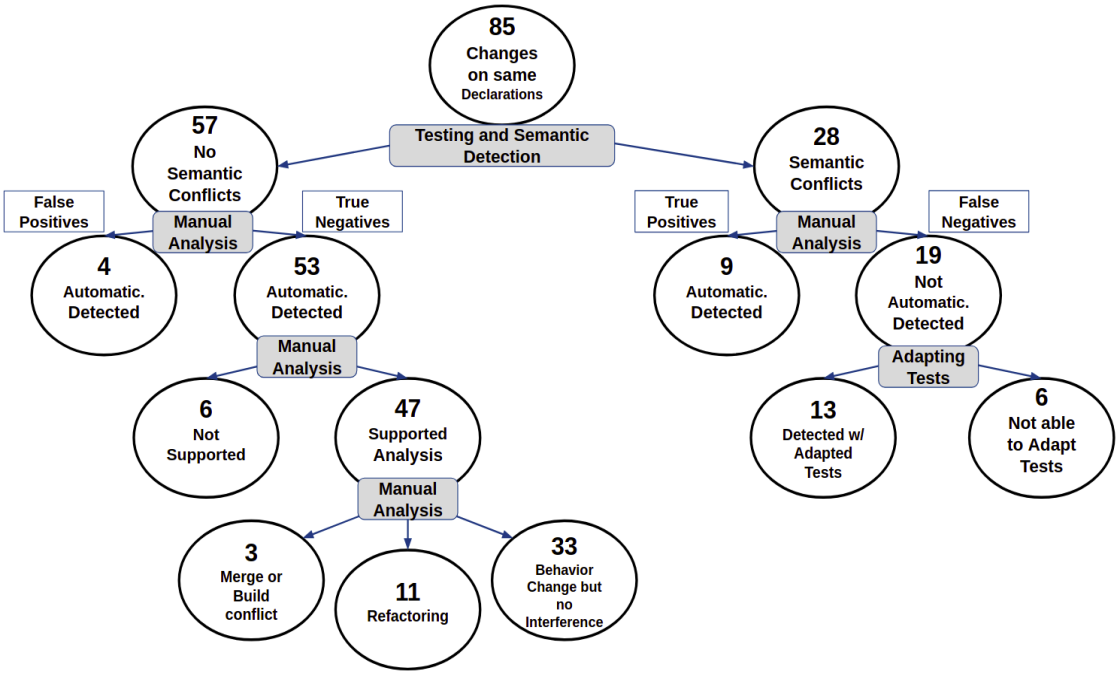
Aiming to understand the limitations that unit test generation tools face—in our context of exploiting the generated test cases to detect interference—we analyze the test suites of the identified false negatives. Based on the test descriptions we wrote during our ground truth analysis, we try to change the unsuccessful generated test cases and check if they could then detect interference. As a result, we identify improvements that could be applied to the tools, as well as to better understand and help assess how close the tools are to generating a test case that would reveal interference.

At the end of this step, we obtain a dataset composed of merge scenarios associated with their build files (original, transformed, and serialized binary files), generated test suites, interference ground truth, and further information on the quality of each test generation tool.

4.3 RESULTS

We now present the results of our analysis of the 85 parallel changes to the same class element declarations in the 51 merge scenarios mined from 31 GitHub Java projects (see Section 4.2.2.1), including how semantic conflicts were detected through our criteria and automatically generated tests by our tool SAM. We also discuss how the test generation tools could be improved to increase conflict detection accuracy. Our focus is first on the cases with conflicts. Later we discuss the cases with no conflicts, as defined by our ground truth, concluding with suggestions for improvement.

Figure 33 – Results based on merge scenarios conflict detection and occurrence using all four unit test tools with different binary file versions. Distribution of changes (on same class element declaration), their classification, and whether a conflict is detected or not. Note: *Refactor.* abbreviates *Refactoring*, *No Interfer.* abbreviates *No Interference*.



Source: The author (2022).

4.3.1 Cases with conflicts

Recall that, to verify the potential of SAM for detecting semantic conflicts, we select 85 cases of changes on the same declarations in the source code from 51 merge scenarios from 31 GitHub Java projects, apply testability transformations to increase their testability, call the test-generation tools to generate test suites, and finally, execute, filter, and remove invalid test cases from our analysis. Figure 33 illustrates our results; the right-hand branch gathers the changes with semantic conflicts, while the left branch represents the changes without conflicts, according to our ground truth. In total, our tool could automatically detect nine conflicts (in five merge scenarios) from 28 conflicts (32%). This number of detected conflicts reinforces our previous findings that unit test tools are useful to detect semantic conflicts (SILVA et al., 2020). Although the tools left many false negatives, the tools do not report many false positives. So, a developer in her working-day life would prefer a tool with some false positives and many negatives instead of a tool with many false positives, considering she might stop her work and investigate a conflict that does not occur in practice.

Analyzing the right branch in Figure 33, we observe that 28 changes have semantic conflicts.[14] Figure 35 shows a change extracted from a merge scenario of the project

[14] Note that, even when the scenarios stemmed from another study, we double-checked them since other

HikariCP with a semantic conflict detected by our tool with a test generated by EvoSuite.[15] In this merge scenario, the left commit added a new condition in the if statement using the local variable `retries` (Line 6 in Figure 35), restricting the call to `incrementAndGet()`, which increments the number of total connections to a pool. In parallel, the right parent commit changed the value assigned to the local variable `retries` (Line 3 in Figure 35), which is referred to the changes performed by left. So, left and right parent commits individually change the program behavior (creating and adding single connections to a pool) based on their needs. As expected, these changes can be integrated without reporting merge conflicts since lines 4 and 5 separate the changes in Figure 35.

This semantic conflict would not be detected unless a test case is available to verify this method's behavior. In Figure 34, we show the test case generated by EvoSuite using the right commit parent. As highlighted in the test case (Line 6 in Figure 34), the expected number of `totalConnections` is 10. This assertion is defined based on the execution of analyzed code that returned this value during the generation process of EvoSuite. This way, running this test case on the R commit, the test passes, showing that the same behavior observed by EvoSuite during the generation process is held by the right commit. For the base commit, the test fails as the received number of connections is 12; it happens because some calls to `decrementAndGet()` (Line 14 in Figure 35) are not performed, which decrements the total number of connections to a pool. These calls are not made as the old if condition is no longer `true` (Line 13 in Figure 35). However, these calls are made during the right version as this commit changes the local variable `retries` used in the if condition. In the same way, for the merge commit, the test case also fails as the received number of connections is -30; this time, no call to `incrementAndGet()` is done (Line 6 in Figure 35) due to the changes performed by the left commit, leading the first condition of the if statement to be `false`. On the other hand, multiple calls are made to `decrementAndGet()` (line 14 in Figure 35), decrementing the total number of connections caused by the right changes.

Figure 34 – Test case generated by EvoSuite detecting test conflict.

```
1 @Test
  public void test1(){
3     {...}
      HikariPool hikariPool0 = new HikariPool(hikariConfig0);
5     {...}
      assertEquals(10, hikariPool0.getTotalConnections());
7 }
```

Source: The author (2022).

The other eight detected conflicts have some common aspects with the example we discussed. First, the conflicts occur because the parent commits change the values assigned

works might have a slightly different understanding of semantic conflicts.

[15] HikariCP - merge commit: 1bca94a

Figure 35 – Test conflict caused by changes that update same variable.

```
1  private void addConnection(){
-     int retries = 0;
3  +     int retries = configuration.getAcquireRetries();
       {...}
5      try{
          if (retries == 0 && totalConnections.incrementAndGet() >
              configuration.getMaximumPoolSize()){
7             totalConnections.decrementAndGet();
       break;
9         }
          {...}
11     } catch (Exception e){
          {...}
13        if (retries++ > configuration.getAcquireRetries()){
             totalConnections.decrementAndGet();
15        }
       }
17 }
```

Source: The author (2022).

to the same objects. So, to detect the conflict, the tools should focus on one specific object. Second, the tools could directly access the object involved in the conflict (side effects). Thus, the test cases should have at least one assertion exploring this object's fields.

Besides understanding the potential of SAM for detecting conflicts, it is also important to assess how each test generation tool used by SAM actually contributes to SAM's potential. As a result, we observe that, in this study, EvoSuite and Differential EvoSuite are the most successful tools, detecting six conflicts each, while Randoop and Randoop Clean detect two conflicts each (see Table 6, second column). None of the tools captured all detected conflicts, though they generated test suites for all scenarios. As we also observe, Differential EvoSuite is the only tool not reporting false positives (presented in square brackets). In contrast, EvoSuite presents three false positives, and Randoop and Randoop Clean present each one case of false positives. Although EvoSuite and Differential EvoSuite detect the same number of conflicts, when we combine their results, together they detect all nine conflicts previously mentioned. All detected conflicts detected by Randoop and Randoop Clean are covered by the other tools. We observe a few cases where Randoop and Randoop Clean were close to detecting the conflict, but they inappropriately explored the object holding the propagated interference. For example, while exploring the object values, the related assertions only verified whether the object was null or not. This way, if we want to derive a tool for catching all detected conflicts of our sample (32%), combining Differential EvoSuite and EvoSuite would be enough. A version of SAM that uses only these two tools would be computationally more efficient and detect the same conflicts.

For these conflicts, applying code transformations is a good starting point to conflict detection, as these transformations increase the testability of the source code under analysis (second and third columns in Table 6). For example, after applying testability transformations, the tools could directly access the objects and, consequently, explore them in their

Table 6 – Distribution of detected conflicts by unit test tools and binary files. Down arrows stand for conflicts detected by the current cell, but not detected by the next cell below. Up arrows stand for conflicts detected by the current cell, but not detected by the next cell above. Left arrows stand for conflicts detected by the current cell, but not detected by the next left cell. Right arrows stand for conflicts detected by the current cell, but not detected by the next right cell. Numbers between brackets represent false positives reported by the tools. *Pr.*, *Re.*, and *Ac.* stands for *precision*, *recall*, and *accuracy*, respectively. The values of *recall*, *precision* and *accuracy* are calculated based on the 28 conflicts with ground-truth present in our sample.

Unit Test Tools	**Binary File Versions**		
	Transformed	**Original**	**Serialized**
Differential EvoSuite	6 (↓3, →3) Pr. 1 Re. 0.21 Ac. 0.74	3 (→3) Pr. 1 Re. 0.1 Ac. 0.70	0
EvoSuite	5 (↑2, ↓4) [3] Pr. 0.62 Re. 0.17 Ac. 0.69	6 (↑3, ↓5, ←1, →6) [1] Pr. 0.85 Re. 0.21 Ac. 0.72	1 (↑1, ↓1) Pr. 1.0 Re. 0.03 Ac. 0.68
Randoop	2 (↑1, →1) [1] Pr. 0.66 Re. 0.07 Ac. 0.68	1 (→1) Pr. 1 Re. 0.03 Ac. 0.68	0 [1]
Randoop Clean	2 (→1) [1] Pr. 0.66 Re. 0.07 Ac. 0.68	1 (→1) Pr. 1 Re. 0.03 Ac. 0.68	0 [1]

Source: The author (2022).

assertions. So the use of transformations led to the detection of three additional conflicts not detected by the same tools applied to the same cases without the transformations. This observation reinforces our previous results that testability transformations help test tools to detect more conflicts in half of the captured conflicts (SILVA et al., 2020). One conflict is not detected by EvoSuite after applying the transformations. In this particular case, the target method is public so that the transformations would have no direct effect regarding calling it from the test suites. We believe this non-detection occurs due to the randomness of EvoSuite when combining different test suites in order to meet the target fitness function. Furthermore, the conflict is also detected when EvoSuite received as input the binary file with serialization and testability transformations.

Regarding the use of Serialization (fourth column in Table 6), eight out of the 20 cases

supported by this binary file version are associated with conflicts. Although all these cases were supported with serialized objects, no conflict is detected using them. The test case generated by EvoSuite, which detects a conflict (fourth column, fourth row in Table 6), does not explore serialized objects. However, in other generated test cases for this case, the tests explored the available serialized objects. This way, for this subsample, we conclude that serialization does not support tools to detect conflicts. For a better evaluation, we should run new studies with a bigger sample in order to understand the weaknesses of this approach.

Comparing our results with previous work shows that our initial results are replicable (SILVA et al., 2020); all four conflicts previously detected are also reported in our current study. Regarding these four conflicts, Differential EvoSuite is the most successful tool for detecting all four conflicts. In contrast, EvoSuite detects two of them, and Randoop and Randoop Clean detect each one conflict. However, regarding all conflicts detected in this study (nine out of 28), we do not observe one single tool detecting all conflicts but rather a combination of two tools. We must mention that we adopt the same Testability Transformations in those scenarios but a larger budget (5 minutes for the generation process). So we may conclude that, at least for these four conflicts, the adopted budget may not impact the detection of conflicts. For a better conclusion, we should run the new five detected conflicts by this current study adopting a smaller budget (2 minutes) and assessing whether the tools detect all conflicts.

Like our previous study, all nine conflicts detected in this study are detected through the same conflict criterion (see Figure 18); a test case that passes on the parent commit and fails on the base and merge commits. Although no conflict was detected using our new two criteria, they are still valid as they explore situations not supported by our previous criteria (see Section 2.2.3). We believe this no detection involving our last two criteria happens because of our approach of generating the test suites. As we generate tests based on parent commits, these tests are expected to pass on these commits, partially limiting our last two conflict criteria. To better evaluate our new two criteria, we should run a new study generating tests against the base and merge commits instead of only parent ones.

Someone may argue that the budget adopted for the tools to generate the test suites negatively impacted the potential of generating more relevant tests. This way, giving more time to the tools, especially EvoSuite, might detect more conflicts. However, previous studies adopt a similar budget when using the unit test tools under analysis. In order to evaluate the impact of higher budgets, considering the complexity of the projects of our sample, we could run a new study adopting different combinations of budgets.

As a final remark, based on the different approaches and techniques we adopt here to detect conflicts, we conclude that a semantic merge tool, like SAM, based on Differential EvoSuite and EvoSuite is the best combination of tools to detect conflicts. Combined with these tools, we recommend the adoption of binary file version for merge scenario commits

with Testability Transformations. Finally, regarding our proposed conflict criteria, the first criterion is the best option for detecting all conflicts reported in this study.

4.3.1.1 False Negatives

Moving to the remaining 19 cases representing false negatives (see Figure 33), we focus on understanding the limitations behind the missed detection. To this end, we analyze the generated test suites and verify if the conflicts could be detected after applying a few changes to the test cases. Our main goal is to evaluate how close the generated tests were to detecting conflicts. To guide us during this adaptation process, we consider the test descriptions that we previously created during our initial manual analysis (see Section 4.2.2.6). The changed test cases could detect the conflict for 13 out of 19 changes on the same declarations.

To illustrate it better, we must look at the changes presented in Figure 36. Both parents' commits add calls to methods `write()` to the same object stored in the field `logger` by writing different values on the log output.[16] While Left updated the message added through the method `info` (Line 4 in Figure 36), Right performs its update using the method `debug` (Line 9 in Figure 36). Although the parents use different writing methods on the object, they are writing on the same character stream. In these circumstances, the original test case assertions are expected to explore the object's contents changed by the target method. However, only one assertion checked it, verifying whether the object was null. Regardless of the particularities of this scenario, an appropriated assertion should compare the value assigned to the object logger during the test execution. This case shows that even for false negatives of unit test generation tools, the generated test suites might reach the location of the infection but failed to explore the propagated interference. As such, the tools were close to detecting interference in this case.

Figure 36 – False-negative test conflict caused by changes on the same variable.

```
1  public void logAutoConfigurationReport(boolean isCrashReport) {
       {...}
3      if (...) {
           this.logger.info("\n\nError starting ApplicationContext. "
5          + "To display the auto-configuration report enabled "
           + "debug logging (start with --debug)\n\n");
7      }
       if (...) {
9          this.logger.debug(getLogMessage(this.report));
       }
11 }
```

Source: The author (2022).

In other cases, the tools could not detect the conflict as the interference is not propagated; so the tools might not access the infection state (see Figure 37).[17] Like our previous

[16] Spring Boot - merge commit: fdd3f12
[17] HikariCP - merge commit: 1bca94a

example, in this case, the parents also write on the same object `logger` (line 6); while Left adds a new message in the target object through the method `info` (line 15), Right removes two messages writing through the methods `info` and `warn` (lines 9 and 11, respectively). So the parents update the contents of the same object, though this object is a local variable. Even if the infection location were reached, the tools would hardly explore the contents of the local variable. In other cases, the tools were not even close. The methods holding the conflict were called, but with arguments preventing the test from reaching the interference location; for example, `null` argument values lead to exceptions before reaching the interference state. Consequently, irrelevant assertions were generated, which could not detect the interference even if it was propagated.

Figure 37 – False-negative test conflict caused by changes on local variable.

```
1  public class HikariConfig {
2
3    {...}
4
5    public void validate(){
6       Logger logger = LoggerFactory.getLogger(getClass());
7       {...}
8       if (acquireRetryDelay < 0){
9  -       logger.error("acquireRetryDelay cannot be negative.");
10      } else if (acquireRetryDelay < 100) {
11 -       logger.warn("acquireRetryDelay is less than 100ms ...");
12      }
13      {...}
14      if (jdbcUrl != null){
15 +       logger.info("Really, a JDBC URL? It's time to party like it's 1999!");
16      }
17      {...}
18   }
19
20 }
```

Source: The author (2022).

However, in some cases, slightly adapting the assertion would be enough to detect the interference. For example, in one conflict of project CloudSlang,[18] the parent commits change the same `HashSet` by adding new different elements. So to detect the interference, it is expected that the test case explores the elements in that `HashSet` or its size. However, as presented in Figure 38, the assertion only checks whether the returning `HashSet` object is `null` (line 6). Changing that assertion and exploring its size would be enough to detect the related conflict (line 7 in Figure 38). Later in this section, we discuss improvements for unit test generation tools.

Dependencies on external resources also lead to the non-detection of conflicts. For example, in one scenario, Left updates an `if` condition, while Right adds another `if` statement and its related code inside an old `for` statement. Since these changes are placed into a `for`, which is only executed if a valid session to a database (external resource) is

[18] CloudSlang - merge commit: 20bac30

available, the new changes would hardly be reached during execution. It occurs because the tests do not handle external dependencies like the case just discussed here. In order to address this limitation, the tools should set up the required external dependencies before executing the test cases. For example, adding a method on the test class responsible for that task, using annotations like `Before` or `BeforeEach` provided by Junit. Alternatively, the tools might also use *mocks*, overcoming that limitation without further directly dealing with additional external dependencies.

Figure 38 – Slightly adaptation of generated test case.

```
@Test
public void test004() throws Throwable {
  {...}
  SlangImpl slangImpl2 = new SlangImpl();
  Set<String> strSet4 = slangImpl2.getAllEventTypes();
  assertNotNull(strSet4);
  assertEquals(22, strSet4.size())
}
```

Source: The author (2022).

4.3.2 Cases Without Conflicts

Focusing now on the left branch in Figure 33, we discuss the 57 changes without conflicts. In this study, using the tools to detect interference might lead to false positives, as discussed below. Next, we also discuss the true negatives by classifying these scenarios based on their support for generating and executing tests.

4.3.2.1 False Positives

SAM might report false positives for a few main reasons. The first reason is regarding the occurrence of flakiness, so the generated test cases pass or fail for no specific reason. For example, consider a flaky test case that sometimes fails in the base and merge commits, while it passes on the parent commit. As the test results satisfy our first conflict criterion, SAM would report the detection of a conflict. In order to address this limitation, SAM runs each test suite three times in each commit, aiming to detect and discard possible flaky tests. However, in this study, we do not detect flaky tests. Second, our heuristics for detecting conflicts are approximations for computing interference. We use them regardless of test characteristics, but they are only valid, for example, whether assertions focus on the state elements changed by the left and right commits.

As a result, during our analysis results, we observe the occurrence of 4 cases of false positives, as illustrated in the leftmost elipse of Figure 33. They occur when the detection conflict criterion is satisfied, but the parent changes are not the cause of the failed assertions. For example, consider a scenario where the parent commits change the same target method

and other methods.[19] While Right sets the values of nine fields of an object called `builder`, Left also changes some fields of the same object but disjoint ones; additionally, Left updates the version of an external dependency. The generated test case based on the Right commit passes in that commit as expected (see Figure 39), as its changes lead to an exception during execution (`IlegalStateException`). When running the test on the Base commit, the expected exception is not thrown as the sets applied by Right are not performed on the base commit (Line 7 in Figure 39). Finally, when running on the merge commit, the test also fails, but this time due to the new external dependency version, which leads to a different unexpected exception (Line 4 in Figure 39). This way, we have a test case that satisfies one of our conflict criteria, but only part of the failed states are caused by the parent changes on the same target method. If we skip the code execution that caused the failure on the merge commit, the expected exception would be thrown, leading the test case to a successful state, and consequently, no conflict would be reported as our conflict criterion would not be satisfied.

Figure 39 – False positive test conflict caused by unrelated parent conflicting contributions.

```
@Test
public void test1108(){
   {...}
   MongoClientOptions mongoClientOptions14 = builder13.build();
   try{
      MongoClient mClient16 = mProp.createMongoClient();
   } Catch (IllegalStateException e) {...}
   assertTrue("'" + boolean3 + "' != '" + false + "'", boolean3 == false
      );
}
```

Source: The author (2022).

We also observe another situation during our investigations that might cause false positives. For example, consider a merge scenario where the parent commits change the same method and other unrelated methods. The generated test case satisfies our conflict criteria, as the failed assertions explore the disjoint methods changed by the parent commits. So the target method, where the conflict takes place, is not executed. This way, we should not compute a conflict since the test case did not reach the target method. In case, the disjoint methods changed by the parent commits represent a conflict, its detection by SAM would occur by chance. As the current version of SAM reports all test cases that fit our conflict criteria, false positives might be reported for developers. We could extend SAM to explore the line responsible for the test case failure for future versions. In case a test fails in different lines when executed on different commits, we might assume this case represents a false positive, and no conflict should be reported.

[19] Spring Boot - merge commit: 958a0a4

4.3.2.2 True Negatives

Overall, SAM does not report many false positives. Most cases without conflicts were correctly classified as true negatives. Now we discuss in detail some cases of true negatives. First, six cases were classified as not supported since the changes to be merged occur in test cases of test classes, not in the code that implements the system functionality. The unit test generation tools could not generate test cases for such classes, as the associated test framework and project environment were not provided to the tools during the generation process. Even if we could provide the requested environment, we would still classify the changes without semantic conflicts based on our manual analysis and interference criteria. The changes involve refactorings or semantic changes but are not interfering with each other. In two cases, although the changed elements were also placed in test classes, the parent commits change regular methods, so the tools might generate tests exploring them.

Moving to the 47 changes (to the same declarations in source code) supported by the unit test generation tools, we expect no test case to detect a semantic conflict. So even if the test suites exercised the parent commit changes, no conflict should be reported in the local context. For 33 changes, the parent commits individually change the program behavior, but when integrated, they do not interfere with each other, at least in the local context. It might be possible that these changes globally interfere. For example, two developers may, in the same method declaration, add assignments to different fields of the same object. So, locally, the changes do not interfere with each other. But, if we consider the surrounding program context, say a method `computeRate()` that calls the changed method and uses the two fields as part of an if condition, we could have interference and conflict. Unit tests that exercise the context classes could still detect this interference by invoking `computeRate()`, but not by focusing only on the class that declares the changed declaration, as we do in our experiment.

Next, we observe 11 changes resulting from structural refactorings, such as variable extraction and method rename. As we explain above, in such circumstances, even if one parent commit changes the program behavior, we expect no semantic conflict. This way, even if the generated tests present different outputs between the base and one parent, let us say Left, the merge commit might behave exactly like the Left commit, as the Right commit did not change the program behavior. Finally, for the remaining three changes (to the same declarations in the source code), the parent commit changes cause other kinds of conflicts during the integration (textual or build conflicts). Our goal in this study is to analyze the interference among contributions, so it is essential to ensure the class files that hold the semantic conflicts have only the changes performed by the parents' commits. When merge or build conflicts occur, human intervention would be necessary to fix these conflicts, and then it would be more difficult to isolate the parent commit changes and perform our analysis. Even with this risk, the tools generated test suites, but no conflict is detected.

Regarding the accuracy of the evaluated tools, we observe that Differential EvoSuite presents the highest value for this measure (0.74, third row, second column in Table 6); it occurs due to its precision of 1.0, though it presents a recall of 0.21. Next, we have EvoSuite with an accuracy of 0.69 (fourth row, second column in Table 6); the precision of EvoSuite (0.62), compared to Differential EvoSuite, is inferior due to the occurrence of false positives, as we previously discussed here. Finally, Randoop and Randoop Clean present the same accuracy (0.68, fifth and sixth row, second column in Table 6, respectively) but with a small recall (0.07). Although we have applied improvements on Randoop Clean, we did not change the core steps adopted by the tools when generating test suites. This way, for example, the non-detection of conflicts caused by the creation of irrelevant complex objects is still a weakness in both tools.

4.3.3 Further Evaluation of Tools and Related Test Suites

Although our main focus here is the detection of semantic conflicts, we are also interested in evaluating other metrics regarding the quality of the generated test suites. This way, in this section, we explore metrics that allow us to better understand the strengths and weaknesses of the unit test generation tools.

4.3.3.1 Behavior Change Detection

Regarding the detection of Behavior Changes, we are looking for test cases that present different outputs when executed against a pair of commits. This way, we might evaluate if the tools are close or not to detect conflicts, in case the reported behavior change is associated with the changed class element. Additionally, the reported change must also not be associated with unrelated changes performed during the parent commits.

Overall, 89 behavior changes are detected by the generated tests. These changes involve different behavior between the parent and the base or merge commits. EvoSuite is the most successful tool detecting 47 out of 89 behavior changes. Next, we have Randoop Clean, Randoop, and Differential EvoSuite with 38, 37, and 28 detections, respectively (see Table 7, second column). Even combining the last three tools, they would not achieve the rate detection of EvoSuite, as there are some overlapping changes among them. Different from the results of conflict detection, Differential EvoSuite did not rank first this time. We believe how each tool works and uses the budget may motivate these results. However, a new study is necessary in order to compare the detection of behavior changes under different budgets. As each tool could detect at least one exclusive behavior change, no combination of tools could capture all reported behavior changes. However, in case we want to know which tools could be combined to report the highest detection rate, we might combine EvoSuite, Randoop Clean, and Differential EvoSuite, as they report together 85 out of 89 behavior changes.

Table 7 – Distribution of behavior changes by unit test tools and binary files. Down arrows stand for behavior changes detected by the current cell, but not detected by the next cell below. Up arrows stand for behavior changes detected by the current cell, but not detected by the next cell above. Left arrows stand for behavior changes detected by the current cell, but not detected by the next left cell. Right arrows stand for behavior changes detected by the current cell, but not detected by the next right cell. The *Total* row stands for the union of behavior changes detected by all tools for each binary file version.

Unit Test Tools	**Binary File Version**		
	Transformed	**Original**	**Serialized**
EvoSuite	47 (↓33, →16)	45 (↓30, ←14, →38)	15 (↓12, ←8)
Differential EvoSuite	28 (↑14, ↓19, →14)	20 (↑5, ↓15, ←6, →19)	5 (↑2, ↓2, ←4)
Randoop	37 (↑28, ↓7, →22)	22 (↑17, ↓7, ←7, →17)	24 (↑21, ↓2, ←19)
Randoop Clean	38 (↑8, →24)	18 (↑3, ←4, →14)	22 (←18)
Total	89	69	29

Source: The author (2022).

Regarding the intersection of detected behavior changes by tools, we observe Randoop and Randoop Clean present the highest intersection with 30 changes in common. We believe these detections are motivated by the original behavior of Randoop, which is not changed/extended by Randoop Clean. Next, we have EvoSuite and Differential EvoSuite detecting the same 14 changes, while all tools detect only four changes of all changes under analysis. This way, we may conclude that based on the particularities of the behavior changes, specific unit test tools are more prominent to detect them.

Finally, considering the detected behavior changes, we did not evaluate the potential of detecting them by executing the original project test suites. However, we believe that in those cases, parent commits are expected to change the source code and its associated test code. So if a parent commit introduces a behavior change, it is expected that this behavior is captured by a new or old test case, based on good software engineering practices.

Since we verify the detection of changes across the different binary file versions, we observe that the adoption of Testability Transformations also helps the tools to detect behavior changes. Like in the semantic conflict detection, 20 additional changes were detected when compared to the original binary files (69 changes, seventh row in Table 7). We also observe that serialized binary files detected 15 changes not covered by the binaty files only with the transformations. In these cases, as the tools have access to valid objects, the generated test cases may further and deeply explore the instructions and branches of the target code under analysis.

Regarding the 19 false-negative cases that could be explored, the tools detect a behavior change for 11 of them. However, the reported behavior change was not caused by the

changes involved in the semantic conflict. For example, in Figure 41, we present a test case that reveals a behavior change involving the base and right commits.[20] As presented in Figure 40, the parent right adds an `if` declaration, which checks whether the first argument is `null` (line 4). If so, a new exception called `IllegalArgumentException` is thrown, as the evaluated argument should not be null (line 5). The test case in Figure 41 checks whether that exception is thrown (line 7). The test passes on the right but fails on the base commit, since there is no check regarding that argument yet. Although the reported behavior change occurs due to the changes applied on the method `buildRelativePath()`, the test case directly calls the method `build`, which holds the semantic conflict and later calls the previous method. However, the conflicting changes done by right on this method were not reached, as the `IllegalArgumentException` exception is thrown before.

Figure 40 – General behavior change caused by changes of the Right commit.

```
1 private String buildRelativeUrl() throws UnsupportedEncodingException {
    {...}
3   Object arg = args[i];
    if (arg == null){
5       throw new IllegalArgumentException("Path parameters must not be null:
            " + param + ".");
    }
7   {...}
    return replacedPath;
9 }
```

Source: The author (2022).

Figure 41 – Generated test case detecting behavior change.

```
1 @Test
  public void test1() throws Throwable {
3   {...}
    try{
5     Request request4 = requestBuilder3.build();
      fail("Expected exception of type IllegalArgumentException; message:
          Path parameters must not be null: ping.");
7   } catch (IllegalArgumentException e) {...}
  }
```

Source: The author (2022).

The discussed case is also a scenario that shows the benefits of applying serialization, as the behavior change is detected when the binary files with serialization are given as inputs for the tools. As we previously reported, 15 new behavior changes are detected under these same conditions. Without applying serialization, the tools face problems generating relevant objects, which are required to deeply reach the code instructions; further discussion about this topic is presented in the next section in detail. Although serialization benefits the generation of tests by the tools, this approach still relies on the quality of the project test suites. For instance, serialization cannot be applied if the original

[20] Retrofit - merge commit: 71f622c

project test suite does not cover a target method. Another weakness of our approach is related to the number of serialized objects given for the tools. As part of the generation process budget is used to deserialize objects by the tools, we should previously select the best objects to reach the conflicting target changes. In future versions of our serialization tool (OSean.EX), we might further investigate these topics and present approaches to deal with them.

We must remember that in some cases of our sample, at least one of the parent commits performed refactorings, so we do not expect to detect behavior changes in those cases, as also changes applied on test classes. As we do not manually evaluate the generated test cases reporting the mentioned behavior changes, we may not discuss the occurrence of false positives and negatives in our results. However, regarding the occurrence of false positives, we believe they might occur due to the flakiness of the tests. Since the concept of behavior change does not require that specific code be executed, as required for semantic conflicts, once a test case reports different results for two versions of a program, we might assume the test case detects a true behavior change. Regarding false negatives, they might also occur in our results, considering that the conflicting changes associated with false negatives are not detected as behavior changes.

4.3.3.2 Test Tools' Relevance

Still, regarding the detection of behavior changes, we are now interested in evaluating how effective the generated test cases are when detecting the changes. For example, in a previous scenario, the right commit changed the method `cleanText()` but also the method `removeDuplicatedWords()` (see Figure 6). So we have a case where the test cases could detect two potential behavior changes. This way, we are looking for the number of test cases that reveal a behavior change. The higher the number of relevant tests, the higher the chances of different behavior changes being detected. For example, Differential EvoSuite might generate `test1` (see Figure 7), exploring the changes performed on the method `removeDuplicatedWords()`, and quit generating new tests as its goal is achieved. In this case, the tool generates a test case that presents different results for two different program versions. However, the changes applied on the method `cleanText()` would not be further explored, though they represent another behavior change. In contrast, the other tools would continue to generate tests and possibly detect the remaining behavior changes.

In order to answer this question, our scripts collect the number of test cases that detect a behavior change for each case and tool. For example, consider that for a given case, Randoop Clean generated 50 test cases that report behavior changes, while EvoSuite, Randoop, and Differential EvoSuite generate 40, 30, and 1, respectively. Next, with these numbers for the evaluated behavior changes reported, we use Wilcoxon's test to compare whether the difference among the tools is statistically significant. As a result, Randoop Clean is the most successful tool compared to Randoop and EvoSuite ($p\text{-}value = 0.003$,

p-value = 0.1, respectively). As Differential EvoSuite generates one test for each test suite, we decide not to include its results in our comparison, as any other tool would always be superior regarding this specific metric. We must remember that EvoSuite performs a minimization step responsible for optimizing the final test suite after the generation process. So it is expected that some test cases are discarded, and possibly, test cases detecting behavior changes. This way, our metric about the number of test cases reporting behavior changes for EvoSuite might represent a lower bound.

Since we force Randoop Clean to add multiple calls to a target method, if a test case detects a behavior change exploring that method, we may expect that additional calls to that method would also detect the same behavior change. For example, in one scenario, the tool generated 1063 tests detecting the same behavior change. However, if the target method has multiple changes that lead to behavior changes, in this case, Randoop Clean might or at least try to explore all of them. We must remember that Randoop Clean also forces the creation of multiple objects of a specific type. This way, methods of the classes under analysis might be called with these different objects leading to the possible detection of behavior changes. So these observations might justify the better results presented by Randoop Clean.

4.3.3.3 Target Code Reachability

Since we propose Randoop Clean based on the original version of Randoop, we aim to evaluate whether our improvements can be observed on the generated test suites. As we previously mentioned, we generate reports in both tools, which allow us to compare different aspects of them. First, we want to evaluate whether Randoop Clean generates more tests that directly call a specific target method than Randoop.

We compare the results for each tool after running them under different budgets (1, 2, and 5 minutes). As a result, we observe that Randoop Clean reaches the target more frequently than Randoop in 2 and 5 minutes, but the difference is not statistically significant.[21] Although Randoop Clean is superior when called for these two budgets, it presents better results when the budget of 2 minutes is adopted. For this budget, Randoop Clean is superior in 43 out of 78 test suites, while for the budget of 5 minutes, this number decreases to 29 of 58 test suites. We believe Randoop Clean did not present the same behavior under the budget of 1 minute because the tool uses part of this budget to generate multiple objects of the multiple target classes under analysis; we will explain it later in this section.

We also evaluate the impact of the binary file versions on the target code reachability. Overall, the Testability Transformations duplicates the number of test suites reaching the target method compared to original binary files. This observation is expected since the transformations allow both tools to directly call all methods of the classes under

[21] *p-value*= 0.153 and *p-value*= 0.481, for the budget of 2 and 5 minutes, respectively.

analysis, leading Randoop Clean to be superior in 38 cases (51%). However, for original and serialized binary files, Randoop Clean was superior in 24 (68%) and 17 (77%) cases, respectively. The highest difference between Randoop and Randoop Clean occurs when the tools are used with serialized jars. We believe this difference occurs because Randoop Clean deserializes more objects than Randoop. Additionally, since the tools have access to relevant objects, the target method calls can be appropriately called; otherwise, the calls are invalid and discarded.

Although Randoop Clean presents superior results than Randoop, these results were not good enough to result in new detected conflicts. We believe the main reason that justifies this observation is related to the use of irrelevant objects, when the target method are called by the test cases. Since these objects are irrelevant in order to reach the *interference* state involving the *conflicting* contributions, calling target methods with the same or slightly different objects does not result in detecting new conflicts. Next section, we further discuss irrelevant objects.

4.3.3.4 Object Diversity

The second improvement we propose in Randoop Clean is the generation of multiple objects of a given type (object diversity). This way, we aim to evaluate which tool generates tests with more diverse objects. For example, consider our motivating example (see Section 2.2.3) when the test case has an object `t`, which has the string *"the␣dog"* in its field `text`. Another test with the same object `t` might have a different value for its field `text`, like the string *"the␣␣dog"*, so the target method `cleanText()` would be called with two different objects. This way, the chances of a behavior change or conflict being detected are higher than with the same objects with the same field values. For that, we access the generated reports by each Randoop and Randoop Clean and compare the number of different handled objects with each other.

Regardless of the adoption of binary file versions, Randoop Clean usually generates more diverse objects than Randoop when larger budgets are adopted, but the difference is not statistically significant.[22] For example, when the tools are called with a budget of 2 for the binary files with testability transformations, Randoop Clean is superior in 91 cases (51%), while for the budget of 5 minutes, Randoop Clean is superior in 39 cases (56%). So when the tools are used with larger budgets, the difference increases, making Randoop Clean better in more cases.

Furthermore, regarding the impact of the binary file versions, we also observe that the Testability Transformations help both tools to generate more diverse objects. This observation is expected as the transformations allow the tools to directly call classes constructors or methods, returning required objects of a given type. About the adoption of

[22] The tools present the same behavior when they are called with original binary files under the budget of 1 minute.

serialized objects, Randoop Clean achieves one of the highest differences by being superior in 50 cases (56%). As previously mentioned, Randoop Clean might deserialize more objects than Randoop as it forces calls to constructors or methods that create objects of the classes under analysis. Specifically for serialized objects, Randoop Clean fills its pool with all methods, used to deserialize objects, and randomly selects one of them when required. As Randoop does not have these forced calls, it might reuse older objects leading to less diversity.

As our previous discussion (see Section 4.3.3.3), the good results of Randoop Clean regarding the diversity of objects were not good enough to result in new detected conflicts. Although Randoop Clean has generated more diverse objects, these objects are still irrelevant to detect conflicts. For example, when target methods are called with irrelevant objects, we might expect that the *conflicting* contributions would not be covered, and consequently, the *interference* state would not be reached. In order to address this limitation, the tools should be extended to generate relevant objects or be supported by additional information, as we do here using serialized objects. However, the relevance of these serialized objects relies on the quality of the original project test suites, as we previously discussed. So if projects do not have strong test suites, with high coverage, for example, we might expect irrelevant objects would be serialized.

4.3.3.5 Code Coverage

The last aspect we evaluate between Randoop and Randoop Clean is the coverage achieved by the test suites. As we have a specific target method to cover, we decide to evaluate the coverage at the instruction level. This way, we may assess which tool explores more deeply the instructions of the target code. As Randoop and Randoop Clean do not provide coverage information about their test suites, we access the coverage information by re-running them with Jacoco; we explain our approach in detail in Section 4.2.2.5.

Due to some weaknesses in our approach, we could not generate coverage reports for all test suites associated with our sample. For example, failed attempts to instrument the binary file version due to duplicated files, even located in different places, or problems during the coverage report generation in CSV. We also consider that environmental issues might impact the coverage report; for example, as Randoop Clean generates many test cases, it may require more available resources during the test execution. Because of these problems, in some cases, reports are not generated for the test suites associated with one specific case by both test tools. Since we must compare the coverage regarding a specific case, we discard these cases and evaluate only the cases we have access to both generated reports.

Overall, the tools present the same coverage in most cases. Even under different budgets, the same results are observed when binary file versions with original and Testability Transformations are given as inputs for the tools. Regarding the budget adopted when

generating test suites, we observe that when we adopt larger budgets, the tools generate more test suites sharing the same code coverage, leading to new cases of ties. Regarding the use of serialized objects, we observe that both tools achieve higher coverage compared to previous test suites generated using original and transformed binary files.Thus we may conclude that the quality of serialized objects allows both tools to deeply explore instructions of the target code under analysis. When comparing the results individually, we observe that Randoop Clean is superior to Randoop. This observation shows that Randoop Clean might deserialize more objects and use them when calling the target method.

4.4 DISCUSSION

In this section, we discuss our findings, their implications, and how they can be applied to assistive software development tools.

4.4.1 Semantic Conflict Detection

The occurrence of conflicts negatively impacts team productivity and software quality. Depending on the kind of conflicts, like merge conflicts, there are novel techniques to support their resolution or even avoid developers spending time fixing them. Even with new ways to handle merge conflicts, these new tools do not support behavioral semantic conflict detection or resolution. While no approach is available to detect these conflicts, they will continue to occur and, at some point, show up to the team or even the final user. The later these conflicts are observed, the harder it will be to fix them.

First, we assess the detection of semantic conflicts based on a semantic merge tool using unit test generation. Although we have evaluated four unit test tools, our results show that combining EvoSuite and Differential EvoSuite is the best option to detect all conflicts in our sample. This way, our proposed tool SAM might spend less time generating tests with tools that are already covered. The detection of conflicts relies on heuristics based on the different program behaviors held by the merge scenario commits. Although we have presented four conflict criteria, in our study, we did not detect conflicts for all of them. However, as previously motivated, our criteria cover conflicting situations during merge scenario integrations.

Second, in this study sample, our results report only a few false positives, which could be addressed by further exploring the failed assertions reported by the test executions. For example, in case our conflict criteria are satisfied due to different failed assertions, our scripts might check whether all merge scenario commits fail/pass on the same test case assertion. Additionally, if our conflict criteria are satisfied but the related test case does not exercise the target method, our scripts might double-check whether the target method was executed during the test case execution. If the target method is not executed, we might assume that no conflict involving the target method was detected. This way, if a

semantic merge tool based on unit test generation is available for a development team, we expect developers would most of the time be warned about real conflicts—the team productivity would not be affected for no reason. In the worst scenario, the tool would never warn about any conflict, which is the same scenario of no tool supporting semantic conflict detection.

Third, regarding adopting the proposed testability transformations, one may argue that they could introduce false positives in our results—a concern we are aware of, but analyzing the reported detected conflicts, we observe they are true positives. The transformations contribute to increasing the testability of the code under analysis without major drawbacks, allowing the tools to directly access and call all elements of a target class.

Finally, we believe the adoption of serialization may support the detection of conflicts though we have not detected any conflict in our study based on this approach. As we previously explained, serialization addresses an essential weakness of unit test tools, which is the creation of complex objects. While tools face problems in generating these objects with multiple dependencies, for example, serialization provides authentic and relevant instancies of those objects for the tools during the generation process. This way, we expect the target code might be not only be reached but also with relevant objects, which would lead to the eventual interference state. As a result, different parts of the target code might be covered, as we observe in our results here.

Another novelty about using regression testing is how the semantic conflict is detected. Other approaches, such as static analysis, could inform that a prominent conflict was caused by the data flow involving two variables. Despite the chances that this merge scenario could represent a false positive, the developer should stop her/his work and spend some time analyzing it until a decision could be made. In case a conflict is detected, it would also be necessary to define how the conflict could externally be observed. Adopting regression testing may decrease the effort to understand how a conflict occurs. The test case limits the amount of source code that should be analyzed and changed to fix the conflict. Second, the test cases can be used to observe the external final state of a program in all commits of the merge scenario, so the behavior changes individually for each commit. Regression testing could also be applied during the conflict fix process. The developer could use these test cases to verify whether the semantic conflict is observed while changes are applied to fix the conflict.

In practice, semantic merge tools based on regression testing, as we evaluate here, can help developers detect semantic conflicts. Due to the observed low number of false positives, the benefits can be obtained by avoiding major costs on wasted developer effort. However, due to the significant number of observed false negatives, developers should not exclusively rely on our semantic merge tool SAM to detect semantic conflicts. They should still try to catch such conflicts by reviewing the code and executing project tests.

Although our primary goal in this study is to evaluate the potential of detecting

semantic conflicts using unit test generation tools, we are also interested in evaluating their potential by detecting general behavior changes. As previously motivated, if a behavior change is detected due to changes applied to the method holding a semantic conflict, we might conclude that the tools were half a way to detect the conflict.

Unlike the reported test conflicts, many cases of our detected behavior changes are captured by exceptions thrown during the target code execution. Detecting a behavior change under these conditions represents a new pattern of detection by tools that we did not observe yet. This observation allows us to conclude that eventual test conflicts that might result in thrown exceptions might be captured by our current approach. However, in our sample, we do not have cases of conflicts under these conditions.

As a final remark, although the proposed merge tool SAM focuses on detecting test conflicts, our tool does not support fixing the conflicts due to the particularities of this conflict type. Different from build conflicts (see Chapter 3), test conflicts are harder to fix as developers must take into account the semantics of the integrated changes that cause the conflicts. This way, applying straightforward changes, like those adopted for build conflict fixes, might not be the most indicated solution considering this practice might lead to the integrated code to unexpected behaviors. As a result, to fix test conflicts, developers must be aware of the program specifications, and then, they might apply the required changes aiming to meet those specifications.

4.4.2 Improvements for SAM

During our evaluation of SAM during our empirical study, we observed some improvements or extensions that might be applied in future versions of our tool. In this section, we think about different usages of SAM and the different contexts our tool might be adopted. Additionally, we present different approaches that might be used to get a good impression when adopting SAM.

Using SAM in a reactive way to detect conflicts. Knowing that SAM requires a specific budget of time to generate the test suites and the time to execute the test suite on each commit of a merge scenario, someone may argue about the real potential of the use of SAM by individual developers. For example, consider that SAM is used with a budget of 5 minutes to generate a test suite, and each execution of the tests on each merge scenario commit takes 3 minutes. Since SAM executes each test suite three times on each commit, in the end, this step would take 41 minutes, which is a significant amount of time for a developer waiting for an answer by SAM. Furthermore, the developers should offer an environment supporting the execution of the steps previously described. So such a concern is a valid issue that might negatively impact the use of SAM.

This way, we might consider using SAM based on a reactive approach. Specifically, SAM might be used as an additional service added to the pipeline of CI services. For example, when a developer sends a new contribution to be integrated into the main development

line of a project, during the build process in the CI service, SAM might be called and perform its analysis. If a conflict is reported, this information might be displayed on the PR page or other information source, and the associated roles can take knowledge of the problem. So the developer, who sends the contribution, would not waste time waiting for an immediate answer and keep working on other tasks, while no local private environment would be required.

Reusing original project test suites as input for SAM. For projects with solid and robust test suites, we might also extend SAM in a way that the detection of conflicts would rely on the execution of the project test suites using tests as *full specifications*. Instead of using unit test tools to generate test suites, SAM would reuse the project test suites, and, based on our conflict criteria, SAM would report the occurrence of conflicts. If a parent adds new tests during its contribution, these tests might also be used by SAM. Similarly, for projects that follow good software engineering practices, like the adoption of CI and committing new contributions or bug fixes with the associated test, the potential to detect conflicts in these is higher. Knowing that these new tests might explore new elements added or changed by a parent commit, running them in the base commit might result in error statuses. However, SAM already handles these issues by discarding those tests (see Section 4.2.2).

Still about the use of *full specifications*, the generation of test suites by unit test tools based on parent commits could be extended to other commits as well. For example, SAM might generate tests based on the merge commit. As a result, instead of using generated tests as *partial specifications*, SAM would use them as *full specifications*, as the target code used to generate the tests holds all changes performed by the parent commits.

Additionally, we might consider a third way of using SAM combining the use of generated and original project test suites. For example, for projects with test suites that do not detect conflicts, SAM might start the generation of new tests and further execute them to detect conflicts. This way, we might offer SAM as a customized tool, as the developer might choose which option better fits the project context: (i) reusing original project test suites, (ii) generating new test suites, and (iii) combining the previous approaches.

4.4.3 Improvements for Unit Test Generation

We observed the following limitations and weaknesses of the generated unit tests in our specific context. For instance, we observed a limitation to create complex objects with internal or external dependencies. As presented in Section 4.2.2, we tried to address some of these limitations applying testability transformations in the code under analysis. However, this rather increased code testability, not the quality of the generated test suites. Based on this perspective, we manually analyzed some generated test suites of false-negative cases to understand the limitations of these suites better.

Recall that, during our manual analysis, we document the conflicts in our scenarios in detail. Based on these descriptions, we try to apply changes to the generated test suites

of false negatives and verify whether the test suites could detect the conflicts with the minimum possible number of changes. Interestingly, the improvements we propose here are not fully exclusive to detect test conflicts, but can help to improve unit test generation tools in general.

Relevant object creation is required to reach the interference location. The unit test generation tools face some problems regarding the creation of objects that should be used directly or indirectly by the test cases. Many test cases finish their executions due to failing attempts to access fields or calling a method from a received parameter, which was not well-created. As an example, consider the case of project Jsoup.[23] The attempt to access fields of the object `textNode0` throws a `NullPointerException`, since the object was not well-created. Giving relevant objects for test cases like this is necessary to the test case reaches at least the *interference* location. Based on this, we extend this test case and the conflict is detected. This topic is also related to cases that *methods holding the conflict are not adequately called*. In these cases, besides the creation of relevant objects, a sequence of method calls must be done aiming to assign values to objects, that are required to reach the *interference* location.

Relevant assertions must explore the propagated interference. Considering the hard work required to reach the location where the conflict occurs, we expected that the assertions could better explore the *propagated* interference. For example, as we mentioned above, one conflict of project CloudSlang,[24] the parent commits change the same `array`. The test case generated for this scenario correctly creates the object, calls the method, where the conflict occurs, and saves the method return object into a local variable. However, the generated assertion checked whether the local variable was `null` instead of exploring the object size. So, depending on the type of a method return object, the unit test generation tools could explore defined aspects that could detect the conflict. This way, tools could provide a list of handlers based on the types of objects under analysis. For example, for array objects, these handlers would force the assertions to explore their size and contents. For strings, assertions might explore comparisons between different strings, as also whether substrings are part of others, and so on.

Relevant assertions rely on the way an interference is propagated. Test case assertions often use the object returned by a method, but not objects that are passed as parameters. For example, again analyzing the changes performed of previous mentioned merge scenario of project Jsoup, the method `outerHtmlHead` requires three parameters as input, not returning any object (void method). However, a semantic conflict occurs, and the first parameter holds the *propagated* interference. The test case generated by EvoSuite focusses on verifying whether an exception is thrown during its execution. The assertions should not be restricted to explore objects returned by a method, but also other objects that are

[23] This case refers to merge commit a44e18a in project Jsoup.
[24] This case refers to merge commit 20bac30 in project CloudSlang.

used by a method or any other way of communication.

On a final note, the weaknesses of the tools we observed may be motivated by the diverse sample of real projects we adopt here. Previous work assessing Randoop (PACHECO et al., 2007), for example, focuses mostly on generating tests for APIs. For EvoSuite (SALAHIRAD; ALMULLA; GAY, 2019), previous work considers other kinds of projects, but many of them come from the same owner. In other work (FRASER; ARCURI, 2014), the evaluation did not focus on detecting behavior changes.

4.5 THREATS TO VALIDITY

Construct Validity

As explained above, we cannot assess semantic conflict occurrence without having access to the developers intentions or specification of the changes they make. So our study focuses on interference occurrence. As manually assessing global interference, and generating and running tests for the whole system, would demand considerable effort, our study is restricted to local interference occurrence. So it is possible that our sample has merge scenarios that parent commit changes do not interfere with each other locally, but interfere globally. The opposite can also occur. So the number of false negatives and false positives with respect to a global notion of interference could be different than our results report. Nonetheless, regression tests could detect global interference if the interference is propagated, and if we generate tests for other classes in addition to the one that integrates the parallel changes made by two developers.

Aiming to increase the testability of the source code under analysis for the unit test generation tools, we decided to apply testability transformations before performing our analysis. For example, we change access modifiers to `public`. This transformation does not semantically change a program; it only makes some elements reachable for the test cases. If a semantic conflict can be observed accessing a class field, but this field is `private`, the unit test generation tools would face many problems trying to indirectly access this attribute without the transformation. Some may argue that, without this transformation, such conflict could never be observed. That might be true if indirect access, for instance with accessor methods, is not available. However, this is not a problem here since our ground truth and analysis focuses on locally observable interference, not globally observable interference.

Internal Validity

Another code transformation adopted by our approach is the change of access modifiers to `public`. This transformation does not semantically change a program; it only makes some elements reachable for the test cases. For example, if a semantic conflict can be observed accessing a class attribute, but this attribute is `private`, our approach would face

many problems trying to access this attribute without the transformation indirectly. Some may argue that without this change, this conflict could never be observed. That is partially true if we consider our notion of local interference, but wrong if we consider the notion of global interference, as local interference may lead to observable global interference.

When creating the interference ground truth, knowing the results of the test generation tools before the manual analysis could have influenced the verdict. For instance, knowing that the tools were not successful for a given scenario brings the risk of precluding a more in-depth analysis from our side. To reduce this threat, we involve a group of six researchers, which were split in pairs during the analysis, and demand they provide an explanation of why there is no interference; this often requires understanding the changes in detail to detect refactorings, changed state elements, and how they impact each other. The risk is significantly reduced for the cases in which the tools are successful, as the threat can be minimized by analyzing the interference revealing threat, running it, and manually checking whether the test case assertions focus on the changed state elements.

As we discuss when presenting our results, we remove from our analysis test cases that are not compilable on at least one commit of the triple of commits in the merge scenario. This limitation come from the fact that test suites are generated for a specific parent commit, but has to be executed also in the base and merge commits. However, if we could only consider the test case, there is a chance the conflict could be detected. So our number of false negatives could be less than we report here. To address this issue, we propose to extend all commits of a merge scenario to a common state so that the test suites would be at least executed in all of them.

Among the test cases we use to perform our study, there is a chance some of them are flaky tests, test cases that may either pass or fail without a specific reason. We decide to remove this kind of test from our analysis as they could introduce both false positives and negatives in our results. Nonetheless, these tests could also detect test conflicts if environmental or configuration problems cause the flakiness, and not the different merge scenario commits. Analyzing our results, we did not identify flaky tests in our executions.

External Validity

Our results are specific to the context of open-source GitHub Java projects. The code transformations, as we discuss, positively impact our results and contribute to increasing the source code testability; in some cases, also detecting the conflict. Applying our proposal of semantic merge tool to other programming languages would require test generation tools for the desired language and also the code transformations, if applicable.

5 RELATED WORK

In this chapter, we summarize and discuss semantic conflict related work. First, in Section 5.1, we discuss empirical studies related to build conflict occurrence and approaches adopted to detect and treat them. Next, in Section 5.2, we do the same for test conflicts discussing related work.

5.1 BUILD CONFLICTS

Empirical studies about build conflicts provide evidence of their occurrence in practice, but none of them investigate the *causes* and *resolution patterns* for build *conflicts* as we do here. Early studies (KASI; SARMA, 2013; BRUN et al., 2011) focus only on reporting conflict *frequencies* (the focus of RQ1 in our first study, see Chapter 3), and use that as a motivation for the tools they propose. In total, these early studies analyze seven open-source GitHub projects, and consider that a merge scenario has a conflict if the corresponding merge commit build fails. We are more conservative because we additionally check whether the changes are related to the build error message. This way we do not consider as integration conflicts build breakages purely inherited from parents. They also replay build creation in their environment, which might introduce noise due to differences to the development environment used by project teams. For example, variations in the environment configuration, or an unavailable dependency, could break the build process, leading the authors to confirm a non existing conflict. Contrasting, we opt for collecting actual build logs created when developers contributed to the projects main repositories. At worst, we create builds in the same cloud environment. We aimed to include the studies analyzed by the previous related work in our study; however, only one project has adopted Travis, and no conflict was reported in our results.

Although we consider a much larger sample than the two mentioned early studies, it is nevertheless biased by practices such as automated build support and continuous integration. Because of that, as better explained earlier, many conflicts do not reach main public repositories as analyzed here. Once developers have these support locally, it is expected they sent their individual contributions to the main repository without problems. Thus, the main repository will be always consistent. This phenomenon helps to further justify the much lower conflict frequencies we observe in our study. On the other hand, the mentioned previous work only consider *clean merges scenarios* as valid subjects, while we also consider scenarios resulting from merge conflicts. To identify conflicts, they try to build locally the *clean merge scenarios* (merge scenarios without merge conflicts). Since they identify conflicts only based on the build process status of the merge commit, some false positives can be introduced. For example, the build process of a merge commit can

fail due to previous changes performed in one of its parent commits. Related work (KASI; SARMA, 2013; BRUN et al., 2011) do not clearly explain if they check the build process status of merge parent commits. In the same way, any variation in the environment configuration or an unavailable dependency could break the build process leading to results that are not entirely consistent with what actually occurred in practice. Finally, although the authors observe conflict frequency, they do not investigate conflict causes nor resolution patterns as we do here. We are not aware of previous studies that explore our research questions 2 and 3.

Sung *et al.* (2020) investigate build conflicts involving different forks in a private project. In this way, conflicts arise when developers change their individual forks (*downstream*) and sync them with the main development fork (*upstream*). To detect conflicts, they manually investigate the data collected of a private C++ project for three months. Then, they classify the conflicts based on the changes performed by the conflict commit fixes. We also perform a manual analysis in our study, but most of our conflicts are automatically detected (72%); for the conflicts detected by manual analysis, our scripts also provide information to guide the analysis and mitigate eventual errors. Although they investigate conflict occurrence in C++ projects and adopt a different terminology, the conflict causes reported are mapped in our catalogue of conflicts for Java projects. They also confirm that *Unavailable Symbol* is the most recurrent build conflict cause (67%), while the remaining causes present distributions in conformity with our findings. However, we report uncovered causes by the related work like *Duplicated Declaration*, *Unimplemented Method*, and *Project Rules*. We believe these uncovered causes are motivated by the different projects we consider for our sample.

Next, for conflicts caused by *renames* involving *Unavailable Symbol* and *Incompatible Types*, the authors evaluate the potential of automated fixes for build conflicts. In this way, they propose a prototype tool called MrgBldBrkFixer. For a broken build caused by a conflict, the tool analyzes the files involved in the conflict using GumTree (FALLERI et al., 2014) and suggests a fix for the error. Next, all files involved are updated with the suggested fix. Although most of this process is automatically done, there are some manual steps required to be done by developers. For example, for a build conflict caused by a method rename, when using MrgBldBrkFixer, the developer must manually inform the name of the missing method and the file where that method should be available. Our prototype also adopts a similar methodology, but its process is completely automated as it adopts the scripts we use to detect the conflicts in our study. Thus, our prototype could be integrated into developers' workspaces and used without requiring any further human help.

Finally, the authors also decide to evaluate MrgBldBrkFixer. As a result, they observe a success rate of 39% of the conflicts analyzed (64 out of 164). As we discuss in Chapter 3, we do not evaluate our prototype, but we believe the current implementation could have

automatically fixed 21% of our sample's build conflicts. Furthermore, our tool does not cover fixes for *Incompatible Types*; it covers three conflict causes, including conflict causes not even mapped by the related work (*Unavailable Symbol*, *Duplicated Declaration*, and *Unimplemented Method*).

Wuensche *et al.* (2020) suggest an approach based on static analysis and a tool to predict the occurrence of build and test conflicts, as they formally call *higher-order merge conflicts*. Based on the changes performed during a merge scenario, they (re)build a call graph and detect potential dependencies among merge scenario code fragments, leading to a conflict. To detect build conflicts, they analyze known build records of build conflicts exploring the merge scenario's changes. Next, they extract pattern changes that lead to the build conflict occurrence. As a result, they provide a list of three causes of build conflicts, but our findings present a better cover as we report a catalogue that splits the build conflict occurrence into six causes. Finally, we report two categories not mapped by them, *Unimplemented Method* and *Project Rules*.

Regarding the distribution of the causes, just like our findings, the authors report *Unavailable Symbol* as the most recurrent cause for build conflicts. However, for the other causes, they present different frequencies. While the two remaining cause frequencies of the related work are 8% and 18%, respectively, our remaining five causes (all causes except *Unavailable Symbol*) vary from 2% to 10%. We believe the difference among these frequencies is motivated by the diversity of projects we analyze in our study; while the authors investigate build conflict occurrence considering one single C++ project, we analyze 451 Java projects finding conflicts in 37 of them. Finally, we go further, not only detecting conflicts but also detecting resolution patterns adopted to fix them. Based on these resolution patterns, we propose a tool that suggests fixes for build conflicts and automatically resolves them.

Regarding the occurrence of false positives, we believe the actual rates discussed and expected by the authors are higher when analyzing other projects. While the authors discuss the impact the level of transitive includes has on false positives, they do not discuss how the tool would behave when analyzing merge scenarios without build conflicts. For example, consider a merge scenario where one branch changes a method while another branch adds a new call for that method. The tool would report a conflict as there is a path of calls from one changed unit to another. However, there is no guarantee the changes are conflicting. Our analysis is more accurate as they check whether a broken build is caused by conflicting contributions or using further heuristics, see Chapter 3. For false negatives, just like us, they report conflicts based on a set of known patterns; so if the patterns do not cover a conflict, a false negative is observed. However, even if our scripts fail to detect a build conflict, it could be reported as a broken build caused by *Post integration changes*, see Chapter 3.

The authors claim that depending on the transitive includes, the call graph used to

detect build conflict can be constructed ranging from two minutes to two hours.Although we did not evaluate the time required to run our tool, we believe not so much time would be spent by our scripts as it involves the generation of syntactic diff and parsing of build logs. In this way, our tool would spend most of the time on the merge scenario build process under analysis. For build processes that take low and medium time (budget), our tool would present a better performance than related work.

Mesbah *et al.* (2019) present DeepDelta, a tool based on a deep neural network that suggests changes for common compilation build problems. Initially, they observe and categorize build problems and how developers fix them. Although they do not investigate build conflicts as we do here, they report five build problems that are already mapped in our catalogue. To suggest a fix for these problems, they provide a technique that learns code changes patterns based on AST diffs between the commit associated with the conflict and its fix. DeepDelta correctly suggests fixes for two problems (*Unavailable Symbol* and *Incompatible Method Signature*), that is not supported by our prototype. However, our prototype supports three causes of build conflicts, also including *Unavailable Symbol*.

A number of studies investigate the causes of errors in the build process (SEO et al., 2014; KERZAZI; KHOMH; ADAMS, 2014; BELLER; GOUSIOS; ZAIDMAN, 2017a; RAUSCH et al., 2017; VASSALLO et al., 2017; HASSAN; ZHANG, 2006; GHALEB et al., 2019), but none of them investigate whether these errors are caused by conflicting contributions, and are therefore related to collaboration or coordination breakdowns. Seo *et al.* (2014) investigate build error causes, categories, and their related frequencies in general, not relating them to integration changes or conflicts as we do here. So they mostly explore the causes for individual developers changes that lead to a broken build, but do not study integration conflicts as motivated here. Although they analyze millions of builds from different projects, they consider projects from a single company. Concerning their error categories, four out of five can be mapped to our conflict causes. The only difference is the fifth category that is related to *syntax* errors. We do not consider this kind of problem as a build conflict because, assuming the parents do not have syntactic errors, a syntactic problem in a merge commit can only result of changes performed by an integrator right after integration. Thus we consider this category only for build errors caused by post integration changes. We also have one conflict cause that has not been observed by Seo *et al.* (2014) (*Project Rules*, see Table 3). The way they treat *Unavailable Symbol* is also different. While they consider all references to an unavailable element as a single problem, we further classify in three subcategories: *classes*, *methods*, and *variables*. This new perspective is necessary and valid when we think about repair tools and their approaches to fix them, as explained in Section 3.3.2. Just adding the missing elements might introduce inconsistencies in the project, resulting in situations such as having two versions of the same class, but in different packages. More important, they simply bring the frequencies of each build breakage cause, not performing further analysis to understand how many of these are actually caused by

interfering contributions, nor understanding the structure of the changes that lead to the breakage and to the associated fix.

The other studies have the same limitation: they focus on build breakage, not integration conflicts. For example, Rausch *et al.* (2017) and Vassalo *et al.* (2017) explore the overall causes of broken builds. Consequently, they also do not go further and reveal the structure of the changes that lead to conflicts, or study resolution patterns, as we do here. Both studies group error categories based on the build process phase in which the error occurs. Although they consider compilation errors as causes for broken builds, they do not investigate further the errors grouping them into specific categories.

Some studies have investigated build process under the perspective of non-deterministic commit (LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017; HERZIG; NAGAPPAN, 2015; ELBAUM; ROTHERMEL; PENIX, 2014). In these cases, different build process associated with a single commit present different results. It may occurs due to environment issues, semantic problems, or just by chance. Although previous studies identified these problems in test failures, they do not occur in our study because of the nature of build conflicts. For instance, even if a build process present a broken status, we further analyze the contributions to ensure the conflict build occurrence. So the broken build status is just a start point of our analysis. If the contributions are not conflicting with each other, we do not consider this case as a build conflict.

While Rausch *et al.* (2017) list compilation problems as causes of broken builds, Vassalo *et al.* (2017) group errors based on build process phase that the error occurs. However, they do not observe if the build error was caused by conflicting contributions among developers. The studies identify general problem categories, but they do not go further and analyze each category. For example, compilation problem is classified as a category, but there is not specific causes for this problem. They also do not focus on the resolution patterns adopted to fix these broken builds. Kerzazi, Khomh and Adams (2014) present a study investigating the impact of broken builds on software development. For that, they perform a qualitative and quantitative study. Just like one of our findings, the qualitative study reports that the primary cause of broken builds is due to missing classes (*Unavailable Symbol*). However they do not investigate other causes like we do here, and also do not present a catalogue of causes. They investigate the costs of broken builds measuring the time spent to fix the broken build.

We measure the cost of broken builds, but instead of checking the time spent, we evaluate the changes performed to fix the broken build. So they also do not present a catalogue of resolution patterns as we do here.

Finally, Ghaleb *et al.* (2019) investigate build breakage causes in Travis CI builds. For that, they adopt a similar methodology to our work, as they analyze GitHub projects linked to Travis CI. Initially, they perform a manual analysis with selected builds to study the causes of the breakages, extract error messages and categorize the causes.

Next, they create heuristics to automatize this process. As a result, they report 33% of the all analyzed breakages are caused by environmental factors. They explore these factors in details reporting 11 categories of breakages divided in 60 subcategories. From all environmental breakages, 79% are caused by the subcategories *Internal CI errors* or *Issues related to external services and exceeding limits*. Although we do not focus here on studying environmental causes of broken builds, we use the mentioned causes to filter out merge scenarios associated with this kind of broken build (see Table 2 in Chapter 3). For the remaining categories, we do not detect broken builds caused by them. It happens because some categories are program language dependent and specific for Ruby projects. The authors analyze not only projects written in Java as we do here. For example, the category *Ruby and bundler issues* is related to errors that happens during the resolution of dependencies required by a project.

Vassalo *et al.* (2019) present *BART*, a tool that has a similar goal to our build conflict repair tool. Initially, the authors investigate how broken build reports could be easier to be understood by developers. They propose a tool that summarizes the causes of Maven projects' broken builds and suggests possible solutions linking online external resources. Although we also present a tool supporting developers when dealing with broken builds, our tool covers only compilation breakages caused by build conflicts. *BART* is more general as it supports all kinds of build breakages. Regarding the summarized information of broken builds, both tools adopt a similar approach. For example, for a broken build caused by a compilation problem, both tools present, when available, the main cause of the breakage, the file, and the line where the breakage takes place. However, the tools adopt different approaches when supporting developers. While *BART* suggests hints to fix the build breakage based on the build log and possibly searching for solutions online in discussion forums, our tool suggests fixes for the breakage. If the developer accepts the suggestion, the fix is automatically performed without further human intervention.

The authors also evaluate the use of *BART* by performing an empirical qualitative study with 17 developers. For that, they perform a controlled experiment followed by a questionnaire. As a result, they find that 76% of the participants benefit from using the tool. The participants emphasize the speed achieved to solve the breakages and build breakage summarization as the most benefits. Although we do not evaluate our tool in this study, we believe the related work benefits could be applied to our tool. First, our tool also presents to the developer that build breakage summarization. Second, our tool automatically fixes build breakages that could reduce even more the speed to fix build breakages.

5.2 TEST CONFLICTS

As previously discussed, a couple studies have investigated the occurrence of test conflicts (KASI; SARMA, 2013; BRUN et al., 2011). Kasi and Sarma (2013) analyze four Java

projects and report a test conflict rate raging from 5% to 35% of the evaluated merge scenarios. Brun *et al.* (2013) report a conflict occurrence rate raging from 3% to 28% from three Java projects. In both studies, to detect test conflicts, they rely on original project test suites, which are often not enough for detecting interference as we explore here. They locally build the merge commits, and if the build process fails because of failed tests, they consider the merge scenario having a semantic conflict. The authors did not eliminate possible flaky tests in both studies, as we do in our study by executing the test suites multiple times. The failed tests are not executed on the parent and base commits of the merge scenario, as we do here, which may result in false positives, as the failed test may occur due to the changes exclusively performed by one parent. These studies have also investigated ways in which conflicts can be prevented early, thereby minimizing their impact on productivity. Sarma *et al.* (2012) present Palantír, a workspace awareness tool, which aims to minimize the occurrence of conflicts by notifying the developers of parallel changes in the same artifact. Brun *et al.* (2013) propose incorporating speculative analysis for early detection and prevention of conflicts.

Concerning about the use of testing to detect behavior changes, previous studies have investigated different ways to evaluate this goal. Evans and Savoia (2007) combine regression testing with progression testing to detect preserved, altered and eliminated behavior of a program. They evaluate a parser written in Java showing significantly better detection rates than regression testing alone. Jin *et al.* (2010) leverage change analyzers to generate test cases for changed parts of a software program. Shamshiri *et al.* (2013) present EvosuiteR, a test generation tool for differential testing that uses search-based algorithms to find regression faults on different versions of a program. While the previous studies evaluate the detection of regression faults between two different versions of a program using regression tests, in this work, we evaluate the potential of regression tests to detect semantic conflicts on merge scenarios (three different versions of a program). We also consider in our evaluation a sample of diverse real Java projects, while the previous studies only consider small or toy projects. Regarding using fuzzing to detect conflicts, we consider Randoop as a tool that works similarly to this approach. The general idea of fuzzing is to generate multiple entries for a target code to evaluate the code behavior under these multiple entries, like throwing exceptions. Randoop adopts this idea as multiple entries are generated during the generation process and further used on a sequence of method calls (PACHECO et al., 2007). Additionally, the tool also generates assertions based on the observed behavior of the program under analysis. Based on the type of generated tests, these assertions might be responsible for capturing object states (regression tests) or errors (error tests) during the test execution.

Cavalcanti *et al.* (2017, 2019) and Tavares *et al.* (2019) conduct empirical studies where they analyze merge scenarios and compare the accuracy of different merge resolution techniques: unstructured, semi-structured, and structured merge. They also propose a new,

semi-structured tool with significant advantages over unstructured merge tools by reducing the false-positive and false-negative rates of earlier semi-structured tools. When comparing with structured merge, the authors verify semistructured merge reports more false positives, but presents less false negatives. Overall, they find that exploring more structure does not necessarily improve merge accuracy. Contrasting with our investigation here, their proposed tools are not able to detect behavioral semantic conflicts, only syntactic and static semantics conflicts.

Nguyen *et al.* (2015) present Semex, a tool for detecting which combination of merged changes causes a test conflict based on a technique called variability-aware execution (NGUYEN; KÄSTNER; NGUYEN, 2014). First, the tool separates the changes done by each parent commit in the merge scenario and encodes each one using conditionals around them (`if` statements) to integrate all these changes in a single program. Semex then uses variability-aware execution to detect semantic conflicts by running existing project tests, if available, on this single program, exploring all possible combinations of the encoded changes. The tool then knows which combinations of commits lead to test failure and reports the set of commits that, if integrated, would cause a test conflict. Reporting a conflict exclusively based on the failure of a test in the merged code does not always imply a conflict or interference. If the test fails in one of the parent commits too, failure in the merge might simply indicate inheritance of a defect. If the test passes in both base and a parent commit, failing in merged code might simply be caused by a non-interfering behavior change from another parent commit. That is why we propose different criteria, based on the idea of tests as partial specifications of the changes to be integrated. Like the previous study, we also rely on tests to detect conflicts. However, instead of relying on existing project tests, which are often missing or have limitations, as described previously in Chapter 4, we use test generation tools to generate new test suites. We also rely on and assess the use of test generation tools to detect conflicts, instead of relying on existing project tests, which are often missing or have limitations, as described previously in Chapter 4. Finally, Semex is preliminarily evaluated using PHP toy projects, with manually created test cases especially aimed at revealing conflicts, as the main focus is on assessing the potential of variability-aware execution for identifying conflicting branch combinations, not really the potential for conflict detection, as we do here. Moreover, in our study we use non trivial Java open-source projects.

Sousa *et al.* (2018) propose *SafeMerge*, a tool that leverages compositional verification to check *semantic conflict freedom* in merge scenarios. In principle, this kind of static analysis should lead to more false positives and fewer false negatives, when compared to the use of tests as we propose here. An evaluation with 52 merge scenarios indicates that *SafeMerge* reports 75% of the scenarios without conflicts, with a false positive rate of 15%. However, analyzing the merge scenarios reported with conflicts, we concluded that some of them do not represent conflicts according to our criteria. In these cases, the changes

involved do not interfere with each other or are only refactorings, leading to no behavior change and consequently no interference.

As previously introduced in Section 5.1, Wuensche *et al.* (2020) suggest an approach based on static analysis and a tool to detect and predict the occurrence of test conflicts, as they formally call higher-order merge conflicts. Based on the changes performed during a merge scenario, they (re)build a call graph and detect potential dependencies among merge scenario code fragments that lead to a conflict. To detect conflicts, the authors manually analyze build records of merge scenarios and extract change patterns that lead to conflicts based on their analysis. As a result, the tool reports 22 potential conflicts out of 1489 merge scenarios. To validate the potential conflicts, the authors search for bugs reported dated after each merge scenario occurrence. However, they do not find any bugs, and consequently, do not confirm the potential conflicts as actual occurrences. Since the authors rely on reported bugs to validate conflicts, some potential conflicts may be true positives. However, they were not reported as bugs due to their unawareness, caused by the project test suite's low coverage and quality or no report from the system users. This work adopts a different approach by generating new test suites for projects under analysis and using these generated tests as partial specifications to detect semantic conflicts involving the parent contributions. While the tool evaluation considers one C++ project, we analyze 28 Java projects finding conflicts in 3 of them.

Arcuri and Galeotti (2021) adopt a related approach by presenting a set of testability transformations; unlike our transformations, they do not focus on changing code element access modifiers or semantic changes (empty constructor) but support and guide the search algorithm when generating tests. Their transformations cover different goals that not only change the target code (preserving its semantic) but can also be used to provide feedback to the search algorithm generating the required data to test a specific target code. Their core idea is based on Method Replacements, which replace specific method calls at the bytecode level with their customized methods. For example, to address the flag problem, their transformations calculate a heuristic distance h, similar to branch coverage, which refers to the closeness an input is to return a specific value. Once the heuristic distance is computed, target tests are created and used as fitness functions leading the search to generate tests that address them. To evaluate their techniques, they implemented them as an extension of the EvoMaster tool and performed an empirical study analyzing ten Rest web service projects (open-source and industrial ones). The results show that the techniques effectively improve code coverage and fault detection. As the authors evaluate the coverage at the code line and branch level, we could use their tool in our sample and compare their results to the other tools regarding conflict detection and other metrics we consider here. Next, the authors discuss how their transformations might be integrated into unit test tools like EvoSuite. So another applicability would be to perform this integration on EvoSuite or Differential EvoSuite, the most successful tools in our study, and evaluate

the achieved improvements.

Tiwari *et al.* (2020) present a related approach regarding the use of serialization to support unit test tools. Unlike our approach to serializing objects based on a target method during merge scenarios, the authors present PANKITI. This tool monitors an application in production serializing objects when target methods are called. While we serialize the current objects holding the target method and its required parameters, the authors also serialize the returning target method objects. For our context, we are not interested in returning objects as we focus on objects that might let us reach infection states of conflicting contributions and not objects with the propagated infection. Furthermore, infections are not always propagated through returning target method objects; in our sample, we observe infections being propagated through parameters, for example. Next, the objects are deserialized, and the tool extracts test inputs that can be used to reveal behavior changes observed in production. In our study, when we call the unit test tools, we feed them with all generated objects, so the tools might randomly select and use objects. The authors adopt a different approach by generating tests for each object profile (receiving, parameters, and returning objects). This way, the generated tests use the receiving object to call the target method and then compare its return object with the previous object returned in production. This approach might work for the detection of conflicts, considering we generate tests for the remaining merge scenario commits and compare their results with the previous returning object, based on the infection state propagation. To evaluate their tool, the authors analyze three open-source projects observing an improvement of 61% on behavior change detection. Although we did not detect conflicts using serialization, we also observed improvements regarding the detection of behavior changes (15 new changes) after adopting serialization.

Similar to our semantic merge tool SAM, Castanho (2021) presents UNSETTLE, a tool for the detection of semantic conflicts based on potential conflicting change patterns and unit test generation. The author motivates two types of unexpected behavior caused by parent conflicting contributions (emergent and lost behavior), similar to our conflict criteria. To verify the occurrence of a test conflict, UNSETTLE first collects a *diff* between the base and the parent commits. If this diff satisfies one of the potential conflicting change patterns, the tool tries to generate a test case that reveals the conflict using a newly implemented version of EvoSuite. Unlike SAM, UNSETTLE does not adopt a conservative approach by checking the behavior of the generated tests in all commits of a merge scenario when applied. For example, while SAM looks for a test case that passes on right and fails on base and merge commits, UNSETTLE does not always check the test output on the base and merge commits. This way, we believe UNSETTLE might present some false positives not shared by SAM. To evaluate UNSETTLE, the author performs an empirical study analyzing real and realistic cases of semantic conflicts. As a result, the tool detected six out of 19 real conflicts mined from previous work. Our study only

evaluates real merge scenarios of open-source projects, achieving a similar detection rate of nine detected conflicts out of 28. For our sample, we also selected merge scenarios from previous work. Still, we double-checked each case of mutual change in order to evaluate whether the case was a conflict based on our notion of conflicting contributions.

Regarding the issues we observed by the tools when generating test suites, previous studies also bring evidence about these problems (FRASER; ARCURI, 2015; SILVA; ALVES; ANDRADE, 2017; SILVA et al., 2020). These related studies discuss the difficulty of generating complex objects, which are required when calling specific methods under analysis. By complex objects, we consider objects with multiple other objects from internal as also external dependencies. This way, consider a class with a `boolean` attribute and another class with multiple attributes from different dependencies. Generating tests for the first class is easier for the tools since the graph of objects requires only objects of one type, while the graph of objects for the last class involves multiple types. As a result, tools may create these complex objects by assigning `null` values to them. So during the test execution, if these objects are directly called, exceptions are thrown, interrupting the execution. In order to address these issues, we propose feeding the tools with serialized objects leading them to reuse previously objects originally created by the original project test suite.

6 CONCLUSIONS

In this thesis, we investigate how semantic conflicts occur during software development, build and test conflicts. For each semantic conflict type, we investigate how developers' contributions conflict with each other, leading to conflicts. For that, we perform empirical studies to verify their occurrence and understand their causes. As a result, we address a gap in the literature regarding the occurrence and causes of semantic conflicts. Based on our findings, we also provide and propose insights and prototype tools that could be supportive during merge scenarios when developers face these conflicts in practice.

Initially, we explore the occurrence of build conflicts performing an empirical study, when we analyze over 57000 merge scenarios from 451 GitHub Java projects revealing the occurrence of 239 build conflicts (see Chapter 3). Our conflict frequency results reveal build conflicts occur less frequently than reported by previous studies (KASI; SARMA, 2013; BRUN et al., 2013). However, we eliminate threats, which related studies are not aware. We analyze and justify that, highlighting the differences in the experiment design, and showing that our results are more conservative and complement previous results because of the focus on a rather different kind of sample. Besides the frequency results, we find and classify build conflicts causes, and their resolution patterns, deriving two novel conflict catalogues. We also find and analyze build failures caused by immediate post integration changes, which are often performed with the aim of fixing merge conflicts. Most build conflicts are caused by missing declarations removed or renamed by one developer but referenced by the changes of another developer. These conflicts are often resolved by updating the dangling reference. We also observe that build conflicts are not restricted to static semantic problems, but also during static analysis performed during the build problems, *Project Rules*. To resolve build conflicts developers often fix the integrated code, instead of completely discarding changes and conservatively restoring project state to a previous commit. Most fix commits are authored by one of the merge scenario contributors.

Based on the catalogue of build conflict causes derived from our study, awareness tools could alert developers about the risk of a number of conflict situations they currently do not support (SARMA; REDMILES; HOEK, 2012). Program repair tools could benefit from the derived catalogue of build conflict resolution patterns, and the study infrastructure, to automatically fix conflicts. We further explore this by developing a prototype program repair tool that supports developers to resolve some kinds of build conflicts we identify in our study.

Next, we focus on the occurrence of test conflicts evaluating the detection of semantic conflicts using automated test-case generation techniques in another empirical study. As opposed to prior attempts, this strategy requires not much setup effort and does not need explicitly defined behavior specifications. We systematically investigate our interference

criteria upon a manually curated ground-truth dataset originating from real merge scenarios mined from GitHub. We combine unit test generation tools with testability transformation in the source code to be analyzed. We show the potential of a semantic merge tool based on unit test generation. While it was only able to detect nine conflicts out of 28 conflicts from 85 changes on same class element declarations, we did not face many false positives according to our interference criteria(see Chapter 4). This suggests that semantic merge tools based on unit test generation would help developers detect semantic conflicts early, otherwise reaching end-users as failures. The transformations improved testability in two of the four detected interference cases, suggesting that they might be useful for interference detection. We discuss necessary improvements to test generation and make our manually curated dataset available in an online appendix (Appendix 2, 2020), for replication and future technique building upon our proposal of semantic merge tool.

6.1 CONTRIBUTIONS

In general, our findings fill a gap in the literature regarding semantic conflicts on software development. While our results regarding semantic conflict occurrence bring evidence and complement findings from previous studies, our results exploring conflict causes address a gap in the literature. Based on our findings, we provide and propose tools to detect these conflicts and support developers when resolving them.

Regarding the contributions of this thesis particularly for build conflicts, we summarize them bellow:

- A catalogue of build conflict causes spread in six categories covering not only static semantic but also static analysis problems revealed during the build process;
- A catalogue of build conflict resolution patterns split in 17 different patterns;
- An automatic repair tool prototype to fix build conflicts already covering three categories of our catalogue of conflicts;
- A list of unsupported build conflict causes by current assistive tools;
- A catalogue of build breakages motivated by post-integration changes performed by integrators during merge scenarios;
- We provide the infrastructure we developed for the quantitative study in our online appendix allowing its replication (Appendix 1, 2022).

Now, we report the following contributions for test conflicts:

- SAM, a new semantic merge tool based on unit test generation and partial specifications;

- Conflict criteria used to detect test conflicts;
- Quantitative empirical evidence of test conflicts detected using test cases and unit test generation tools;
- A new approach to detect conflicts considering unit test cases as partial specifications;
- Randoop Clean, a new unit test generation tool based on Randoop, and its related evaluation;
- OSean.EX, a serialization tool to serialize objects based on a target method;
- Testability transformations that could be adopted by software practioners aiming to increase the testability of source code under analysis;
- Dataset of changes on mutual class elements with ground truth;
- A list of improvements for unit test generation tools regarding the weanesses they face during the test suite generation;
- Evaluation and comparison of unit test generation tools regarding general behavior change detection;
- We also provide the infrastructure we developed for the quantitative study in our online appendix allowing its replication (Appendix 2, 2020).

6.2 FUTURE WORK

As the presented studies are part of a broader context, a set of related aspects will be left out of scope. Thus, the following topics are not directly addressed in this thesis, but we suggest them as future work:

Regarding build conflicts, we would like to further investigate and explore the next topics:

- Based on our scripts provided to perform our studies, replications could be performed using proprietary projects;
- To better understand the impact of continuous integration practices on build conflict frequence, we might run a new study evaluating merge scenarios before using CI;
- Our automatic repair tool prototype could be extended aiming to cover all build categories of our catalogue and its evaluation;
- Based on our catalogue of build conflicts, an investigation regarding factors that lead to the conflict occurrence could be performed, like previous studies did for merge conflicts evaluating commit size, merge scenario duration, changed files, and

dependencies. However, we are aware that we should increase the number of conflicts in our catalogue for this kind of study;

- Increase our sample of build conflicts, as we might explore other projects, CI servers, build managers aiming to detect new cases of build conflicts.

Regarding test conflicts, we would like to further investigate and explore the next topics as also the associated tools:

- Investigation of test conflicts exploring other scenarios, like when different developers change different methods but one directly depends of the other;
- Investigation of the impact of different budgets to generate test suites by unit test tools regarding the detection of conflicts and behavior changes;
- Improving unit test generation tools based on our list of improvements reported and its evaluation. For example, generation of assertions for all objects involved on a target code using handlers, which might drive the generation based on object types;
- For a future version of our tool SAM, we plan to apply some changes, like adopting a different approach by not creating a merge commit until no semantic conflict is reported. Additionally, SAM might generate test suites not only for the class holding the changed mutual element but also for all classes changed during a merge scenario. We also plan to explore the generation of test suites considering other commits of the merge scenario, like the base and merge commits;
- Extend our tool OSean.Ex in order to support projects with other build managers, like Gradle;
- Further explore the use of serialization regarding the detection of behavior changes.

REFERENCES

ACCIOLY, P.; BORBA, P.; CAVALCANTI, G. Understanding semi-structured merge conflict characteristics in open-source Java projects. *Empirical Software Engineering*, Springer US, v. 23, n. 4, p. 2051–2085, 2018.

ADAMS, B.; MCINTOSH, S. Modern release engineering in a nutshell–why researchers should care. In: *International Conference on Software Analysis, Evolution, and Reengineering*. [S.l.]: IEEE, 2016.

ALMASI, M. M.; HEMMATI, H.; FRASER, G.; ARCURI, A. An industrial evaluation of unit test generation: Finding real faults in a financial application. In: *International Conference on Software Engineering*. [S.l.]: IEEE, 2017.

APEL, S.; LESSENICH, O.; LENGAUER, C. Structured merge with auto-tuning: balancing precision and performance. In: ACM. *International Conference on Automated Software Engineering*. [S.l.], 2012.

APEL, S.; LIEBIG, J.; BRANDL, B.; LENGAUER, C.; KÄSTNER, C. Semistructured merge: rethinking merge in revision control systems. In: *European software engineering conference and Symposium on the Foundations of Software Engineering*. [S.l.]: ACM, 2011.

Appendix 1. 2022. URL : https://spgroup.github.io/papers/build-conflicts.html.

Appendix 2. 2020. Available at: https://leusonmario.github.io/papers/test-conflicts-in-the-wild.html.

ARCURI, A.; GALEOTTI, J. P. Enhancing search-based testing with testability transformations for existing apis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, v. 31, n. 1, p. 1–34, 2021.

BASS, L.; WEBER, I.; ZHU, L. *DevOps: A Software Architect's Perspective*. [S.l.]: Addison-Wesley Professional, 2016.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Oops, my tests broke the build: An explorative analysis of travis ci with github. In: IEEE. *International Conference on Mining Software Repositories*. [S.l.], 2017.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: IEEE. [S.l.], 2017. (International Conference on Mining Software Repositories), p. 447–450.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: IEEE. *International Conference on Mining Software Repositories*. [S.l.], 2017.

BIRD, C.; ZIMMERMANN, T. Assessing the value of branches with what-if analysis. In: *Symposium on the Foundations of Software Engineering*. [S.l.]: ACM, 2012.

BRINDESCU, C.; CODOBAN, M.; SHMARKATIUK, S.; DIG, D. How do centralized and distributed version control systems impact software changes? In: *International Conference on Software Engineering*. [S.l.: s.n.], 2014.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Proactive detection of collaboration conflicts. In: *European Software Engineering Conference and International Symposium on Foundations of Software Engineering*. [S.l.]: ACM, 2011.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, IEEE, v. 39, n. 10, p. 1358–1375, 2013.

CASTANHO, N. G. N. *Semantic Conflicts in Version Control Systems*. 2021.

CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Assessing semistructured merge in version control systems: A replicated experiment. In: IEEE. *International Symposium on Empirical Software Engineering and Measurement*. [S.l.], 2015.

CAVALCANTI, G.; BORBA, P.; ACCIOLY, P. Evaluating and improving semistructured merge. *ACM Transactions on Programming Languages and Systems*, ACM, v. 1, n. OOPSLA, p. 59:1–59:27, 2017.

CAVALCANTI, G.; BORBA, P.; SEIBT, G.; APEL, S. The impact of structure on software merging: semistructured versus structured merge. In: IEEE. *International Conference on Automated Software Engineering*. [S.l.], 2019.

CHACON, S.; STRAUB, B. *Pro git*. [S.l.]: Apress, 2014.

CONRADI, R.; WESTFECHTEL, B. Version models for software configuration management. *ACM Computing Surveys*, ACM New York, NY, USA, v. 30, n. 2, p. 232–282, 1998.

CVS. 2020. Accessed: October 2020. Disponível em: <http://savannah.nongnu.org/projects/cvs/>.

DIAS, K.; BORBA, P.; BARRETO, M. Understanding predictive factors for merge conflicts. *Information and Software Technology*, Elsevier, v. 121, p. 106256, 2020.

ELBAUM, S.; ROTHERMEL, G.; PENIX, J. Techniques for improving regression testing in continuous integration development environments. In: *International Symposium on Foundations of Software Engineering*. [S.l.]: ACM, 2014.

EVANS, R. B.; SAVOIA, A. Differential testing: a new approach to change detection. In: ACM. *European software engineering conference and Symposium on the Foundations of Software Engineering*. [S.l.], 2007.

FALLERI, J.-R.; MORANDAT, F.; BLANC, X.; MARTINEZ, M.; MONPERRUS, M. Fine-grained and accurate source code differencing. In: *International Conference on Automated Software Engineering*. [S.l.]: ACM, 2014.

FILHO, R. S. B. *Using information flow to estimate interference between developers same-method contributions*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, 2017.

FOWLER, M. *Feature Toggle*. 2010. Accessed: December 2017. Disponível em: <https://goo.gl/QfJ6mM>.

FOWLER, M. FeatureBranch. *ThoughtWorks Insights*, martinfowler.com, Sep 2009. Disponível em: <https://martinfowler.com/bliki/FeatureBranch.html>.

FRASER, G. A tutorial on using and extending the evosuite search-based test generator. In: *Search-Based Software Engineering*. [S.l.]: Springer, 2018.

FRASER, G.; ARCURI, A. Whole test suite generation. *IEEE Transactions on Software Engineering*, IEEE, v. 39, n. 2, p. 276–291, 2012.

FRASER, G.; ARCURI, A. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology*, ACM New York, NY, USA, v. 24, n. 2, p. 1–42, 2014.

FRASER, G.; ARCURI, A. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering*, Springer, v. 20, n. 3, p. 611–639, 2015.

GHALEB, T. A.; COSTA, D. A. da; ZOU, Y.; HASSAN, A. E. Studying the impact of noises in build breakage data. *IEEE Transactions on Software Engineering*, IEEE, 2019.

GIT. 2020. Accessed: October 2020. Disponível em: <https://git-scm.com/>.

GOUES, C. L.; FORREST, S.; WEIMER, W. Current challenges in automatic software repair. *Software Quality Journal*, Springer, v. 21, n. 3, p. 421–443, 2013.

GOUSIOS, G.; PINZGER, M.; DEURSEN, A. v. An exploratory study of the pull-based software development model. In: *International Conference on Software Engineering*. [S.l.: s.n.], 2014.

GOUSIOS, G.; ZAIDMAN, A.; STOREY, M.-A.; DEURSEN, A. V. Work practices and challenges in pull-based development: The integrator's perspective. In: IEEE. [S.l.], 2015. (International Conference on Software Engineering), p. 358–368.

Gradle. 2019. URL: https://gradle.org//.

GRINTER, R. E. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work*, Kluwer Academic Publishers, v. 5, n. 4, p. 447–465, 1996.

HASSAN, A. E.; ZHANG, K. Using decision trees to predict the certification result of a build. In: IEEE. *International Conference on Automated Software Engineering*. [S.l.], 2006.

HENDERSON, F. *Software Engineering at Google*. 2017. Accessed: December 2017. Disponível em: <https://arxiv.org/abs/1702.01715>.

HERZIG, K.; NAGAPPAN, N. Empirically detecting false test alarms using association rules. In: IEEE. *International Conference on Software Engineering*. [S.l.], 2015. v. 2.

HILTON, M.; TUNNELL, T.; HUANG, K.; MARINOV, D.; DIG, D. Usage, costs, and benefits of continuous integration in open-source projects. In: ACM. *International Conference on Automated Software Engineering*. [S.l.], 2016.

HODGSON, P. *Feature Branching vs. Feature Flags: What's the Right Tool for the Job?* 2017. Accessed: December 2017. Disponível em: <https://goo.gl/4D2AMv>.

HODGSON, P. Feature toogles (aka feature flags). *ThoughtWorks Insights*, martinfowler.com, Oct 2017. Disponível em: <https://www.martinfowler.com/articles/feature-toggles.html>.

JACOCO. 2022. URL: https://www.eclemma.org/jacoco/.

JIN, W.; ORSO, A.; XIE, T. Automated behavioral regression testing. In: IEEE. *International Conference on Software Testing, Verification and Validation*. [S.l.], 2010.

KASI, B. K.; SARMA, A. Cassandra: proactive conflict minimization through optimized task scheduling. In: *International Conference on Software Engineering*. [S.l.]: IEEE, 2013.

KERZAZI, N.; KHOMH, F.; ADAMS, B. Why do automated builds break? an empirical study. In: IEEE. *International Conference on Software Maintenance and Evolution*. [S.l.], 2014.

KOREL, B.; AL-YAMI, A. M. Automated regression test generation. *Software Engineering Notes*, ACM New York, NY, USA, v. 23, n. 2, p. 143–152, 1998.

LABUSCHAGNE, A.; INOZEMTSEVA, L.; HOLMES, R. Measuring the cost of regression testing in practice: a study of java projects using continuous integration. In: ACM. *International Symposium on Foundations of Software Engineering*. [S.l.], 2017.

LESSENICH, O.; SIEGMUND, J.; APEL, S.; KÄSTNER, C.; HUNSEN, C. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, Springer, v. 25, n. 2, p. 279–313, 2018.

MAHMOOD, W.; CHAGAMA, M.; BERGER, T.; HEBIG, R. Causes of merge conflicts: A case study of elasticsearch. In: ACM. *International Working Conference on Variability Modelling of Software-intensive Systems*. [S.l.], 2020.

MAVEN. 2019. URL: https://maven.apache.org.

MCKEE, S.; NELSON, N.; SARMA, A.; DIG, D. Software practitioner perspectives on merge conflicts and resolutions. In: *International Conference on Software Maintenance and Evolution*. [S.l.]: IEEE, 2017.

MENS, T. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, IEEE, v. 28, n. 5, p. 449–462, 2002.

MERCURIAL. 2020. Accessed: October 2020. Disponível em: <https://www.mercurial-scm.org/>.

MESBAH, A.; RICE, A.; JOHNSTON, E.; GLORIOSO, N.; AFTANDILIAN, E. Deepdelta: learning to repair compilation errors. In: *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2019.

MUNAIAH, N.; KROH, S.; CABREY, C.; NAGAPPAN, M. Curating github for engineered software projects. *Empirical Software Engineering*, Springer, v. 22, n. 6, p. 3219–3253, 2017.

NGUYEN, H. V.; KÄSTNER, C.; NGUYEN, T. N. Exploring variability-aware execution for testing plugin-based web applications. In: IEEE. *International Conference on Software Engineering*. [S.l.], 2014.

NGUYEN, H. V.; NGUYEN, M. H.; DANG, S. C.; KÄSTNER, C.; NGUYEN, T. N. Detecting semantic merge conflicts with variability-aware execution. In: ACM. *Symposium on the Foundations of Software Engineering*. [S.l.], 2015.

OWHADI-KARESHK, M.; NADI, S.; RUBIN, J. Predicting merge conflicts in collaborative software development. *International Symposium on Empirical Software Engineering and Measurement*, 2019.

PACHECO, C.; ERNST, M. D. Randoop: feedback-directed random testing for Java. In: ACM. *ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. [S.l.], 2007.

PACHECO, C.; LAHIRI, S. K.; ERNST, M. D.; BALL, T. Feedback-directed random test generation. In: IEEE. *International Conference on Software Engineering*. [S.l.], 2007.

PERRY, D. E.; SIY, H. P.; VOTTA, L. G. Parallel changes in large-scale software development: an observational case stud. *ACM Transactions on Software Engineering and Methodology*, ACM, v. 10, n. 3, p. 308–337, 2001.

POTVIN, R.; LEVENBERG, J. Why Google stores billions of lines of code in a single repository. *Communications of ACM*, ACM, v. 59, n. 7, p. 78–87, 2016.

RAUSCH, T.; HUMMER, W.; LEITNER, P.; SCHULTE, S. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In: *International Conference on Mining Software Repositories*. [S.l.]: IEEE, 2017.

ROCHA, T.; BORBA, P.; SANTOS, J. P. Using acceptance tests to predict files changed by programming tasks. *Journal of Systems and Software*, v. 154, p. 176–195, 2019.

RODRÍGUEZ-PÉREZ, G.; ROBLES, G.; SEREBRENIK, A.; ZAIDMAN, A.; GERMÁN, D. M.; GONZALEZ-BARAHONA, J. M. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, v. 25, n. 2, p. 1294–1340, 2020.

SALAHIRAD, A.; ALMULLA, H.; GAY, G. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 29, n. 4-5, p. e1701, 2019.

SARMA, A.; REDMILES, D. F.; HOEK, A. V. D. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, IEEE, v. 38, n. 4, p. 889–908, 2012.

SEO, H.; SADOWSKI, C.; ELBAUM, S.; AFTANDILIAN, E.; BOWDIDGE, R. Programmers' build errors: a case study (at Google). In: *International Conference on Software Engineering*. [S.l.]: ACM, 2014.

SHAMSHIRI, S.; FRASER, G.; MCMINN, P.; ORSO, A. Search-based propagation of regression faults in automated regression testing. In: IEEE. *International Conference on Software Testing, Verification and Validation*. [S.l.], 2013.

SHAO, D.; KHURSHID, S.; PERRY, D. E. Evaluation of semantic interference detection in parallel changes: an exploratory experiment. In: IEEE. *International Conference on Software Maintenance*. [S.l.], 2007.

SHEN, B.; ZHANG, W.; ZHAO, H.; LIANG, G.; JIN, Z.; WANG, Q. Intellimerge: A refactoring-aware software merging technique. *ACM Transactions on Programming Languages and Systems*, Association for Computing Machinery, v. 3, n. OOPSLA, 2019.

SILVA, I. P.; ALVES, E. L.; ANDRADE, W. L. Analyzing automatic test generation tools for refactoring validation. In: IEEE. *International Workshop on Automation of Software Testing*. [S.l.], 2017.

SILVA, L. D.; BORBA, P.; MAHMOOD, W.; BERGER, T.; MOISAKIS, J. Detecting semantic conflicts via automated behavior change detection. In: IEEE. [S.l.], 2020. (International Conference on Software Maintenance and Evolution), p. 174–184.

SILVA, L. D.; BORBA, P.; PIRES, A. Build conflicts in the wild. *Journal of Software: Evolution and Process*, Wiley Online Library, p. e2441, 2022.

SILVA, L. M. P. D. *Build and Test Conflicts in The Wild*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, 2018.

SOUSA, M.; DILLIG, I.; LAHIRI, S. K. Verified three-way program merge. *ACM Transactions on Programming Languages and Systems*, ACM, v. 2, n. OOPSLA, p. 1–29, 2018.

SOUZA, C. R. B. de; REDMILES, D.; DOURISH, P. Breaking the code, moving between private and public work in collaborative software development. In: *International ACM SIGGROUP Conference on Supporting Group Work*. [S.l.]: ACM, 2003.

SUBVERSION. 2020. Accessed: October 2020. Disponível em: <https://subversion.apache.org/>.

SUNG, C.; LAHIRI, S. K.; KAUFMAN, M.; CHOUDHURY, P.; WANG, C. 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*. [S.l.: s.n.], 2020.

TAVARES, A. T.; BORBA, P.; CAVALCANTI, G.; SOARES, S. Semistructured merge in JavaScript systems. In: IEEE. *International Conference on Automated Software Engineering*. [S.l.], 2019.

TICHY, W. F. Tools for software configuration management. In: *Proceeding of the International Workshop on Software Version and Configuration Control*. [S.l.: s.n.], 1988.

TIWARI, D.; ZHANG, L.; MONPERRUS, M.; BAUDRY, B. Production monitoring to improve test suites. *arXiv preprint arXiv:2012.01198*, 2020.

VASILESCU, B.; SCHUYLENBURG, S. V.; WULMS, J.; SEREBRENIK, A.; BRAND, M. G. van den. Continuous integration in a social-coding world: Empirical evidence from github. In: IEEE. *International Conference on Software Maintenance and Evolution*. [S.l.], 2014.

VASILESCU, B.; YU, Y.; WANG, H.; DEVANBU, P.; FILKOV, V. Quality and productivity outcomes relating to continuous integration in github. In: ACM. [S.l.], 2015. (Joint Meeting on Foundations of Software Engineering), p. 805–816.

VASSALLO, C.; PROKSCH, S.; ZEMP, T.; GALL, H. C. Every build you break: Developer-oriented assistance for build failure resolution. *Empirical Software Engineering*, Springer, p. 1–40, 2019.

VASSALLO, C.; SCHERMANN, G.; ZAMPETTI, F.; ROMANO, D.; LEITNER, P.; ZAIDMAN, A.; PENTA, M. D.; PANICHELLA, S. A tale of CI build failures: An open source and a financial organization perspective. In: IEEE. *International Conference on Software Maintenance and Evolution*. [S.l.], 2017.

VEEN, E. V. D.; GOUSIOS, G.; ZAIDMAN, A. Automatically prioritizing pull requests. In: IEEE. [S.l.], 2015. (Mining Software Repositories), p. 357–361.

WUENSCHE, T.; ANDRZEJAK, A.; SCHWEDES, S. Detecting higher-order merge conflicts in large software projects. In: IEEE. *International Conference on Software Testing, Validation and Verification*. [S.l.], 2020.

XU, S.; DONG, Z.; MENG, N. Meditor: Inference and application of api migration edits. In: IEEE. [S.l.], 2019. (International Conference on Program Comprehension), p. 335–346.

ZAMPETTI, F.; BAVOTA, G.; CANFORA, G.; PENTA, M. D. A study on the interplay between pull request review and continuous integration builds. In: IEEE. [S.l.], 2019. (International Conference on Software Analysis, Evolution and Reengineering), p. 38–48.

ZAMPETTI, F.; VASSALLO, C.; PANICHELLA, S.; CANFORA, G.; GALL, H.; PENTA, M. D. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, Springer, v. 25, n. 2, p. 1095–1135, 2020.

ZHAO, Y.; SEREBRENIK, A.; ZHOU, Y.; FILKOV, V.; VASILESCU, B. The impact of continuous integration on other software development practices: A large-scale empirical study. In: *International Conference on Automated Software Engineering*. [S.l.]: IEEE, 2017.

ZIMMERMANN, T. Mining workspace updates in cvs. In: *International Conference on Mining Software Repositories*. [S.l.]: IEEE, 2007.

ZIMMERMANN, T. Mining workspace updates in cvs. In: IEEE. *International Workshop on Mining Software Repositories*. [S.l.], 2007.