



Pós-Graduação em Ciência da Computação

LÉUSON MÁRIO PEDRO DA SILVA

BUILD AND TEST CONFLICTS IN THE WILD



Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE

2018

Léuson Mário Pedro da Silva

BUILD AND TEST CONFLICTS IN THE WILD

A M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: Paulo Henrique Monteiro Borba

RECIFE

2018

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

S586b Silva, Léuson Mário Pedro da
Build and test conflicts in the wild / Léuson Mário Pedro da Silva. – 2018.
111 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn,
Ciência da Computação, Recife, 2018.
Inclui referências e apêndices.

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

005.1 CDD (23. ed.) UFPE- MEI 2018-059

Léuson Mário Pedro da Silva

Build and Test Conflicts in the Wild

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 19/02/2018

BANCA EXAMINADORA

Prof. Dr. Marcelo Bezerra d'Amorim
Centro de Informática / UFPE

Prof. Maurício Finavaro Aniche
Department of Software Technology

Prof. Dr. Paulo Henrique Monteiro Borba
Centro de Informática
(Orientador)

*I dedicate this thesis to all my family, friends and professors
who gave me the necessary support to get here.*

ACKNOWLEDGEMENTS

Agradeço a todos que contribuíram diretamente ou indiretamente para a realização deste trabalho.

Primeiramente ao meu orientador, professor Paulo Borba, pela sua orientação e dedicação para a realização deste trabalho. Mais do que simplesmente uma orientação, o senhor tem me mostrado a buscar a excelência que um pesquisador necessita ter.

Gostaria de agradecer aos membros do SPG pelos conselhos e amizade. Agradeço também aos demais integrantes do LabES pelos momentos de descontração. Agradeço ao CIn/UFPE e aos seus funcionários pela estrutura e recursos fornecidos, assim como ao INES — Instituto Nacional de Ciência e Tecnologia para Engenharia de Software.

Agradeço também aos professores Marcelo d’Amorim e Maurício Aniche por terem aceito nosso convite para avaliar esta dissertação.

A meus pais, Aparecida e Lourival, e irmãos, Leidemara, Luzivânia e Lourismar, por toda a compreensão nos momentos em que estive ausente, e por todo o suporte e apoio que venho recebendo ao longo de toda a minha vida acadêmica.

Gostaria de agradecer aos meus amigos que perto ou longe sempre estiveram me motivando. Em especial minha colega de orientação Klissiomara, e Neto, pelas conversas e momentos de descontração. Aos meus amigos João Carlos e Beatriz pela amizade e companheirismo que só se fortalece ao longo dos anos.

Além desses, também agradeço à Capes por financiar a minha pesquisa.

Por último, não menos importante, agradeço à Deus, por tudo que ele me concedeu e por todos os sonhos que me permite sonhar.

*Daher ist die Aufgabe nicht sowohl zu sehen was noch keiner gesehen hat,
als bei Dem was Jeder sieht, zu denken was noch Keiner gedacht hat.*

—ARTHUR SCHOPENHAUER

ABSTRACT

Collaborative software development allows developers to simultaneously contribute to the same project performing different activities. Although this might increase development productivity, it also brings conflicts among developers contributions. Conflicts may arise in different development phases: during merging, when different contributions are integrated (merge conflicts); after integration, when building the integration results fail (build conflicts); or when testing, unexpected software behavior happens (test conflicts). To understand how different contributions from merge scenarios influence build and test conflicts occurrence, in this thesis we investigate the frequency, causes and adopted resolution patterns for these conflicts. We perform an empirical study evaluating merge scenarios from Java projects that use Travis CI for continuous integration and Maven as build manager. To identify conflicts, we access information from git repositories of the projects and their associated build process. Filters were applied to select merge commits that present unsuccessful build processes (caused by problems during the build creation or test execution), while their parent commits have successful build process. Besides parsing build logs for identifying the causes behind the broken builds, we also parse the source code to establish interference between contributions. Different from previous studies, we also evaluate scenarios that caused merge conflicts fixed by integrators, leading us to classify our results based on contributor and integrator changes to fix the conflicts. Although the number of build conflicts caused by contributor changes is high (66,4%), we have evidence that build conflicts can also be caused by changes performed after the integration by integrators (33,6%). The most recurrent causes of build conflicts concentrate on static semantic problems (80%), especially *Unavailable Symbol* (52,8% of all build conflicts that represents the attempt to use a symbol no longer available in the project). Test conflicts are caused by failed test cases, that are not restricted to old tests but also new or updated during the merge scenario. Additional analysis performed after tests execution can also cause test conflicts. Related to the adopted resolution patterns, build and test conflicts were fixed without returning to an old project version. Typically, parent commits authors are also responsible for fixing build and test conflicts. Our findings bring the evidence of build and test conflicts showing they occur independent of merge conflict occurrence during integration. For build conflicts, we define a catalog of causes that can be applied to assistive tools on software development aiming to avoid or treat such problems. For test conflicts, our scripts can be used to develop an assistive tool to developers when trying to understand the test case failure since we filter the parent's contributions informing only those parts involved in the test failure.

Keywords: Collaborative Development. Build failures. Build conflicts. Test conflicts. Conflict-ing contributions.

RESUMO

Desenvolvimento colaborativo de software permite que desenvolvedores contribuam simultaneamente para um mesmo projeto realizando diferentes atividades. Embora esta abordagem possa aumentar a produtividade de desenvolvimento, ela também traz consigo conflitos entre contribuições de desenvolvedores. Conflitos podem surgir em diferentes momentos. Durante cenários de merge, quando contribuições diferentes são integradas (conflitos de merge), como também após a integração, durante a tentativa de realizar o processo de build, quando este processo falha devido o resultado da integração falha (conflitos de build), ou mudança no comportamento do software (conflitos de teste). Para entender como contribuições diferentes de cenários de merge influenciam em conflitos de build e teste, nesta dissertação nós investigamos a frequência, causas e padrões de resolução adotados para resolver estes conflitos. Para tanto, nós realizamos um estudo empírico avaliando cenários de merge de projetos Java usando Travis (para integração contínua) e Maven (como gerenciador de build). Para identificar conflitos, nós acessamos as informações de repositórios git como também seus processos de build associados. Sucessivos filtros foram aplicados para selecionar *commits* de merge que apresentavam processos de build mal-sucedidos (causadas por problemas durante a criação da *build* ou execução dos testes), enquanto seus *commits* pais apresentavam processos bem-sucedidos. Além de analisar os logs das builds para extrair as causas das quebras, nós também avaliamos o código-fonte para identificar as interferências entre contribuições. Diferente de estudos anteriores, nós também avaliamos a ocorrência de conflitos em cenários oriundos de conflitos de merge levando-nos a classificar nossos resultados baseados nas mudanças dos contribuidores e integradores. Embora o número de conflitos de build causados por mudanças dos contribuidores seja alto (66,4%), nós temos evidência que conflitos de build também podem ser causados por mudanças feitas após a integração por integradores (33,6%). As causas mais recorrentes de conflitos de build concentram-se em problemas estáticos semânticos, especialmente *Símbolo Indisponível* (52,8% de todos os conflitos de build representando a tentativa de usar um símbolo indisponível no projeto). Conflitos de teste são causados por casos de teste falhos, que não são restritos a casos de teste antigos como também novos ou atualizados durante o cenário de merge. Análises adicionais realizadas após a execução de testes também causar conflitos de teste. Em relação aos padrões de resolução de conflitos, a solução mais utilizada consiste em adaptar as causas dos conflitos ao novo estado do projeto, ao invés de retornar para um estado antigo. Tipicamente, desenvolvedores autores pelos *commits* pais, também são responsáveis por resolver os conflitos. Nossas descobertas trazem a evidência de conflitos de build e teste mostrando que estes ocorrem independente de conflitos de merge durante a integração. Em relação aos conflitos de build, nós identificamos um catálogo de causas que pode ser aplicado em ferramentas de suporte ao desenvolvimento de software evitando ou tratando estes problemas. Para conflitos de teste, nossos scripts podem apoiar o trabalho de desenvolvedores durante a tentativa de entender a falha de um caso de teste, uma vez que nós filtramos as contribuições dos *commits* pais informando

apenas aquelas envolvidas na falha do teste.

Palavras-chave: Desenvolvimento colaborativo. Falhas de build. Conflitos de build. Conflitos de teste. Contribuições conflitantes.

LIST OF FIGURES

2.1	Centralised version control paradigm	20
2.2	Decentralised version control paradigm	20
2.3	Commits history of a project. Each commit presents an arrows indicating its parent(s)	22
2.4	Merge scenario without merge conflict	22
2.5	Merge scenario with merge conflict	23
2.6	Merge scenario with build conflict	24
2.7	Compilation problem on build process after merge scenario	25
2.8	Merge scenario with dynamic semantic conflict	26
2.9	Failed test case after merge scenario	26
2.10	Build and test conflicts in practice	27
2.11	Log of errored build due to compilation problem	27
2.12	Merge scenario with false positive merge conflict	29
2.13	Integrator changes introducing inconsistencies on source code	29
2.14	Merge scenario without merge conflicts reported by improved merge tools	30
2.15	Build process life cycle on Travis CI [1]	33
3.1	Study design	37
3.2	Merge scenarios filters adopted during the analysis	39
3.3	Different build status for the same commit. X indicates an errored build, while ! indicates a failed one.	40
3.4	Checking contributions performed on merge scenarios parents	43
3.5	Build log of errored process	46
3.6	Original version of GumTree diff for conflicting contributions	46
3.7	Build log of errored process due to contributor changes	47
3.8	Original version of GumTree diff for integrator changes	47
3.9	Build log of errored process due to missing dependency	48
3.10	Failed test case on failed build process	49
3.11	Test conflicts analysis	51
3.12	Build log of failed process due to unachieved metric	52
3.13	Commits information to identify commit fixes	53
3.14	Build log of errored process due to unavailable symbol	54
3.15	GumTree diff for commit fix	54
4.1	Improved messages for different unavailable symbol types	71
4.2	Prototype of assistive tool for test failures	72
A.1	Build log of broken build due to malformed program	90

A.2	Build log of broken build due to unimplemented method	91
A.3	Build log of broken build due to incompatible method signature	92
A.4	Build log of broken build due to duplicated declaration	93
A.5	Build log of broken build due to incompatible types	93
A.6	Build log of broken build due to unfollowed project guideline	94
C.1	Travis.yml file example for running failed test case	98

LIST OF TABLES

3.1	Build error messages taxonomy and how they related to conflicts or the lack of conflicts?	42
3.2	Unavailable Symbol information from Travis Log	45
3.3	Covered and Changed Classes involved in Test Conflict	51
4.1	Broken builds caused by build and test conflicts.	57
4.2	Causes of build and test conflicts	59
4.3	Distribution of build conflict causes by category and motivation	60
4.4	Distribution of test conflict causes by category	63
4.5	Resolution patterns for build and test conflicts	65
D.1	Sample	100
E.1	Build conflicts from errored builds	108
E.2	Build conflicts from failed builds	110
E.3	Test conflicts from failed builds	111

CONTENTS

1	INTRODUCTION	15
2	BACKGROUND	19
2.1	Version Control Systems	19
2.1.1	Evolution and Integration in VCS	21
2.1.2	Conflict Types	21
2.1.2.1	<i>Merge Conflicts</i>	23
2.1.2.2	<i>Build Conflicts</i>	24
2.1.2.3	<i>Test Conflicts</i>	25
2.1.2.4	<i>Build and Test Conflicts in Practice</i>	25
2.1.2.5	<i>Build and Test Conflicts by Integrator Changes</i>	28
2.2	Continuous Integration (CI)	28
2.2.1	CI Life Cycle	30
2.2.1.1	<i>Build and Compilation</i>	30
2.2.1.2	<i>Automated Static Analysis</i>	31
2.2.1.3	<i>Test Execution</i>	32
2.2.2	Travis CI	32
3	IDENTIFYING CONFLICTING CONTRIBUTIONS IN MERGE SCENARIOS	35
3.1	Problem Statement	36
3.2	Study Design	36
3.2.1	Mining Repositories	37
3.2.1.1	<i>Sample</i>	37
3.2.1.2	<i>Filtering Sample Projects</i>	38
3.3	Conflicts Identification	40
3.4	Classifying Conflicts	44
3.4.1	Classifying Conflicting Contributions on Build Conflicts	44
3.4.2	Classifying Conflicting Contributions on Test Conflicts	48
3.5	Resolution Patterns Identification	52
3.5.1	Resolution Patterns for Build Conflicts	53
3.5.2	Resolution Patterns for Test Conflicts	54
4	RESULTS	56
4.1	Research Questions	56
4.1.1	RQ1: How frequently do build and test conflicts occur?	56
4.1.2	RQ2: What are the structure of the changes that cause build and test conflicts?	58
4.1.3	RQ3: What are the resolution patterns adopted on build and test conflicts fixes?	64
4.2	Discussion	66
4.2.1	Conflicts are recurrent	66

4.2.2	Findings and Implications	68
4.2.2.1	<i>Awareness Tools</i>	68
4.2.2.2	<i>Automatic Repair Tools</i>	69
4.2.2.3	<i>Better Guidelines Support for Developers</i>	71
4.2.2.4	<i>Better Merge Tools</i>	72
4.3	Threats to Validity	73
4.3.1	Construct Validity	73
4.3.2	Internal Validity	74
4.3.3	External Validity	77
5	CONCLUSION	78
5.1	Contributions	80
5.2	Related Work	80
5.2.1	Empirical Studies	80
5.2.2	Build Errors Diagnosis	82
5.3	Future Work	83
	REFERENCES	85
	APPENDIX	89
	APPENDIX A - Build Conflicts Identification	90
	APPENDIX B - Build Conflict Fixes	95
	APPENDIX C - Travis Configuration File Instrumentation	97
	APPENDIX D - Study Sample	99
	APPENDIX E - Build and Test Conflicts	107

1 INTRODUCTION

During collaborative software development, developers often contribute without being aware of other team members. Each contribution is performed in private copies of a project supported by Version Control Systems (VCS). Eventually, these individual contributions (parents, composed by one or more commits) are integrated with each other aiming to keep the main development line stable and accessible for all involved (**merge scenario**). Although this might increase productivity, the integration process may lead to conflicts among developer's contributions, requiring time and human intervention to fix them.

Conflicts may arise in different development phases: during merging, when different contributions are integrated (merge conflicts), as also after a merge scenario [2]. For example, **build conflicts** happen when the source code result of a merge scenario cannot be compiled and built during a build process even though the individual contributions of the merge scenario present a successful compilation and build process. In the same way, **test conflicts** occur when after the integration, some of the test cases associated to the project present failed status. However, these failed test cases were passed before the merge scenario. These conflicts directly impact developers' productivity since understanding and solving them is often a demanding and error-prone activity [3] [4]. Dealing with build and test conflicts is even worse since developers must be aware of the interaction of their contributions with the work of other team members. The difficulty is not restricted to understanding the conflict causes but also how to fix it, especially for test conflicts since any change can impact the software behavior.

Although most reported evidence about the occurrence of conflicts in software development relies on merge conflicts [5] [6] [7] [4], there are studies investigating the frequency and limited characteristics of build and test conflicts [8] [2]. Despite such evidence, it is important to better understand frequency and the causes behind these conflicts. Once the causes responsible for conflicts are identified, the circumstances associated with the causes would also arise allowing the development of approaches to avoid these conflicts. In the same way, the investigation of how conflicts are fixed would bring information of adopted resolution patterns. This information might help to improve assistive tools aiming to treat such problems. In this way, this thesis focuses on verifying the frequency, causes and resolution patterns adopted for fixes of build and test conflicts.

With that aim, we perform an empirical study evaluating the occurrence rate of build and test conflicts in 60991 merge scenarios from 529 Java projects. In the end we identify more than

150 conflicts in more than 120 merge scenarios. Different from related work [5] [9], we consider merge scenarios as the triple formed by one merge commit and its Left and Right parents. For each merge commit, we verify whether its build presents unsuccessful status, while its parents have superior ones. For example, for builds that present *errored* status, we consider *failed* and *passed* as superior status. For builds with *failed* status, only the *passed* status is a valid superior status. In case one of the parents has the same status of the merge commit, we assume the broken build of the merge commit is carried over from the broken parent. So there is not a conflict between parent commits, but simply a defect in one of the parent contributions, which ended up reaching the merge commit.

Previous work perform their studies adopting approaches that could bias the results. For example, building merge commits locally can introduce bias since any difference in environment configuration could break the build process leading the authors to a wrong conclusion. To eliminate these threats, we adopt a new perspective for getting information about build process from merge scenarios. Instead of (re)build all commits, we consider only projects that use Continuous Integration (CI) [10] to obtain the needed information from CI logs. This practice is responsible for automatically building and running tests or other analysis to a project, helping to ensure the quality of contributions [11] and to increase developers productivity [12] [13] [14]. In this context, [Travis CI](#) (one of the most used CI service [15] [16]) provides such information (build status and associated logs) through its [API](#).

As build and test conflicts arise from unsuccessful build process (broken builds), we cross-check source code changes and build process information to ensure whether a merge commit associated with a broken build corresponds to a build or test conflict. In this way, we investigate how the merge commit parent contributions conflict with each other (causes), and, when applicable, the resolution patterns adopted for fixing the conflicts. In summary, we investigate the research questions motivated and presented bellow.

Some studies investigate build and test conflicts during software development. Brun et al. [8] present 33% of all analyzed scenarios represents cases of build or test conflicts. Kasi and Sarma [2] present build and test conflicts range from 2%-15% and 6%-35%, respectively. Despite such evidence, it is important more substantial characteristics about these conflicts since the previous studies have threats, which could bias the results. Therefore, our first research question is:

- **(RQ1) - Frequency:** How frequently do build and test conflicts occur?

To answer RQ1, we assess merge commit status information from Travis. We do the same for the merge commit parents. Nevertheless, as the information associated with some of these commits might not be available in Travis because they were not automatically built, we force the creation of by forking the original project. Exclusively for test conflicts, we use the information of the Travis log associated with the failed build to replicate each failed test case

with a new build on Travis. We do this to ensure the failure occurrence is motivated by the parent contributions (some test failures are caused by external or environment restrictions).

Despite the evidence about build and test conflicts, there is a lack of information about the causes responsible for them. Seo et al. [17] present the technical causes responsible for broken builds. Nevertheless, the results are not specific for build and test conflicts. A deeper analysis focused on the circumstances conflicts happen could bring the causes, and consequently, insights on how to deal with them. Hence, our second research question is:

- **(RQ2) - Causes:** What are the structures of the changes that cause build and test conflicts?

The second question investigates the causes behind the conflicts. For build conflicts, based on the error messages reported by Travis log, we perform further analysis on source code parent contributions classifying and categorizing conflicts causes. For example, if a build has errored status because of a missing method reference, we verify whether one parent commit removes the referenced method, while the other adds a new reference for it. Depending on build error cause, we also do further analysis on changes performed by the integrator during the merge scenario integration. For example, if a missing referenced method is not removed by one of the parents, we verify whether the integrator is responsible for the removal. For tests conflicts, we generate the code coverage for each test execution during its replication on Travis. We compare this coverage with the parent contributions looking for dependencies among them. If a dependency is identified, we assume the integration of changes causes the test failure, once the changes isolated do not impact on unexpected software behavior (test failure).

Dealing with conflicts is a demanding and boring task [18] [4], and the way adopted to fix conflicts is particular for each integrator. For example, in merge conflicts, integrators normally tend to keep their contributions instead of accepting others [19]. To know how build and test conflicts are fixed can bring insights of improvements for assistive tools. In the same way, knowing who does the fixes can inform how difficult is the process to fix build and test conflicts. For example, merge conflicts do not require knowledge about the contributions intent since they involve only text. In the other side, build and test conflicts involve additional effort once semantic aspects must be taken in account. Thus, our third research question is:

- **(RQ3) - Resolution Patterns:** What are the resolution patterns adopted to fix build and test conflicts?

- **(RQ3.1) - Fixer:** Who does fix build and test conflicts?

Finally, to answer RQ3, we identify the closest commit following the merge responsible for fixing the conflict. For that fix commit, we check its build status on Travis, when possible, to ensure it is a true fix. Thus, we could verify the changes patterns adopted on each conflict type as also metrics related to who fixed them.

Besides answering our research questions about the evidence and characteristics of build and test conflicts, we propose applications that our results can be applied. Additionally, for doing our experiment, we improved GumTree tool aiming to achieve a better and more informative syntactic diff. All of our scripts are available in our on-line Appendix [\[20\]](#) to the community encouraging the replication of this study. The remainder of this study is organized as follows:

- Chapter 2 reviews the main concepts used in this study. It includes Version Control Systems and Continuous Integration principles;
- Chapter 3 motivates our study and presents in detail the research questions. It also explains how our study is conducted, its mining step, subjects selection, and evaluation strategy;
- Chapter 4 presents our results and answers to our research questions. We also present the discussion and the threats to the validity of our study;
- Chapter 5 details our conclusions and discusses related and future work.

2 BACKGROUND

In this chapter, we present the main concepts used in this study. Initially, in Section 2.1, the concepts related to Version Control Systems (VCS) are presented focusing on the types of conflicts, and how they arise during software development (Section 2.1.2). In Section 2.2, we describe the use and history of Continuous Integration (CI), and how a build process works. The particularities of Travis CI important to this thesis are detailed in Section 2.2.2.

2.1 Version Control Systems

Collaborative software development is only possible due to the adoption of Software Configuration Management (SCM), and consequently, the use of VCSs. SCMs are responsible for all software artifacts that can evolve over time. In this way, SCMs provide development tools and techniques able to deal with the evolution of artifacts, especially when this evolution includes the parallel work of several developers [21]. VCSs are a particular approach responsible for dealing with evolution on software artifacts allowing changes and updates, as also ensuring the share and management of information among developers.

Different paradigms of VCSs can be used for software development differing each other on the way information is accessed and delivered to developers. For example, in the Centralized Version Control Systems (CVCS) paradigm, there is a central repository working as the unique server holding the whole information of the project and getting all associated changes history (Figure 2.1). Thus, at the beginning of each task, a developer updates its particular repository from the central one to start doing its contributions. After completing the task, the contributions are integrated into the central repository (synchronization process) aiming to keep the current development stable and accessible for all involved. Many services implement this approach, including Subversion [22] and CVS [23].

In the Decentralized Version Control Systems (DVCS), as the name suggests, there is not a central repository shared with all developers. Every particular repository (project copy) can be the source of information for any another contributor holding, individually, its particular changes history (Figure 2.2). Git [24] and Mercurial [25] are examples of highly used VCSs that follow this paradigm. In this way, contributions performed by a developer can be accessed and shared by others directly without the intermediation of a secondary repository (direct communication). This approach has become popular in software development, especially in the open-source community,

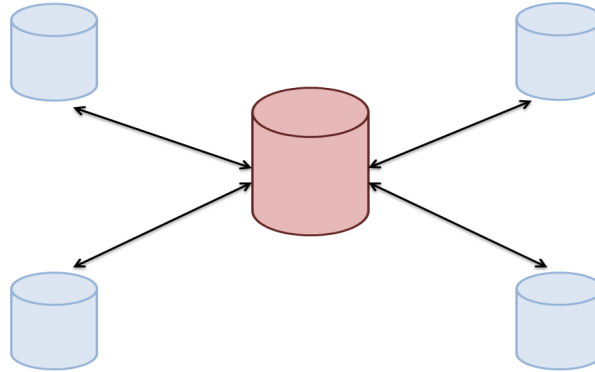


Figure 2.1: Centralised version control paradigm

The arrows among the individual repositories and the central one indicate only the central repository can accept changes from the others. 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.

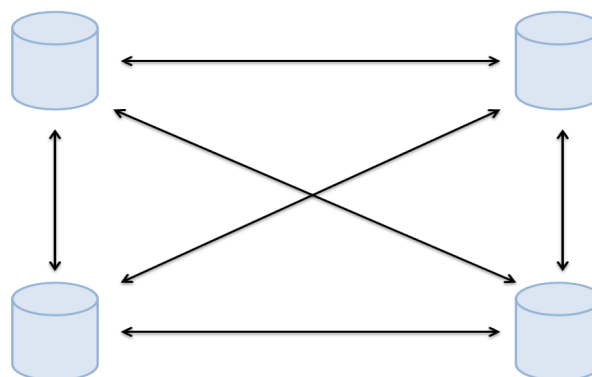


Figure 2.2: Decentralised version control paradigm

The arrows among the repositories indicate every repository can accept changes and update themselves from the others. Individual repositories can be accessed directly without any intermediate repository.

because of its simplicity in how information can be propagated in many directions and directly among developers [26] [27]. Many services implement this idea, including GitHub, the biggest service available online holding 67 million repositories.

In both paradigms, the evolution of software artifacts is achieved by integration of different contributions over a project life cycle. During such integration, conflicts (merge, build and test conflicts) may arise impacting directly on team productivity. The investigation of these problems might identify improvements for assistive tools. For example, new causes of build conflicts can reflect in new features for tools like Palantír proposed by Sarma et al. [28]. In the same way, merge tools can benefit of these findings bringing insights of how to treat build and test conflict during contributions integration. For example, improved merge tools [7] [29] can already identify duplicated method declarations in a class, which would break the build process.

2.1.1 Evolution and Integration in VCS

Before presenting conflict types, it is important to detail some concepts related to the contribution integration process. Such process involves merge scenarios, which we consider as a triple formed by two parent commits (Left and Right) integrated into a merge commit. These parent commits evolve based on a common base ancestor commit (the point where the contributions start to diverge with each other). For instance, as presented in Figure 2.3, the *commit* *C5* is a merge commit, while the pair of *commits* [*C3*, *C4*] are its parents (the bottom in Figure 2.3). *Commit* *C4* is also a merge commit having as its parents the pair [*C1*, *C2*] (the bottom in Figure 2.3). The difference between the *commits* *C4* and *C5* is perceived when we look for their *base commits*. *Commit* *C4* has as its base the *commit* *C1*, which is also one of its parent commits. Scenarios with this property are known as *fast-forward* (only one of the parents evolves) [30]. For *commit* *C5*, its base commit is also *C1*. However, none of the parents is the same to the base commit revealing this case as a valid merge scenario for us (both parents have evolved). In this thesis, we discard *fast-forwards* commits.

We present an example of a merge scenario in Figure 2.4. In this example, *Left* adds a new attribute (*classification*) for the class *Movie*, while *Right* introduces a new method declaration (*isNewRelease*). Although the contributions are performed on the same file, no problem (conflicts) happens since the contributions are not performed in the same area. As result, all parents contributions are preserved characterizing this case as a **clean merge scenario**.

2.1.2 Conflict Types

Conflicts may arise in different development phases: during merging, when different contributions are integrated (merge conflicts); after integration, when building the integration result fails (build conflicts); or when testing, unexpected software behavior happens (test conflicts) [2]. In the next sections, we explain the nature of these conflicts (merge, build and test conflicts), and how the occurrence of one type can impact in others.

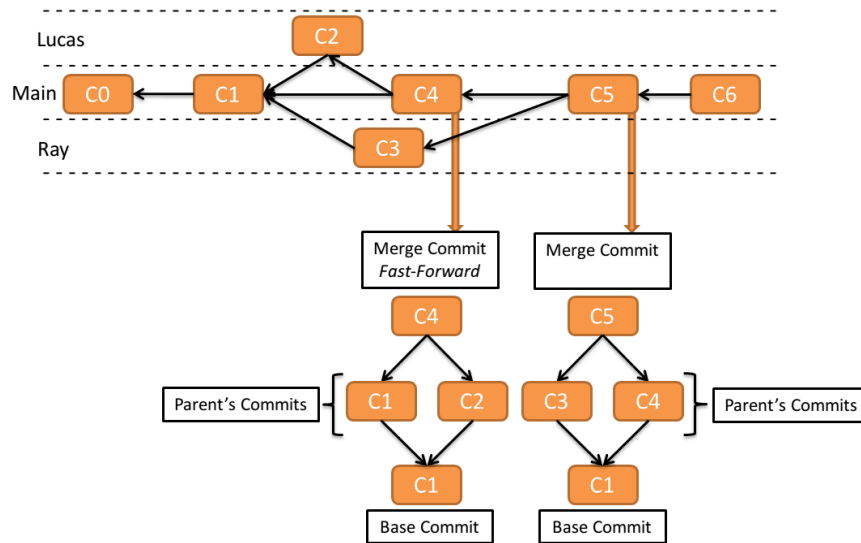


Figure 2.3: Commits history of a project. Each commit presents an arrows indicating its parent(s)

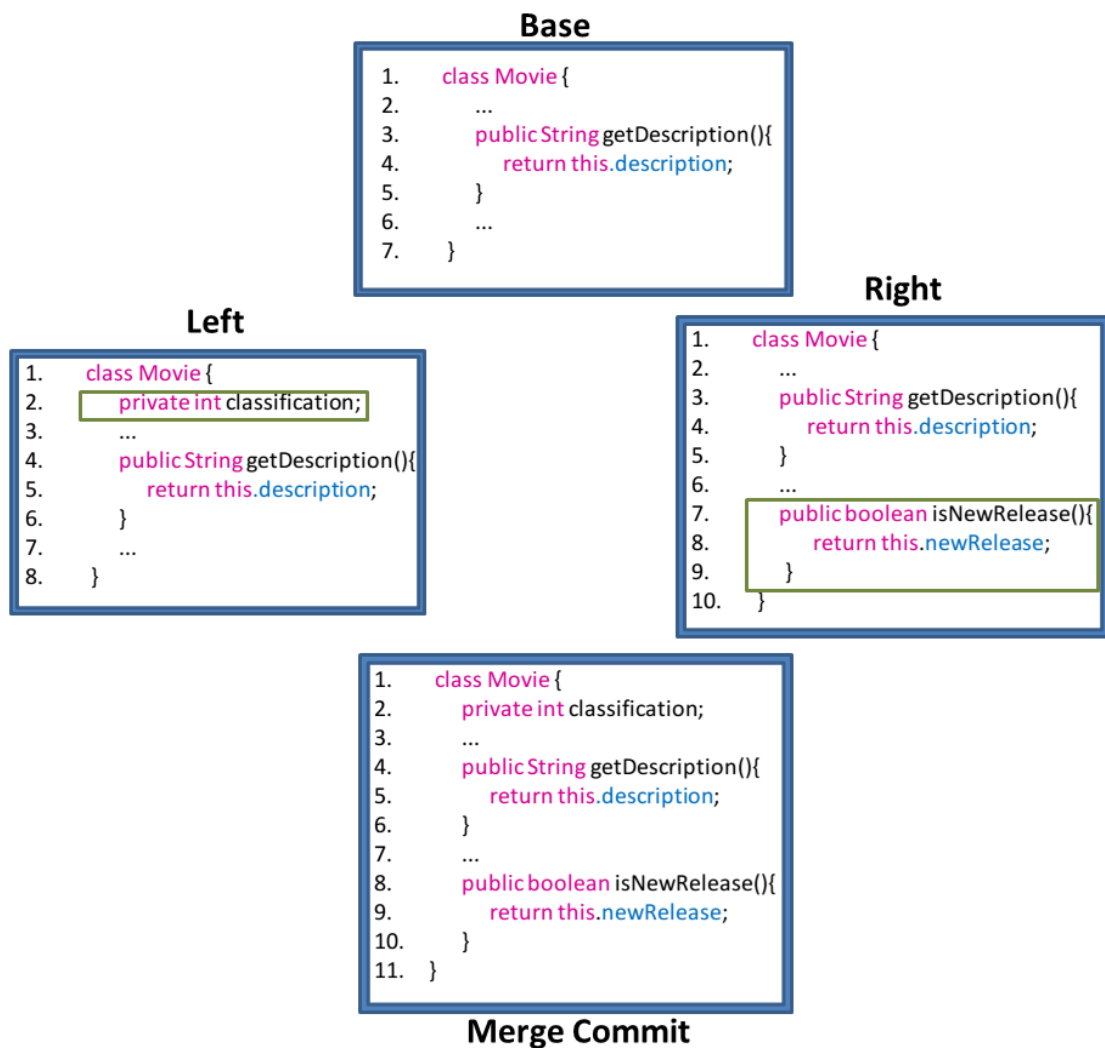


Figure 2.4: Merge scenario without merge conflict

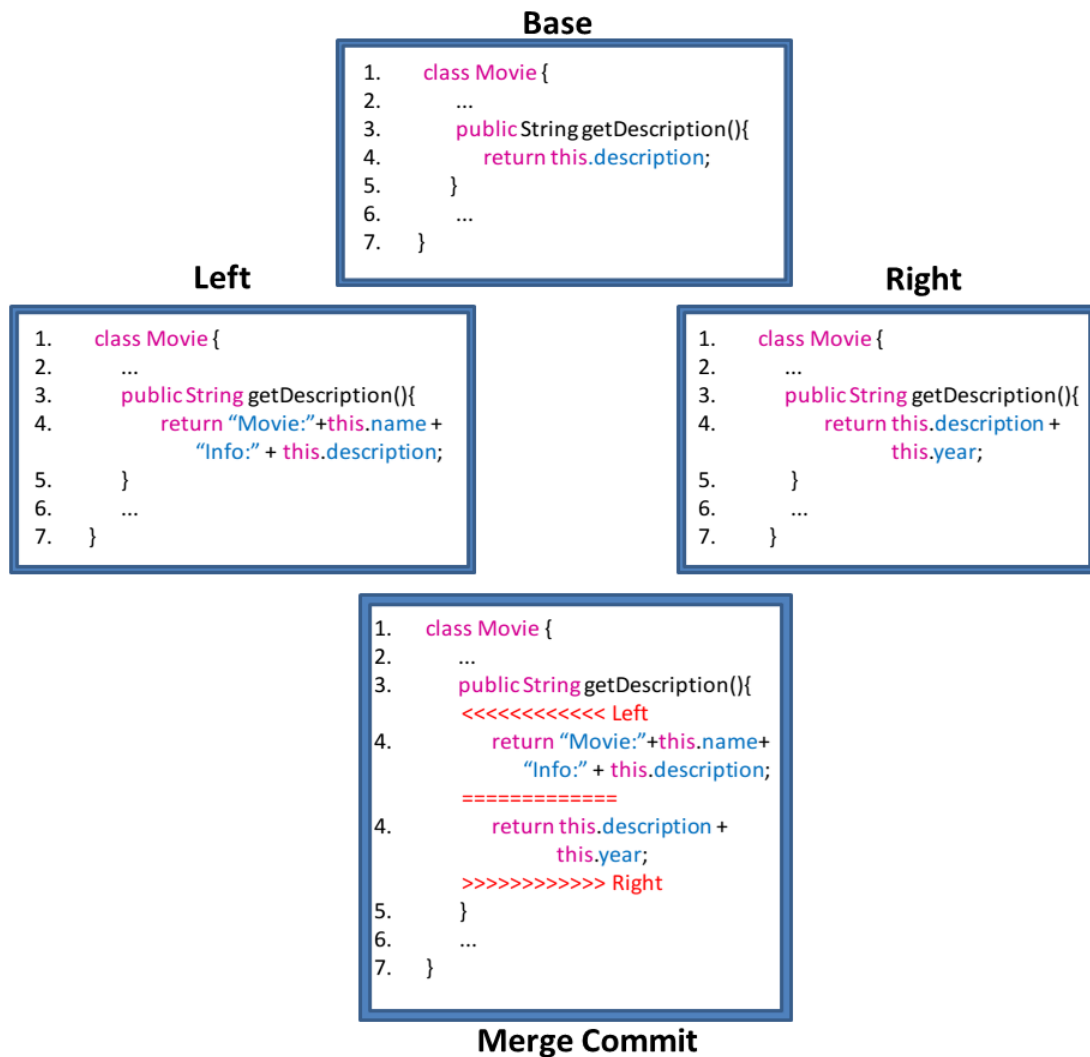


Figure 2.5: Merge scenario with merge conflict

2.1.2.1 Merge Conflicts

Merge conflicts occur when the text of the contributions to be integrated contain changes to the same areas. We present an example of a merge scenario in Figure 2.5. In this situation, the method *getDescription* is declared in the *Base* commit. During parent's contributions, both *Left* and *Right* commit update the return statement of the method. Since the changes are performed in the same area, specifically in line 4, a conflict is reported (presented in *Merge Commit*).

As any merge conflict, human intervention is necessary to fix the conflict. Nevertheless, when trying to solve merge conflicts, integrators might introduce inconsistencies in the source code. For example, the way adopted to solve a merge conflict cannot preserve all parent contributions, which would lead to other problems perceived only in next steps of the integration process. We now present examples of those kinds of conflicts.

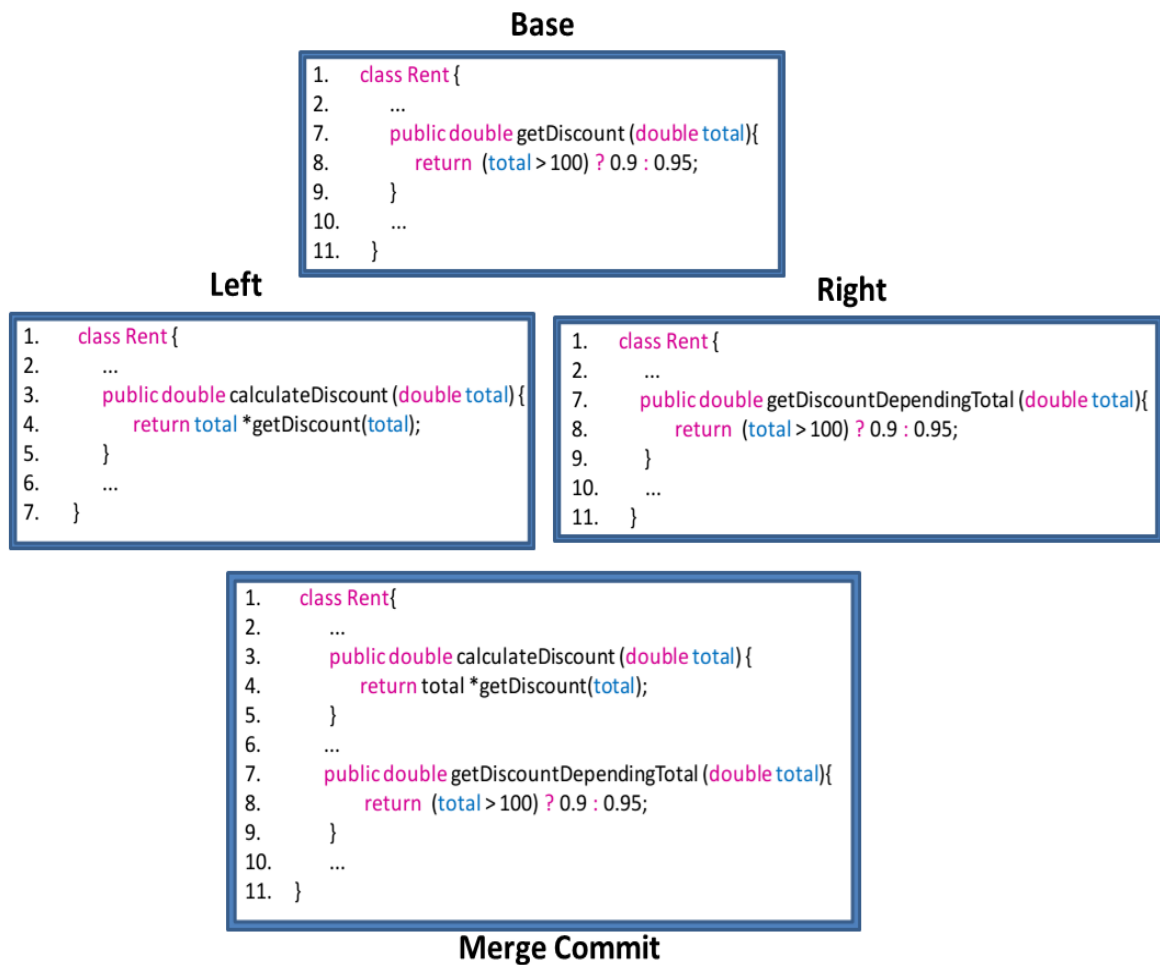


Figure 2.6: Merge scenario with build conflict

2.1.2.2 Build Conflicts

Build conflicts occur when contributions are successfully merged but they are incompatible in the sense that keeping both contributions leads to a broken build process [28]. We present an example of build conflict in Figure 2.6.

In this case, *Left* contribution adds a new method responsible for calculating the discount for a rent (lines 3-6). The discount to be applied depends on the final amount that is received as a parameter (*total*). The method responsible for informing the discount is *getDiscount* (present in *Base* commit and not modified by the *Left*). Simultaneously, *Right* renames the method leading to the new, more intuitive, *getDiscountDependingTotal* (line 7). Consequently, all calls are updated for using the new method signature.

During the merge integration, no conflicts happen but the associated build process fails because a method reference could not be satisfied (Figure 2.7). The problem happens because *Left* tries to reference a method with a signature not available, and despite the call updates performed by *Right*, the new reference introduced by *Left* still uses the old signature (*getDiscount*). In this scenario, it is clear how a developer impacts the work of other with parallel related tasks.

```
Compilation Failure:  
[ERROR] /home/.../Rent.java:[] cannot find symbol  
[ERROR] symbol: method getDiscount(java.lang.double)  
[ERROR] location: class Rent
```

Figure 2.7: Compilation problem on build process after merge scenario

2.1.2.3 Test Conflicts

Tests conflicts occur when contributions are merged and a build is successfully created, but tests that were successfully executed with one of the contributions fail when executed with the integrated build process [28].

In Figure 2.8, we present a scenario representing a test conflict. The test case *totalRentTest* of *RentTest* file is responsible for evaluating the software behavior related to the calculation of rents amount (presented in merge commit). In this test case, the assertion (line 4) verifies whether a set of 4 rents costing of 40 will result in a final amount of 36.

During *Left* contributions, it updates the *getDiscountDependingTotal* method of *Rent* class only to apply discount for three or more items at the same time (line 6); otherwise, no discount is applied. Since the test cases is tested with 4 items, the program behaves as expected resulting in a passed test case.

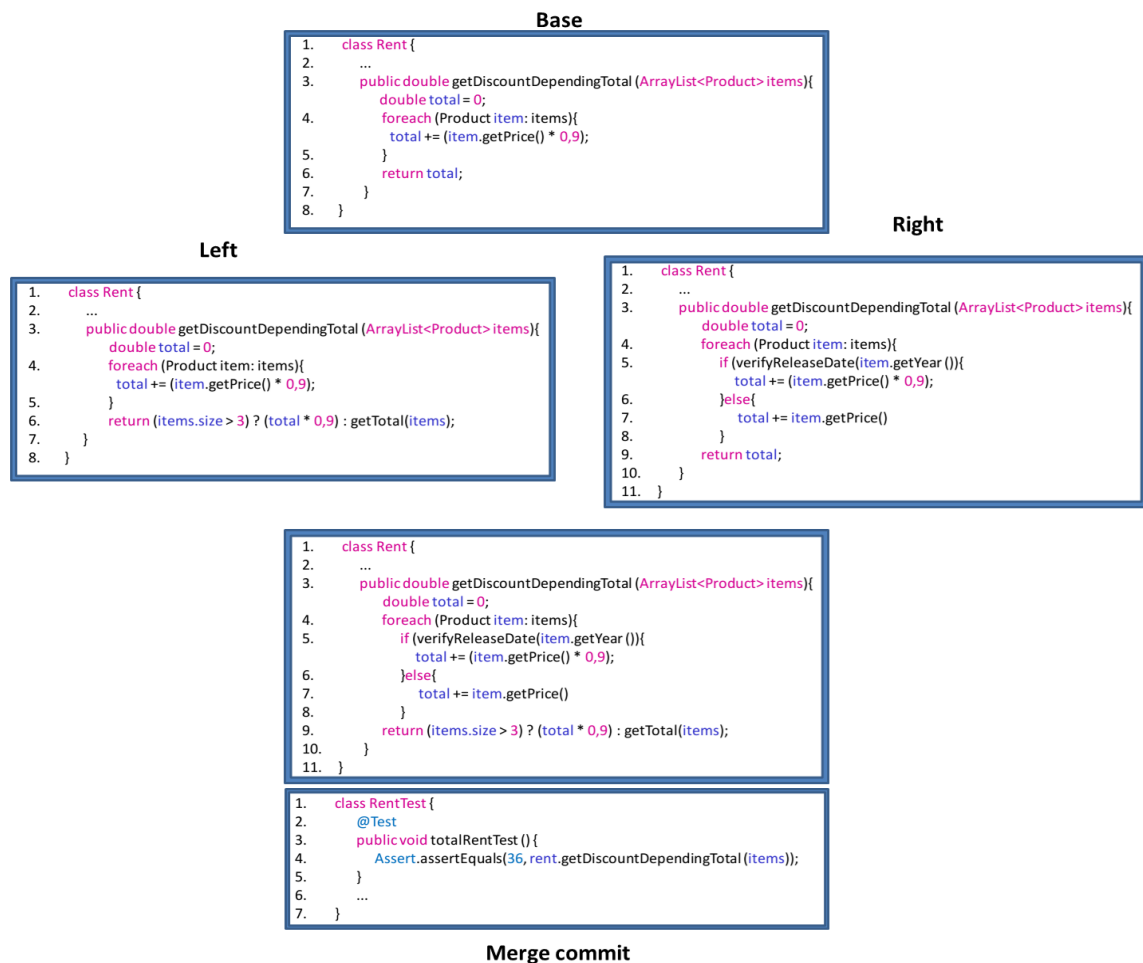
On the other side, *Right* changes the way of how the discount is calculated. It decides to apply discount only for items that have associated movies dated before 2015 (lines 4-8). Since the 4 rents tested in the test case follow this restriction, the final discount applied for the amount of 40 is also 36 (passed test case).

During the merge, some merge conflicts happen, but all are fixed leading to the inclusion of the parents contributions into the merge commit. Running the build process, the merge result source code is compilable, and a build is successfully generated, but a test case fails. Although all test cases have passed status during parent's contributions, when the different contributions are integrated, it directly impacts in the software behavior arising a test conflict. This failure happens because the discount is calculated two times (each calculation inserted by one parent). Hence, an amount of 40 produces a final amount of 32,40 instead of 36 (the expected behavior) leading the test case to fail (Figure 2.9).

Different from merge conflicts, build and test ones are hard to be identified and treated by known merge tools, even those improved. Since the nature of these conflicts is related to semantic issues, they require tools able to take these constraints into consideration when solving these problems.

2.1.2.4 Build and Test Conflicts in Practice

Here we present an example of a build and test conflict extracted from a real project. Consider that Lucas and Ray work on the same project in different sites sharing code through a

**Figure 2.8:** Merge scenario with dynamic semantic conflict**Figure 2.9:** Failed test case after merge scenario

Results:

Tests in error:

RentTest.totalRentTest unexpected result: expected "36" received "32,40"

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0

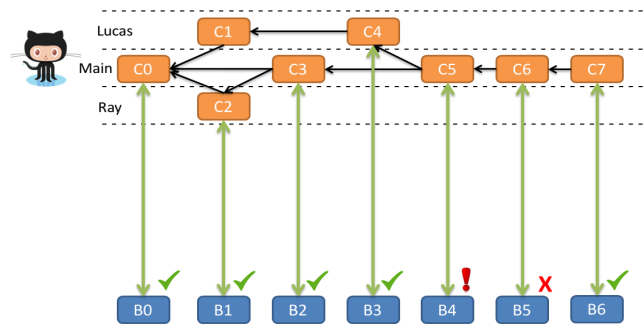


Figure 2.10: Build and test conflicts in practice

Black arrows represent the offspring of commits. The set of commits (orange boxes) between dashed lines represent the commits performed in a specific branch. Green arrows associate commits and their builds (blue boxes). Above each build, we present its status (✓, X and ! for successful, failed, and errored builds, respectively).

Compilation failure

[ERROR] /home/[...]/quickml/src/main/java/quickml/**StaticBuilders.java**:[67,116] cannot find symbol

[ERROR] symbol: method **ignoreAttributeAtNodeProbability(double)**

[ERROR] location: class quickml.supervised.classifier.decisionTree.**TreeBuilder**

Figure 2.11: Log of errored build due to compilation problem

DVCS (git using the GitHub service, as illustrated in Figure 2.10). At some point (*commit C0*), they are assigned to different but related tasks. Although Lucas commits his initial contributions (*commit C1*), Ray finishes his work before (*commit C2*) and runs the scripts responsible for building the application. As a result, a build is successfully created (*build B1*). Since the main upstream branch has not evolved, he merges his contributions into it (*commit C3, fast-forward*). He runs the build process again. Since no problem happens, he pushes the changes to GitHub repository updating the main upstream branch (*build B2*).

One day after, Lucas finishes his task (*commit C4*). Before merging his contributions with the main upstream branch, he runs the build process for the most recent commit resulting in a successful process (*build B3*). Due to Ray's contributions, Lucas merges his contributions to the actual project state (*commit C5, merge scenario*). Before updating the main upstream, he runs the build process again. Some minutes later, he realizes the build process fails (*errored status, build B4*). Verifying the build log (Figure 2.11), he concludes a method call for *ignoreAttributeAtNodeProbability* of the *StaticBuilders* class cannot be compiled because its declaration is missing in *TreeBuilder* class. Investigating the *TreeBuilder* and *StaticBuilders* classes, he confirms the missing method (*ignoreAttributeAtNodeProbability*) is not even declared; however, he is sure this method declaration was available when he performed his task. Consulting the project history, he notices Ray's contributions rename the method signature leading to the compilation problem with the newly added call. To solve the problem, the old call is updated with the new method identifier

(*attributeIgnoringStrategy*). After the changes (*commit C6*), the build process is executed again (*build B5*).

This time, the first and second phases of the build process are successfully executed (build and compilation, and automated static analysis), but one of the associated tests fails (third phase of the build process). Aiming to understand the cause of the failure, Lucas debugs the test case trying to find the cause. After a large effort, he concludes two dependent methods executed by the test case were modified. One method (modified by Lucas) receives as parameter the result of the other method (modified by Ray). After some time, he realizes both changes are consistent updating the failed test case (*commit C7*). As a result, the build process works successfully again (*build B6*), and the changes are pushed to main upstream branch.

2.1.2.5 Build and Test Conflicts by Integrator Changes

Although build and test conflicts might happen after clean merge scenarios, as described in the previous section, conflicts can also arise from unclean scenarios. In these cases, merge conflicts are manually fixed by the integrator. For example, consider the merge scenario presented in Figure 2.12. In this case, *Left* and *Right* add each one a different method declaration on the same file area (lines 3-5). Despite the difference in methods signature, the scenario is reported as a conflict.

If the integrator is not careful, the changes they apply to solve the merge conflict might introduce build or test conflicts. For instance, if the integrator preserves only *Left* contributions, any reference for the non-preserved method of *Right* contributions (*isNewRelease*) will break the build process because of the unavailable method (Figure 2.13, see Section 2.1.2.2). In this way, build and test conflicts might happen as a consequence of a badly fixed merge conflict.

The conflict of our example is considered a false positive since the parent's contributions are different. Such conflict could be fixed by improved merge tools [9] [7] [29] discarding human intervention, as also the possibility of introducing inconsistencies. These tools treat source code elements as nodes treating each node individually reducing the incidence of merge conflicts. Using these tools for merging our scenario, both methods added by the parents would be preserved on the resulting merge commit as presented in Figure 2.14.

2.2 Continuous Integration (CI)

In this context of collaborative development, CI currently is one of the software engineering practices highly adopted by software development. Such practice is responsible for automatically building and running tests or other analysis to a project, helping to ensure the quality of contributions [11] and to increase developers productivity [12] [13] [14].

CI has originally arisen as one of the twelve Extreme Programming (XP) practices [10]. Its use became common over time with the popularity of collaborative development and

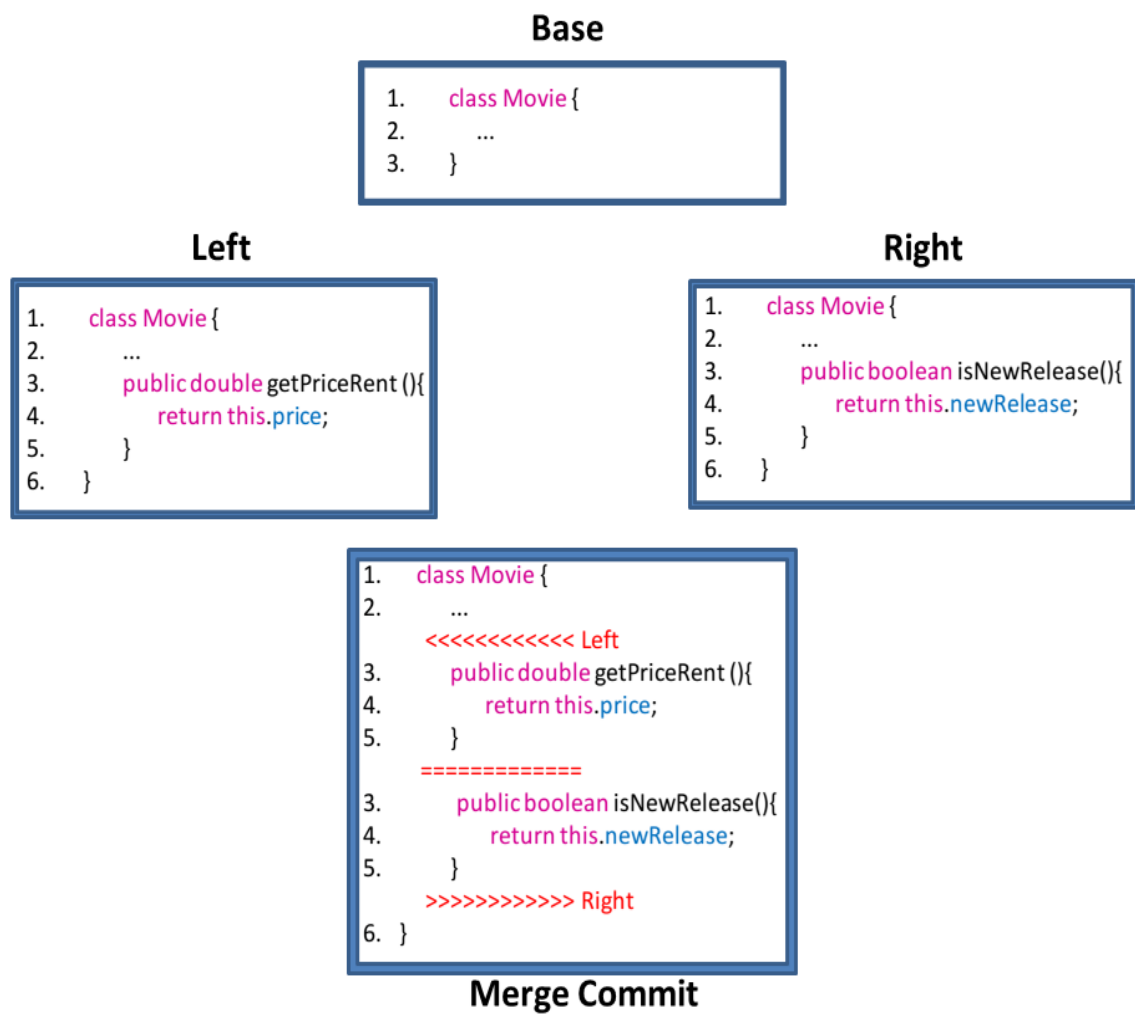


Figure 2.12: Merge scenario with false positive merge conflict

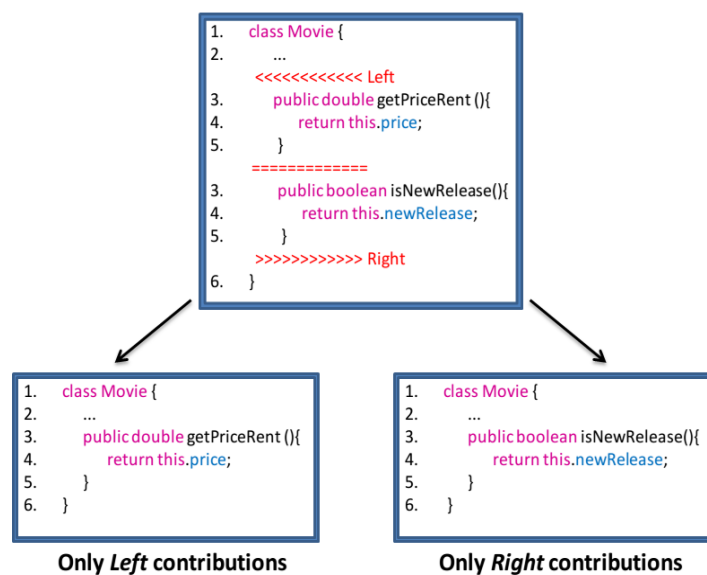
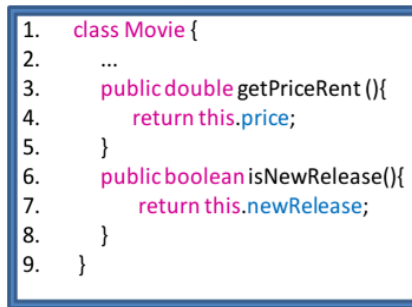


Figure 2.13: Integrator changes introducing inconsistencies on source code



```

1.  class Movie {
2.      ...
3.      public double getPriceRent(){
4.          return this.price;
5.      }
6.      public boolean isNewRelease(){
7.          return this.newRelease;
8.      }
9.  }

```

Figure 2.14: Merge scenario without merge conflicts reported by improved merge tools

open-source projects [27] offering diversity in services/tools for such practice like Travis CI [31] and Jenkins [32].

2.2.1 CI Life Cycle

A common CI life cycle build process may be composed of many phases. In our study, we adopt a build process involving three phases: (i) a build and compilation phase, (ii) Automated Static Analysis (ASAT) execution, and (iii) testing phase.¹ Despite the differences and particularities of programming languages, some steps during the build and compile phase are necessary only for some languages. The build process is sequential; if a previous phases fails, all subsequent phases are aborted. We now present the main concepts of a CI build process focusing on the phases that can fail due to problems caused by changes in source code base.

2.2.1.1 Build and Compilation

The first step of a build process is the compilation phase responsible for translating a program into a form in which it can be executed by a computer [33]. If any problem in the source code is detected during this translation, they are reported, and this cycle continues until a resulting valid translation can be achieved. This process is composed of four general steps done in sequential mode presented as follows.

The first step is the **Lexical Analysis**, also known as *scanning*. It is responsible for reading a program (source code) and groups set of characters in meaningful sequences called *lexemes* (keywords, numbers, operands, and operators).

The second step is **Syntax Analysis**, also known as *parsing*. Using the set of tokens, the parser builds a syntax tree to represent the grammatical structure of the program. Such process is done based on the programming language grammar. Some errors can arise during the construction of the syntax tree. In case of a variable declaration, the code must follow the production rules defined by the language grammar. Any problem in the variable declaration

¹We consider the term *build process* to refer the entire process of building, testing and deploying in CI context. To refer the first phase of a build process, we adopt the term *build and compilation* phase.

breaks the build process. Thus, checks are done to evaluate the conformity. Common errors in this phase are inappropriate *Variable identifiers* (like a reserved keyword, *public*), missing required attribution symbol (=), or even a missing semicolon in the end of a line (;).

The third step is **Semantic Analysis**, which focuses on identifying inconsistencies in the program according to the programming language. For that, the analysis uses the syntax tree and the symbol table. An important verification done in this step is the *type checking*, which evaluates whether the code uses the required element types. Another verification relies on the availability of variables and methods. In this case, the attempt to use a method or variable not declared yet in the program would also lead the process to fail.

Finally, the final step is responsible for **Code Generation**. The generation translates the instructions in machine code also treating the mechanical aspects (like allocation of memory, registers). We do not go further in this step since only problems during the previous steps can be the causes of broken builds. Thus, if the compilation process comes until this last step, it means the program is valid, and any problem that could occur would not be motivated by contributions of the merge scenario.

During this phase of build and compilation, the dependencies of a program are also satisfied aiming to build the software as an artifact. Project dependencies are not restricted to tools but also any external dependency required. For example, an external service, which supports the testing phase (last phase of a build process, see Section 2.2.1.3). Since these dependencies are well defined, an attempt to achieve them is done. If a dependency could not be satisfied, the build process will break. On our context, dependencies are organized in a configuration file (*pom.xml*) of each project. Just like *Syntax problems* in source code, the related file can also not be well formed leading the build process to fail. In the other hand, even dependencies well formed cannot be satisfied due to external problems (unavailability of dependency repositories) also breaking the build process. Seo et. al [17] verifies 64,71% of all build process break due to problems related to unsatisfied dependencies.

2.2.1.2 Automated Static Analysis

Besides the build and compilation phase described in the previous section, build process might include ASAT. This phase verifies the properties of the code aiming to find pre-defined problems in the project structure. It involves two kinds of problems: (i) functional and (ii) maintainability [1]. In functional problems, the tools can identify incorrect logic, like using integer division instead of floating-point division, which might impact directly in the operation result. Maintainability problems involve more issues related to project structure. For example, it can evaluate whether the project contents follow the patterns adopted for the project (style conventions) or even duplicated code.

Over time many areas have applied ASAT, like security issues [34] and vulnerabilities on web projects [35]. For Java language, many tools are available to be used, like FindBugs [36],

Check Style [37] and IntelliJ IDEA [38]. Any nonconformity found by these tools also breaks the build process.

2.2.1.3 Test Execution

After the program translation (build and compilation, and ASAT phases), the next phase is responsible for verifying whether the merge scenario result presents the expected behavior. Testing phase is not mandatory allowing some projects to skip this important verification for code quality. A report, using Travis information, shows that 20% of all projects do not include tests in their build process (in Java this percentage increases to 31%) [39]. For those that include, many test suite can be executed using a diversity of test types, like unit (most used type), integration and system tests. If any test case fails or presents errors during its execution (for example, an exception not treated), the build process breaks, and often, reports the failed test case.

Besides the execution of tests, other dynamic analysis can be performed. For example, during the execution of test cases, data about the source code exercised by the tests are extracted and used to compute, for example, code coverage. Depending on the tool for measuring the coverage, such analysis can also break the build process. For example, Jacoco [40] allows one to set up the expected coverage percentage working as a test case. If these percentages are not achieved, the build process also breaks.

We do not explain the steps after the test execution like deployment (releases of generated software artifacts). These steps involve only technical issues. Errors during the execution are not caused by parent contributions (they do not motivate the occurrence of build or test conflicts).

2.2.2 Travis CI

Among the diversity of services and tools to support CI, Travis CI [31] is the most used service offering a free and online service supporting projects hosted on GitHub. Its use and popularity have increased over time because of its simplicity and support for different programming languages. For example, the [top 10](#) most used languages in GitHub are supported by Travis; among the most popular are JavaScript, Ruby, and Java [15].

To enable Travis support for a project, it is necessary to perform the following steps. Initially, the repository must be hosted on GitHub, and the user (repository owner) must login into her Travis account using GitHub access data to grant the GitHub access permission. Thus, events sent to a GitHub repository will be reflected on build process initialization on Travis. The Travis configuration file (*.travis.yml*) must also be introduced in the project.

The *travis.yml* file is composed of instructions (common and optional) holding all information about how the build process must be performed. Despite the diversity of instructions, we only detail those that are important to the final build process status (optional instructions are used in specific cases to support common ones during the build process).

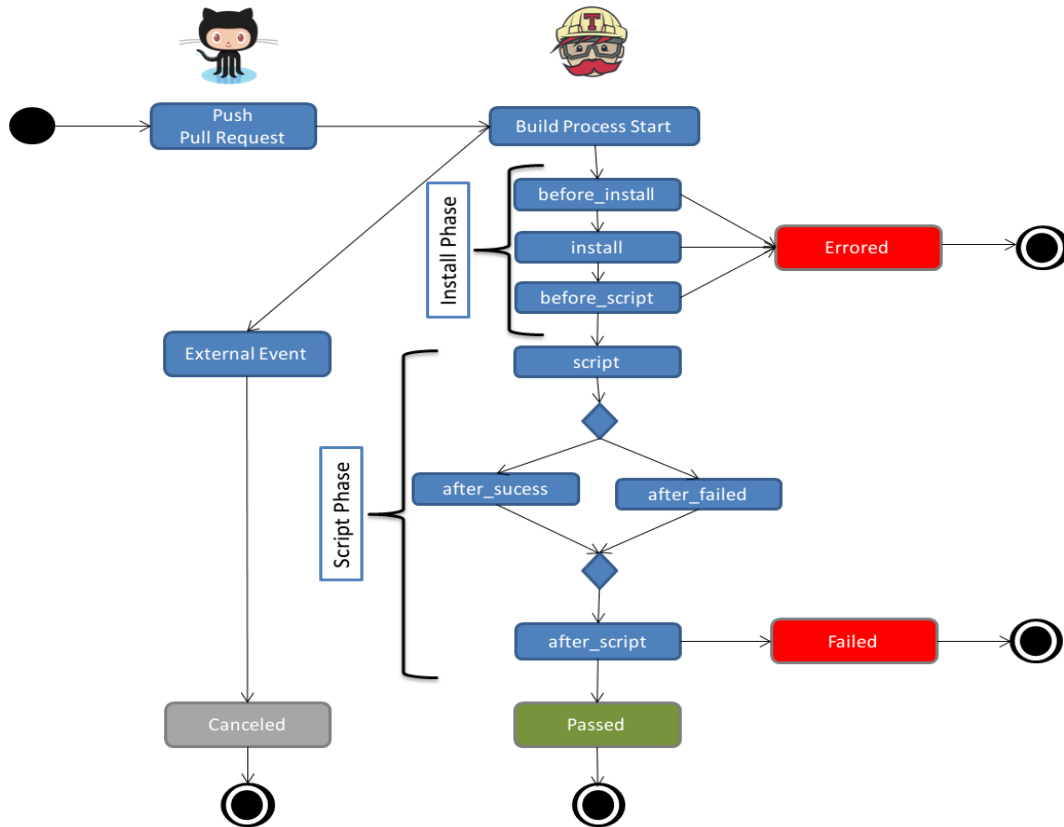


Figure 2.15: Build process life cycle on Travis CI [1]

Build Process in Travis

Moving on to how a build process works, some particularities must be explained (involving instructions of *travis.yml* file as also Travis support). In Figure 2.15, we present a state machine diagram illustrating a common life cycle build process in Travis, and over the example, we detail some instructions present in the *travis.yml* file.

As can be seen, an external event sent to GitHub repository is responsible for starting a build process (*pushes* and *Pull Requests*, PR). In this way, when such event happens, Travis try to build the project according to the new changes done in the last commit of a branch or of a PR. Pushes or PRs can involve a collection of commits, but only the most recent is built in the moment they are sent to GitHub. In this thesis, we only consider *push* events since we want to verify the real impact of build and test conflicts in software development. In this way, PR might represent only the end of many attempts not considering the entire process. However, the build process for PR follows the same principle we explain here.

During the build process, several steps are performed and their execution determines the final status of the whole build process. Figure 2.15 only comprises the final status of a build process not presenting intermediary ones (except for *started*) since they do not reflect in changes in the final status.

After the validation of *travis.yml* file (check of its instructions and structure after *Build Process Start*), the first phase of the build process can be initialized (build and compilation

phase). A build process in Travis is composed of the steps (i) **install** and (ii) **script** (highlighted in Figure 2.15). The first step is responsible for preparing the environment, which includes dependencies resolution (external and internal dependencies) and set-up of the project. This step is composed of three instructions: *before_install*, *install* and *before_script*. If any of them fails, the build process breaks with an *errored* status. The second step consists of running the build process based on the instructions informed on *travis.yml* file configuration. It is normally composed of four instructions: *script*, *after_success*, *after_failure*, and *after_script*. If during ASAT execution any problem happens, it also leads the build process to break with an *errored* status.

With the ASAT phase finished, it starts the testing phases and dynamic analysis are performed. Some projects might perform additional analysis more related to project structure like metrics. Others use this time for deploying to a server the generated build artifacts and build documentation. If any test case fails or any additional step presents an error, the build process breaks with a *failed* status. Finally, if the build process does execute all the previous steps without any problem, the *passed* status is achieved meaning the CI life cycle was successfully performed. The last possible final status is *canceled*, which is motivated by an external event done at any time during the build process (a user cancels the process before it ends).

Each build in Travis is associated with an exclusively commit. However, a commit can have many builds. For example, if a commit *CI* has a build *B1* in Travis, and after some time, a reset is done to point to this commit (*CI*), a new build process will be started in Travis (build *B2*). Even these different builds are associated to the same commit, they can achieve different status. The build *B1* can be successfully built, while *B2* can fail during the first phases since dependencies can become deprecated over time.

Besides the final build status, for each build, a log is reported showing how the build process works. Part of this log is composed of the outputs generated by the build manager adopted for the project. For Java language, Travis allows the adoption of three **build managers**: Maven [41], Gradle [42] and Ant [43]. Depending on the adopted build manager, some logs are more informative than others. For our experiment, a log report as complete as possible is critical since we use it to get the causes of broken builds and additional information used to perform our analysis (we detail it in Section 3.3)

3 IDENTIFYING CONFLICTING CONTRIBUTIONS IN MERGE SCENARIOS

The complexity and the difficulty to understand the causes of build and test conflicts are a common problem in software development. Although we know that build and test conflicts happen in practice and might involve different and dependent files [8] [2], it is important to better understand the frequency and causes of these conflicts, so new improvements could be applied to assistive tools aiming to treat or even avoid such problems.

Previous studies present evidence of build and test conflicts in practice. Brun et al. [8] reveal 33% of all evaluated merge scenarios (scenarios without merge conflicts) represent cases of build and test conflicts. In the same way, Kasi and Sarma [2] bring such evidence separately: build conflicts ranging from 2% to 15%, while test conflicts ranging from 6% to 35%. Despite such evidence, both studies use the same approach for identifying conflicts (building and testing all merge commits and checking whether the build process can be successfully built or present problems), which can introduce bias in the results. For example, they do not check the parent commit build process even though inconsistencies can come from the parents. Although they consider a small sample, they execute the build process, and consequently, any different environment configuration can break the build process leading them to a wrong conclusion.

In this way, to further investigate these kinds of conflicts eliminating these known threats, we analyze 60991 merge scenarios from 529 GitHub Java projects that use Travis CI revealing more than 150 conflicts. For each merge scenario, we check whether it leads to a build or test conflict: the merge commit build is broken while its parent commits present superior status (successful or failed, for build conflicts, and successful, for test ones). We consider *failed* status for build conflicts since the source code associated to a *failed* build process could at least be compiled and built during the build process. It means the source code should not have any problem that could be associated to a build conflict.

In some merge scenarios, some commits do not have an associated build in Travis. For these cases, we use Travis to build them aiming to achieve completeness in our experiment. In Section 3.3, we discuss how we check whether a broken build is the result of a build or test conflict. In both cases, we classify the conflicting contributions between parent's contributions. For build conflicts, we use the GumTree tool [44] (a syntactic diff) to compare the contributions of each parent, and see how they impact each other. For test conflicts, we re-run the failed test cases aiming to ensure the failure happens in practice (not motivated by external issues, like parallel test execution and unavailable resource during execution). During this step, we also

instrument our script to compute the associated code coverage for each test execution. Comparing coverage with the changes performed by the parent contributions, we look for some intersection among them (for example, same classes changed by parents and exercised during the test case).

3.1 Problem Statement

Previous studies have investigated the occurrence of conflicts during software development [8] [2]. Especially for build and test conflicts, these studies do not focus on investigate the nature of these types of problems. They perform a superficial analysis aiming to verify the frequency of each type, and based on their findings, tools are presented responsible for avoiding such conflicts. Nevertheless, these tools involve superficial characteristics related to conflicts causes since there is a lack of information about the causes responsible for them.

The first main difference in our study, when compared to related work, is the investigation of the structures of changes associated with the causes of build and test conflicts. It is known that build conflicts involve dependent files [2], but there is a lack of information about the level and direction of such dependency. For example, there are many ways for a dependency involving two files in a project (like interface implementation, heritage), and it is not known the dependency types that might cause build conflicts. In the same way, for test conflicts is known that after a merge scenario, the software behaves unexpectedly presenting failed test cases. However, it is not known how the contributions in each parent commit impact with each other leading a test case to fail.

Since the causes of conflicts are not investigated in related work, the resolution patterns adopted to fix the conflicts might not be verified. We go further and for each conflict we identify its commit fix. Thus, we check its associated resolution pattern aiming to verify the pattern itself as also to understand the circumstances such resolutions are applied. We may check whether a conflict can be fixed involving more than a resolution pattern, and how the fix is related to the parent's contributions. For example, whether a fix adapt the conflict to the new project state or undo contributions going back to an old commit (previous project state). With this information, we can provide insights or improvements for assistive tools responsible for dealing with conflicts.

3.2 Study Design

Our experiment consists of three main steps as illustrated in Figure 3.1. The first step is responsible for getting all information required for the experiment, which includes mining repositories (source code and builds information). The second step performs verifications aiming to filter non conflicts. Finally, the last step identifies the causes of build and test conflicts and the adopted resolution patterns (we identify the closest commit following the merge verifying the fixes and metrics related to who fixed it, when available).

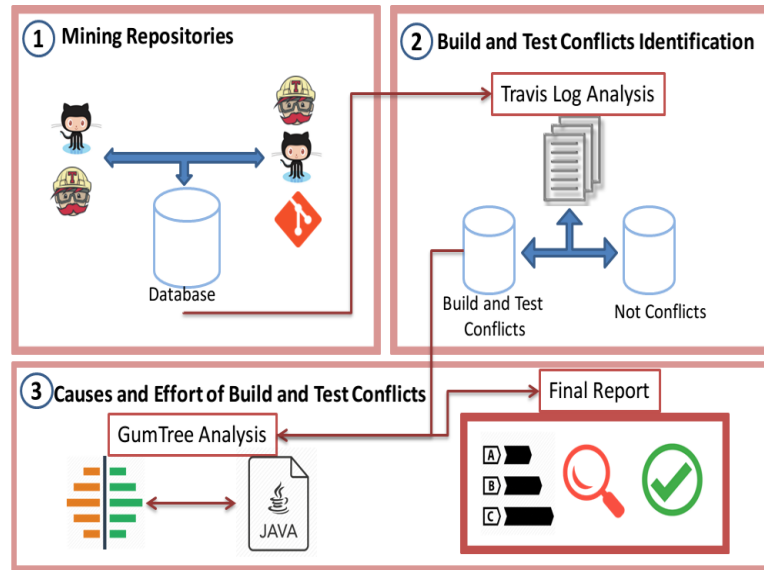


Figure 3.1: Study design

3.2.1 Mining Repositories

In the first step of our study, we access source code repositories to collect merge scenarios and associated information of the build process, indicating whether the build process of a merge commit fails during its attempt, while the parents commits are successfully built. As subjects, we first select real Java projects available on GitHub. To obtain the associated information of build processes, we could run build process for merge scenarios locally (merge and its parent commits) obtaining the desired information. Given the result of a build process might heavily depend on environment configuration, generating builds in our machines could introduce bias in the results. As this would also demand large manual effort, we decide to opt for selecting projects that use Continuous Integration (CI). As not all CI tools and services provide their data openly, we focus on projects that use Travis CI (one of the most used CI service [15] [16]). It openly provides all build information associated with a project through its public [API](#).

3.2.1.1 Sample

To select our sample, we select projects from Munaiah et al. [45] and Beller et al. [46] database as such studies consider real projects. However, we select only Java projects as there are many projects with different programming languages. To select a relevant sample, we consider only projects with more than 40 stars and 50 forks. Such constraints allow us to analyze real projects excluding those toys. The projects need to be active since we would build some merge scenarios, and projects with deprecated dependencies would not present successful builds.

With the list of projects selected, for each sampled project, we do three checks: (i) to verify the presence of the `.travis.yml` file, (ii) to consult its status on Travis service (active or not), and (iii) to verify the exclusive presence of the `pom.xml` file. These checks are necessary to

ensure only the presence of projects using Maven as build manager as also with data available on Travis.

Although we have not systematically targeted representativeness or even diversity [47], we believe that our sample has a considerable degree of diversity concerning, at least, the number of developers, source code size, and domain. It contains projects from different domains such as APIs, platforms, and Network protocols. They also vary in sizes and number of developers. For example, [XChange](#) project has only 20,4 KLOC, while [Wicket Bootstrap](#) has approximately 282 KLOC. Moreover, [Wire](#) project has 37 collaborators, while [Java Driver](#) has 92. The list of the analyzed projects can be found in Appendix D.

3.2.1.2 Filtering Sample Projects

With our sample defined, as presented in the previous section, for each project selected, we clone it and identify all merge commits using Git API returning a list of all merge commits in a specific branch.¹ To get all merge commits of a project, we locally checkout in every available branch, and for each branch we collect its commits grouping all commits into a single set. Since Travis CI is relatively a new technology, most projects adopted it later in their life cycle. For these projects, the list of commits returned by Git API includes commits performed before the adoption of Travis, and consequently, they do not have build information available on Travis. To select only merge commits performed after Travis CI adoption, we parametrize the git command to return only commits dated after the first build finished in Travis resulting in 60991 merge commits (step 1 in Figure 3.2). As described in Section 2.1.1, we discard *fast-forwards* merge commits.

This filter also contributes to exclude merge scenarios from projects that are configured to use Travis, but in practice, they do not use [15] [16]. This might happen because the adoption of Travis is composed by two steps: (i) inclusion of Travis configuration file on the project, and (ii) permission for the project be built on Travis. However, some projects configure Travis doing only the first step. We notice such behaviour when a project is configured to adopt Travis (presence of *.travis.yml* file), but none build in the Travis repository is associated to the project.

For each selected merge commit, we try to identify its associated build, but not all commits have an associated build on Travis. For example, in some cases, a merge commit is built, but one of its parents is not. Other cases, even the merge commit does not have a build in Travis. This might happen because only the last commit of a push is built; if new commits are done after a merge integration, these last will have a build instead of the merge commit. Another reason for commits without build information is the origin of these commits. Depending on the branch where commits are done, none build will be started on Travis. This might happen due to Travis restrictions as it is possible to restrict a [list of branches](#) to build commits. If one of the parents was not in this list of branches, it will not have permission to start build process. Another

¹`git log -merges`

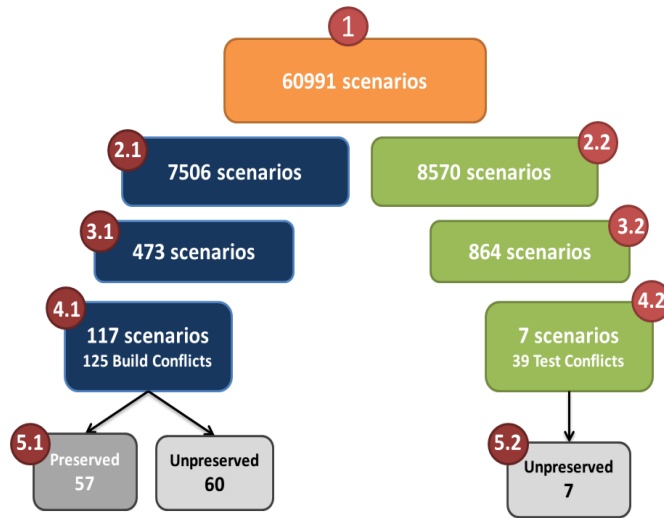


Figure 3.2: Merge scenarios filters adopted during the analysis

Boxes represent the number of remaining scenarios in each step (blue for build conflicts, and green for test conflicts): (1) All merge scenarios, (2.1, 2.2) errored and failed builds, (3.1, 3.2) errored and failed builds with superior parent build statuses, (4.1, 4.2) removal cases caused by external causes or weakness by our scripts, (5.1, 5.2) classification of scenarios based on parent's contributions preservation.

reason is related to the lack of continuity when a fork is done. For example, if one of the parents of a merge commit is done in a fork, build information will be only available in the related Travis repository of this fork. However, most fork projects do not activate Travis not performing any build process. We try to access fork repositories on Travis, but only a small percentage of build process are available.

To make sure all commits in a merge scenario have associated Travis builds, we force the creation of builds when they are not available. We basically fork each project on GitHub using its [API](#) and perform resets pointing the heads to these commits not built yet in Travis. After this, we push them to our remote fork, which triggers Travis to start the build process. Although these new build process are not done in the official environment, Travis ensures each process has the appropriated environment configuration. After making sure each selected merge commit has an associated Travis build, we further filter our sample by selecting only the scenarios associated with errored or failed builds (6903 and 8149, respectively, steps 2.1 and 2.2 on Figure 3.2).

Merge scenarios having merge commits with associated errored or failed builds do not necessarily represent build or test conflicts. In fact, the issue with the build might have been inherited from one of the parents, and not caused by conflicting contributions. So we further filter the samples by selecting only the scenarios having parent commits with superior build statuses (failed or passed for errored builds, and only passed for failed ones). This results in 473 and 864 scenarios, respectively (steps 3.1 and 3.2 in Figure 3.2). In case one of the parents has the same status of the merge commit, we assume the broken build of the merge commit is carried over from the broken parent. So there is not really a conflict between development tasks, but

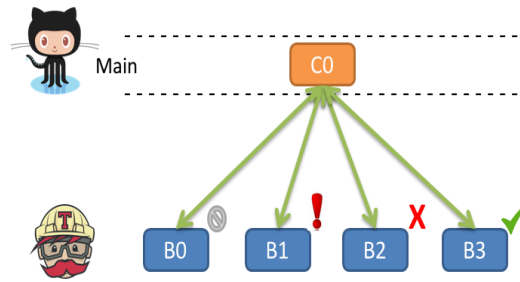


Figure 3.3: Different build status for the same commit. **X** indicates an errored build, while **!** indicates a failed one.

simply a defect introduced by one of the tasks, eventually, reaching the merge commit.

As a single commit can be associated with more than one build (see Section 2.2.2), each one with a different final status, we adopt the build status based on the precedence rules defined by Travis. These rules reflect the final build process status according to the occurrence of problems during the phases of a build process. Thus, we have: *canceled* < *errored* < *failed* < *passed* (organized from the lower to the higher status). For instance, a *commit* C0 can be linked to four builds (B0, B1, B2, and B3), each having a different status (Figure 3.3). In this situation, we consider the status of build B1 as the one associated to the commit C0 (errored status) because a problem during the compilation and build phase breaks the build process. In the builds B2 and B3, this phase is done without errors. We do not consider the canceled status because it is triggered by an external event (user cancels the build process) or previous configurations that cannot be motivated by parent's contributions. However, different builds related to a single commit, commonly, present the same result.

As the output of this step (3.1 and 3.2 in Figure 3.2), we have a set of merge scenarios having merge commits associated with broken builds (473 and 864, errored and failed merge commit builds, respectively), and parent commits with superior build process statuses.

3.3 Conflicts Identification

To ensure the selected merge scenarios so far are associated with conflicts, and to better understand what caused the errored or failed status, we need to further investigate the build status and associated logs (step 2 in Figure 3.1). This is needed because the non passed build status might have been caused by a number of reasons not related to the merged contributions. So we analyze, for each build, its Travis log report. Our scripts automatically seek for specific message errors listed in Table 3.1. The message patterns that compose this table were identified by a manual exploratory analysis of build logs. Each problem can hold many different message patterns since different versions of Travis and build managers imply in different patterns. Some of them are positively related to the occurrence of conflicts, whereas others are certainly not related. In particular, some of the messages indicate that external (having no relation with the

contributions) causes were responsible for interrupting the build process:

- **Remote Problems:** The build fails because of Travis restrictions or external services required by a build process. For example, if a build process does not present any activity in an interval of 10 minutes, the associated build process will break. We discard these scenarios because they do not reflect issues caused by the developer's changes, therefore not characterizing a conflict.
- **Environment Configuration:** Build process fails due to unresolvable or wrong project dependencies. We only discard the scenarios with no changes performed on configuration files during a merge scenario. This restriction ensures an external problem is responsible for the failure; if contributions change configuration files, conflicts might arise, leading to this kind of message, and demanding further analysis.

Discarding these scenarios caused by external problems, and also those that our script cannot evaluate due to its limitations or lack of precise information, the remaining merge scenarios compose the set we consider to further investigate the occurrence of build and test conflicts. This corresponds to 117 and 7 merge scenarios, respectively, (steps 4.1 and 4.2 in Figure 3.2).

We explore the particularities of each kind of broken build. For instance, errored builds are caused due to problems during build and compilation, and Automated Static Analysis (ASAT) phases of the build process. In the same way, failed builds are caused when the previous phases are successful done, but problems happen during the testing phase (see Section 2.2.1). Based on these problems, we manually investigate a few cases aiming to extract message patterns related to them. This is detailed when we present our strategy to classify conflicts. For now, we classify the problems in categories of broken builds informing whether they can cause build or test conflicts. Table 3.1 presents for each cause of broken build (third column) its related message patterns (fourth column).

Preserved vs Unpreserved Merges

Build conflicts may arise from merge scenarios with conflicting contributions (contributor changes), or caused by changes performed after the integration (integrator changes). These changes made after the integration might be motivated by merge conflicts fixes as also addition of new contributions. Related work only consider contributor changes as the cause for build conflicts, because they cannot evaluate merge scenarios resulted of merge conflicts during the integration. Assuming integrator changes as a potential cause is an attractive area not explored yet. It is important to mention that an integrator responsible for the integration can also be one of the contributors of the merge scenario. Nevertheless, our focus at this point is to identify the conflict cause, and consequently, who was responsible for introducing the inconsistency.

Table 3.1: Build error messages taxonomy and how they related to conflicts or the lack of conflicts?

Conflict Type	Category	Name	Message
Build Conflict	Syntax	Malformed Program	illegal start of type
			unexpected token
			illegal character
			missing return statement
	Static Semantics	Unimplemented Method	does not override abstract method
		Duplicated Duplication	is already defined in
		Unavailable Symbol	cannot find (symbol method variable class)
			could not (find transfer) artifact
		Incomp. Methods Signature	no suitable method found for
			cannot be applied to actual and formal argument lists differ in length
		Incomp. Types	error: incompatible types: [...] cannot be converted
	Other Static Analysis	Project Rules	Some classes do not have the expected license header
Test Conflict	Dynamic Semantics	Test failures	There (are was) test failures Failed tests:
	Other Analysis	Unachieved Coverage	Build failed to meet Clover coverage targets
No Conflict	Environment Resource	Remote Problems	The job exceeded the maximum time limit for jobs, and has been terminated
			No output has been received in the last [interval time]
			Your test run exceeded 50.0 minutes
	Dependency	Environment Configuration	Could not (find transfer) artefact

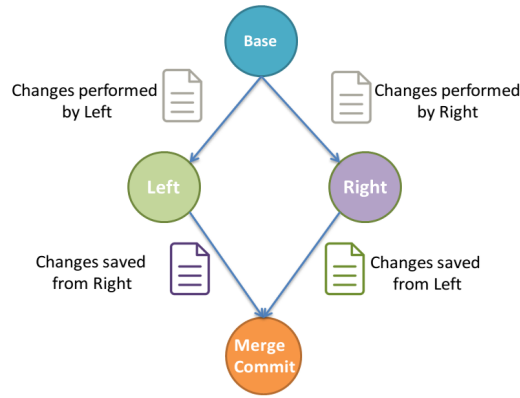


Figure 3.4: Checking contributions performed on merge scenarios parents

In this way, two kinds of changes made by integrators might break builds: (i) changes applied to fix merge conflicts and (ii) changes performed after the integration. For the first cause, we could identify improvements for merge tools since the integrator needs to deal with merge conflicts because of merge tools weakness. For example, as presented in Section 2.1.2.5 (Chapter 2), merge scenarios can present merge conflicts reported only by textual merge tools. Improved merge tools like S3M [9] are able to treat merge conflicts without reporting them to integrators. In the other side, for the second cause, we could identify the circumstances that lead an integrator to perform changes after the integration.

To have a better insight of how incident build conflicts can occur for each cause (contributor or integrator changes), we classify merge scenarios based in parent's contribution preservation. In this way, a merge scenario can be classified as *preserved* (all parent's contributions are preserved in the merge commit) or *unpreserved* (at least one parent contribution is not preserved in the merge commit). This classification allows us to verify if a merge scenario type (preserved or unpreserved) is likely to have more build conflicts caused by an specific cause (contributor or integrator changes).

To classify the scenarios, we simulate each merge scenario locally. We checkout in each parent commit and try to perform a merge between them. If none merge conflict happens, we compare the integration result with the original merge commit. For simulations without differences to the original merge commit, we consider the related merge scenario as *preserved* (step 5.1 in Figure 3.2). It means all parent's contributions are preserved. It is important to mention that new source code can be introduced by the integrator, but this addition does not imply in parent's contributions removal. For the 57 merge scenarios classified as preserved, a compilation problem is likely to be a build conflict since only after the merge integration the build process breaks. For the remaining simulations (those with merge conflicts), further analysis is necessary to check whether all parents contributions are preserved even with the occurrence of merge conflicts and changes performed after the integration. Thus, we use the syntactic diff tool GumTree [44] to identify and associate changes to either contributors or integrators.

Understanding Contributions in Unpreserved Merges

To verify whether parents contributions are preserved even with the occurrence of merge conflicts, we generate syntactic diffs to get the difference between the base and the parent commits. For that, we extract the base commit (common ancestor) between the merge parents.² Our scripts generate a syntactic diff from each parent and the base gathering all changes performed by each parent (**base-parents-diffs**, the top diffs illustrated in Figure 3.4). This diff considers only the changes present in the last commit of each parent. If a previous commit introduces a new file, and a following commit removes it before the final parent commit, such action of file introduction/removal will not be present in the diff. The important issue for us is the set of changes presented in the final commit based on the base commit.

To check if all changes are preserved in the merge commit, we also generate syntactic diffs between the parents and the merge commit gathering only the changes preserved by each parent (**parents-merge-diff**, the bottom diffs illustrated in Figure 3.4). If **base-parents-diffs** are not included in **parents-merge-diffs**, we conclude at least one parent contribution is not preserved classifying these scenarios as *unpreserved*. As result, 60 scenarios of build conflicts are classified as unpreserved merge scenarios (step 5.1 in Figure 3.2).

For test conflicts, we use the same approach to classify the related merge scenarios based in parent's contribution preservation (preserved or unpreserved). In this case, 7 unpreserved scenarios are identified (step 5.2 in Figure 3.2).

3.4 Classifying Conflicts

After the removal of merge scenarios associated with broken builds caused by external or remote problems, we perform further analysis in the remaining merge scenarios to classify the conflicts into categories. Next sections we present our approach to classify build and test conflicts.

3.4.1 Classifying Conflicting Contributions on Build Conflicts

As presented in Table 3.1, errored builds might have many causes. In this section, we discuss how we classify build conflict causes into categories. Since *Malformed Program* cause is related to a syntactically problem on source code, we verify if merge conflicts happen meaning the integrator is responsible for the conflict. Otherwise, we evaluate the case manually. For *Unavailable Symbol*, we analyze whether a new reference for a symbol is added, while such symbol is removed by the other parent. In the same way, for *Incompatible Method Signature*, we verify whether a method declaration is updated (change on parameters list or types), while a new reference for such method is added. For *Duplicated Declaration*, we analyze whether two declarations of the same type sharing the same name are introduced in a class. For *Unimplemented*

²`git merge-base <HASH-commit> <HASH-commit>`

Table 3.2: Unavailable Symbol information from Travis Log

Element	Description
Main Class	it represents the class that refers the missing symbol
Missing Symbol	it represents the identifier of the missing symbol
Secondary Class	it is the class that held the missing symbol

Method, we verify whether a new implementing class does not implement a method introduced in its related interface. For *Incompatible Types*, we verify whether a type in a call is updated, while a new reference for such element is done. For *Project Rules*, we verify whether the classes follows the project style guidelines. These last two causes are checked manually.

Here we detail how we classify build conflicts due to *Unavailable Symbol*, which are related to the attempt to reference an element no longer available in the project, even though such element is available in the parent commits (step 3 in Figure 3.1). The other causes are similarly identified and are detailed in Appendix A). The source code of all scripts and classification algorithms are available in our online [repository](#) [20].

An unavailable symbol can be a variable, a method or even an entire class. For example, Travis log reports for an unavailable symbol triggered by two classes: (i) main class, which requires the missing symbol, and (ii) secondary class, which should hold the missing symbol. It is possible that these two classes are the same class or dependent ones. For example, a class requiring a method inside another method body, or from another class. Although each symbol type share the same problem nature, each one has its particularity related to classify the conflict. Here we present our approach to classify build conflicts caused by unavailable method. We explore three possibilities that might be responsible for a method becoming unavailable in a project during a merge scenario (caused by contributor or integrator changes, as also due to project dependency).

To identify interference between parents contributions and classify the conflicts, we use syntactic *diffs* generated in previous steps (**base-parents-diffs** and **parents-merge-diffs**, see Section 3.3).

Unavailable Method — Contributors source code changes

To illustrate how we classify unavailable symbol conflicts, we consider first the Java-driver project presenting an example of build conflict caused by Unavailable Symbol (missing method) due to contributor changes. Looking at the build log,³ it is possible to see a reference for a symbol that could not be compiled (*cannot find symbol*, Figure 3.5). This message means a symbol becomes unavailable during the merge scenario.

Travis log also presents useful information (Figure 3.5), which are explained in Table 3.2. In this case, the *Activator* class (main class) requires the use of the missing method *builderWithHighestTrackableLatencyMillis* provided by the *PerHostPercentileTracker* class

³Build ID: datastax/java-driver/144600040 - Merge Commit: 203a409

Figure 3.5: Build log of errored process

```

Compilation failure
[ERROR] /home/[...]/datastax/driver/osgi/impl/Activator.java:[77,89] cannot
find symbol
[ERROR] symbol: method builderWithHighestTrackableLatencyMillis(int)
[ERROR] location: class com.datastax.driver.core.PerHostPercentileTracker

```

Figure 3.6: Original version of GumTree diff for conflicting contributions

```

//PerHostPercentileTracker.java
// Diff between Base <4cf58a8> and Left Parent <bc4f313>
Delete SimpleName: builderWithHighestTrackableLatencyMillis(45)

// Empty diff between Base <4cf58a8> and Right Parent <932e387>

```

```

//Activator.java.java
// Empty diff between Base <4cf58a8> and Left Parent <bc4f313>

// Diff between Base <4cf58a8> and Right Parent <932e387>
Insert SimpleName: builderWithHighestTrackableLatencyMillis(235)
    into MethodInvocation(237) at 1

```

(secondary class). With this information, we use the syntactic diffs between the base and each parent commit (**base-parents-diffs**) and try to verify if their changes caused this problem. Figure 3.6 presents the diff for the *PerHostPercentileTracker* and *Activator* classes.

As can be seen, the Left parent removes the missing method declaration (*builderWithHighestTrackableLatencyMillis*) from *PerHostPercentileTracker* class (secondary class) and does not perform any change in the requiring class (*Activator*, main class). On the other hand, Right adds a reference for the missing symbol in *Activator* class and also does not perform any change in the *PerHostPercentileTracker* class. Thus, we have an example of interference between parent's contributions. The change performed by one of the contributors impact the work of the other. To automatically classify this kind of conflict, our algorithm has the following steps:

- **Step 1:** Identification of the parent responsible for updating or removing the missing symbol using the *Secondary class diff* (in our example, Left parent).
- **Step 2:** Verification of a new reference for the missing symbol in the *Main class diff* in the other parent (in our example, Right parent). If all these steps are successfully performed, we identify a build conflict caused by **integrator changes**.

Unavailable Method — Integrator source code changes

If parent's contributions are not conflicting, we further investigate the changes performed by the integrator. To illustrate how we classify *Unavailable Symbol* conflicts caused by integrator

Figure 3.7: Build log of errored process due to contributor changes

```

Compilation failure
[ERROR] /home/[...]/downloader/HttpClientDownloaderTest.java:[128,24]
cannot find symbol
[ERROR] symbol: method putParams(java.lang.String,java.lang.String)
[ERROR] location: variable request of type us.codecraft.webmagic.Request

```

Figure 3.8: Original version of GumTree diff for integrator changes

```

//Request.java
// Diff between Left <74110e6> and Merge Result <fe95a68>
Delete SimpleName: putParams(355)

// Diff between Right <395396c> and Merge Result <fe95a68>
Delete SimpleName: putParams(433)

```

changes, consider the merge scenario in the Webmagic project.⁴ Both merge parents change the same file area of a class leading the integrator to a merge conflict. After the integrator fixes the conflict, the resulting build presents an errored status on Travis, as can be seen in Figure 3.7.

The build fails because a test case of *HttpClientDownloaderTest* (main class) references the missing method *putParams* no longer available in the *Request* class (secondary class). Analysing the syntactic **base-parent-diffs** of the *Request* class, no parent removed the declaration of the missing method. Nevertheless, the **parent-merge-diffs** show the removal of the missing method in both parents as presented in Figure 3.8.

Thus, we conclude the integrator removes the missing method after the merge integration. The change performed by the parents are not conflicting, but the contributor changes introduces the cause of the broken build. Based on this idea, our approach is done:

- **Step 1:** Verification of *missing element* removal in both *Secondary* class **parent-merge-diffs**. If true, we conclude a build conflict happens due to **integrator changes**.

Unavailable Method — Integrator configuration code changes

If changes on source code do not cause an errored build, we further investigate whether changes on project dependencies can cause a build conflict. For example, in the Solo project, an errored build is caused by unavailable symbol (Figure 3.9).⁵

Despite the missing symbol in the *Markdowns* class, only the Right parent performs changes in configuration files and source code. Some of the modification in the configuration

⁴Build ID: leusonmario/webmagic/237768118 - Merge Commit: fe95a68

⁵Build ID: leusonmario/solo/260314193 - Merge Commit: bc6fa3b

Figure 3.9: Build log of errored process due to missing dependency

```

Compilation Failure:
[ERROR] /home/[...]/solo/util/Markdowns.java:[53,15] cannot find symbol
[ERROR] symbol: class PegDownProcessor
[ERROR] location: class org.b3log.solo.util.Markdowns
[ERROR] /home/[...]/solo/util/Markdowns.java:[53,55] cannot find symbol
[ERROR] symbol: class PegDownProcessor
[ERROR] location: class org.b3log.solo.util.Markdowns
[ERROR] /home/[...]/solo/util/Markdowns.java:[53,72] cannot find symbol
[ERROR] symbol: variable Extensions
[ERROR] location: class org.b3log.solo.util.Markdowns

```

file (pom.xml) is to use new dependencies.⁶ Left parent changes only configuration files (*POM* and *travis.yml*); one of these changes in the POM file is in the same area changed by Right. Consequently, merge conflicts arise leading the integrator to fix them. Nevertheless, during the fixes, the new dependency added by the Right parent is removed by the integrator leading to a broken build. In this scenario, the dependency removed by the integrator during merge conflicts is responsible for the broken build.

In such cases, we perform a manual analysis to ensure the build conflict occurrence is caused by dependency issues and categorize its motivation (contributor or integrator changes).

3.4.2 Classifying Conflicting Contributions on Test Conflicts

As presented in Table 3.1, we identify two possible causes of test conflicts, which we detail as follows (step 3 in Figure 3.1).

Failed Test Cases

Problems during the testing phase of a build process are often caused by failed test cases, which may be motivated by failures as also errors during the test execution. Failures represent an unexpected software behavior, while errors represent the incapability of executing a test case due to unexpected operations.⁷ This leads to failed builds as discussed previously in this section (see Table 3.1). Travis log contains for such build process the set of failed test cases, also indicating for each failed test case, the file containing its declaration. For example, in one build of the Cloud-Slang project,⁸ only the test case *testSubscribeOnAllEventsWithListener* of the *SlangImplTest* class fails because of a unexpected software behavior. Figure 3.10 presents the associated test log highlighting the failed test case.

To ensure test case failure occurs because of the intersection of parent's contributions and not by external causes, we decided to individually execute each failed test case again. To

⁶Pegdown: <https://mvnrepository.com/artifact/org.pegdown/pegdown>

⁷These terms come from build manager tools reported as output for each build process.

⁸Build ID: CloudSlang/cloud-slang/75814949 - Merge Commit: 20bac30

Results:
Failed tests:
SlangImplTest.testSubscribeOnAllEventsWithListener:
Tests run: 14, Failures: 1, Errors: 0, Skipped: 0

Figure 3.10: Failed test case on failed build process

evaluate whether the methods and classes exercised during the execution are also changed by the parents commits, we also analyze the coverage of this individual re-execution of the failed test case (using syntactic diffs). If a failed test case exercises source code changed during parent's contributions, these changes might have a direct impact in the unexpected software behaviour. In the other side, source code changed during parent's contributions, but not exercised by the failed test, do not have any impact in the failure since it is not part of the software behavior scope evaluated by the test case.

Our scripts analyze the Travis log identifying the failed test case and the associated class. With such information, we use Travis to perform a new build process restricting the testing phase for each failed test case. We re-execute the failed test case since any external problems could be responsible for the fail. For example, parallel tests execution or the unavailability of a required external server. This is done in two phases: (i) test case execution and (ii) analysis of code coverage and parent's contributions.

Phase 1 - Test Case Execution

In the first phase, just like the approach used for building non-built commits in merge scenarios (see Section 3.2), we continue to use Travis to run the build process. However, this time, our scripts do not only perform resets. Since for computing code coverage information, one has to change configuration files adding a new dependency for a tool responsible for generating the coverage. Additionally, it is necessary to deploy this information out of Travis environment. So our scripts reset the head of the our fork project to point to the merge commit associated with the failed build and adapt the configuration files to our needs (coverage generation and deployment on GitHub using tags for each commit). After the test case execution, we instrument our scripts to save the coverage associated with each test case in GitHub.

Initially, our scripts instrument the POM file with a new dependency ([Coverage](#)). Additionally, we introduce a new dependency, the [Maven Surefire plugin](#), to allow the execution of a single test case during the build process. In the same way, our scripts change the *travis.yml* file to ensure the build process will only execute the steps necessary until the testing phase start. Thus, if the *travis.yml* file has instructions for deployment, they are removed and discarded. The approach adopted for adapting *travis.yml* file is detailed in [Appendix C](#).

After all changes are performed, we commit and push the new changes to GitHub associating each commit with a Git tag in the commit message. It is important to mention that a failed build can be associated with one or more failed test cases. Since we perform for each one

a new build process, we also associate them with an exclusively Git tag. Each tag will be used to deploy the coverage in GitHub as also to download it during the next phase.

Phase 2 - Parent Contributions and Coverage Analysis

As the previous phase generates the code coverage for a failed test case, we aim to verify whether the covered files involved in the test case execution are also changed during the parent's contributions. So this phase is responsible for evaluating the code coverage and syntactic **parent-merge-diffs** information.

There are two types of changes we can evaluate about the classes changed by parent's contributions: class structure or methods. This observation is important since methods of a class can be exercised, and the parent's contributions did not change these methods but the class structure (information provided by syntactic diffs). For example, the type of an attribute, which is used in one method not changed by Left, can be modified during a contribution of Right. Our scripts do not capture such change nature, but they can identify that changes happen in the related class. We consider these changes can also reflect in direct or indirect impact in the test execution failure. All this information reflects the scope of source code involved in the test case execution until the failure occurrence. That means, the combination of these changes causes the failed test case.

For getting class structure changes, we consider any change done in a class, which can be identified through the original GumTree report. Although GumTree presents all changes performed between two files, it does not associate the changes with their methods. Nevertheless, to perform part of this analysis in this step, our scripts must know the changed methods. To solve this issue, we improve the GumTree tool report by adding the information of related methods into diff reports when it applies.⁹

After the test case execution, we download the coverage information from GitHub releases. The coverage file presents, for each Java class in the project, its set of methods (independently of exercised or not). For the exercised methods, a value different from zero (0) is assigned to the *coverage* attribute meaning that some part of the method is exercised. After parsing this file, we have only the list of files with their exercised methods.

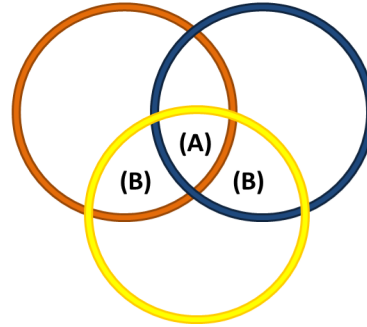
Considering the example of failed test case presented previously (see Figure 3.10), in Table 3.3, we present the covered and changed files involved in the test case. Despite many changes performed by each parent, only the changes in *SlangImpl* class are exercised in the failed test case, specifically, the method *getAllEventTypes*. So we identify a dependency between the parent contributions. Based in this idea, we analyse the coverage results and the files changes during parent's contributions looking for two types of dependency, as presented in Figure 3.11:

- The first possible dependency (A) looks for an intersection among the changed classes by the parent's contributions and the code coverage results (Figure 3.11). If (A) is not

⁹<https://github.com/leusonmario/gumtree/commit/b582cdb>

Table 3.3: Covered and Changed Classes involved in Test Conflict

Parent	Classes	Changes
Left	SlangImpl	getAllEventTypes (method)
Right	SlangImpl	getAllEventTypes (method)

**Figure 3.11:** Test conflicts analysis

Orange and blue circumferences represent the set of changed classes by each parent, while yellow the set of exercised classes by the test case. (A) Intersection among exercised and changed parents classes. (B) Intersection between exercised and changed classes for each parent.

empty, merge parents change same classes involved in the failed test case execution. For these classes, our scripts also verify whether changes on methods, class structure or both are involved in this potential dependency.

- The second dependency (B) lies on an intersection of exercised and changed files by a single contributor (Figure 3.11). In this case, if any (B) is not empty, we assume the associated parent performs changes on dependent files (called at any time during the test case). For example, Left can modify a method of class, which calls a second method from another class changed by Right.

Additionally, we verify whether changes happen in the failed test cases leading us to consider interferences between changed classes and test cases (new, changed or old test cases). For instance, a parent can change a test case structure, while the other updates methods exercised during the test execution.

We consider as test conflicts failed test cases of all nature (new, changed or old test cases). This observation is important since even if a new test case (included in the project by one parent during the merge scenario) fails, such failure is resulted of intersection between the parent's contributions impacting the software behavior as such failed test case presented *passed* status before the integration.

Unachieved Coverage Metrics

The other type of test conflict we consider is related to coverage metrics on source code

Figure 3.12: Build log of failed process due to unachieved metric

Build failed to meet Clover coverage targets: The following coverage targets for null were not met:

[ERROR] Total coverage of 93.3% did not meet target of 100%

[ERROR] Method coverage of 95% did not meet target of 100%

[ERROR] Statement coverage of 92.5% did not meet target of 100%

[ERROR] Conditional coverage of 94.4% did not meet target of 100%

that is done after the test execution. In this situation, the coverage metrics established for the project are not achieved. For example, in JJWT project, the coverage metric defined previously for the project is not achieved, as presented in Figure 3.12.¹⁰

For these scenarios, we do not run any test case since the conflict happens due to the not-achievement of the metric defined previously. To rerun the test suit would produce the same coverage results ensuring the test conflict.

3.5 Resolution Patterns Identification

To identify the resolution patterns adopted for build and test conflict fixes, and who is responsible for applying the fixes, we look for fix commits. As a fix, we consider a commit that is associated with a superior build status and immediately follows a merge commit associated with a broken build. In this way, a commit fix has as its parent the merge commit. Nevertheless, a commit fix may be a merge scenario (the fix commit has two parents, and one of them is the merge commit). For example, on the Okio project, we have the information of the merge and its fix commit presented in Figure 3.13. In this case, the fix commit [d07412c](#) has as one of its parents the merge commit [ffc28d6](#).

Since we have the commit fix, we check its build status. If the fix does not present a build on Travis, or its build status still presents the broken status, we try to identify the next commit using now the previous fix as the parent commit. We adopt this strategy because many attempts can be performed trying to fix the conflicts until the fix is achieved. In the same way, a conflict fix can be performed, but after that, other commits can be done. Consequently, the last commits will have a build on Travis instead of the commit fix. We do this process until we identify a fix commit with a build status that reflects a fix. For build conflicts, we consider as a valid build status *failed* or *passed* status. For test conflicts, we consider only the *passed* status as a valid one.

Once we identify the commit fix, we extract its info using Git API, and who is responsible for the resolution.¹¹ For that, we extract the name of the author commits (merge parents and fix commits) and check whether they are the same person. In our example, the developer is involved in the merge commit and also in the fix.¹²

¹⁰Build ID: jwtk/jjwt/280062604 - Merge Commit: 8cfc9f5

¹¹`git cat-file -p <commit>`

¹²We omitted the name of the developers involved in the commits due to ethical issues.

Figure 3.13: Commits information to identify commit fixes

Commit Fix: d07412c tree 1d0fcc756d746d3878e30106a7fa6ed239ae091a parent ffc28d67927d01c2900f3646710e564fc945cd7a parent a462d119ab0dd445022e9be00439acb268a113b3
Merge Commit: ffc28d6 tree ccf9ec544d232ff5f859d0a74cad8d8992722a61 parent 755bde66369d61a11432810dcda61a5043428068 parent 7a9dd8c8c9e0096ba89d9d21960d0eda70de3753

As we identify a commit fix, it is important to know this commit can hold not only changes for fixing the conflicts but also other changes not related. Nevertheless, our approach to verify the resolution patterns is based on the the conflicts causes, which we already know. In this way, we only analyze changes in the classes involved in the conflict. Changes not related with the conflict fix do not impact our approach.

Each kind of conflict has its approach to identify the resolution patterns, which are presented in next sections.

3.5.1 Resolution Patterns for Build Conflicts

After finding the fix commit, based on the build conflicts causes and the related information (see Table 3.2), we can analyze whether the involved classes in the conflict are changed. Consider as an example, a build from Quickml project that fails because of a reference for a method declaration that is no longer available, as presented in Figure 3.14.¹³

To check how this conflict is fixed, we apply GumTree tool for computing the changes between the merge commit of the broken build and its commit fix (**merge-fix-diff**).¹⁴ Based in the information extracted from Travis log (see Table 3.2), we analyze the involved classes looking for pre-defined resolution patterns. As can be seen in Figure 3.15, the call for the method (*missing symbol*) is removed from the *Main* class. This resolution patterns makes clear how the fix adopts the conflict to the new project state. Instead of reintroducing the missing method or even updating the method signature to its old one, the request is removed meaning such reference is not critical to the new project state.

As a conflict might be fixed using different resolution patterns, we further analyze the **merge-fix-diff** trying to identify other patterns. We identify resolution patterns checking manually some fixes leading to get its related message patterns (extracted from syntactic diffs). For that, we try to match different message patterns into the **merge-fix-diff**. The other resolution patterns we consider for this conflict are:

¹³Build ID: sanity/quickml/53571613 - Merge Commit : 62a2190

¹⁴Build ID: sanity/quickml/53575229 - Merge Commit: f98e5eb

Figure 3.14: Build log of errored process due to unavailable symbol

Compilation failure:
 [ERROR] /home/[...]/StaticBuilders.java:[67,116] cannot find symbol
 [ERROR] symbol: method ignoreAttributeAtNodeProbability(double)
 [ERROR] location: class TreeBuilder<quickml.data.ClassifierInstance>

Figure 3.15: GumTree diff for commit fix

```
//StaticBuilders.java

// Diff between Merge <62a2190> and Fix Commit <f98e5eb>
Delete SimpleName: ignoreAttributeAtNodeProbability(224) on Method
    getOptimizedDownsampledRandomForest
```

- **Missing symbol reintroduction:** in the *Secondary* class **diff**, we look for the addition of the missing symbol in the class using the message pattern:

```
Insert SimpleName: {missing_symbol_identifier}
```

- **Reference removal:** in the *Main* class **diff**, we look for the removal of the reference for the missing symbol using the pattern:

```
Delete SimpleName: {missing_ symbol_identifier}.
```

- **Reference update:** update of the reference (missing symbol) for the new symbol identifier in the *Main* class **diff** using the pattern:

```
Update SimpleName: {missing_ symbol_identifier} to
{new_method_name}.15
```

For cases our scripts could not identify the resolution patterns, we perform a manual analysis. The approach for the other conflicts are detailed in Appendix B.

3.5.2 Resolution Patterns for Test Conflicts

For test conflicts, we manually check for the adopted resolution patterns. For failed test cases, our analysis is based on two kinds of changes:

- **Update on test case:** We verify whether the test class and the failed test case are changed in the fix commit;
- **Exercised changes updated:** The second verification looks for changes in the set of classes and methods changed that are involved in the test case failure. These changes can be in methods as also in the class structure. In case of changes in methods, only

¹⁵The new method name is not known. However, the important aspect is that the missing method name is involved in an action of name update.

those involved in the failed test case execution are verified. For example, the methods presented in Table [3.3](#).

For test conflicts from dynamic analysis, we look for changes in metrics parameters as also on the source code.

4 RESULTS

Given the design presented in Chapter 3, we perform an empirical study to check the frequency, causes and resolution patterns adopted for build and test conflicts. This chapter details our results answering our research questions. Initially, we mine source code repositories from GitHub and build information from Travis CI. After the sample selection, we filter merge scenarios (60991 from 529 projects) to select conflicts and understand their causes. For the identified build and test conflicts, we investigate the adopted resolution patterns for fixing them.

4.1 Research Questions

Considering the motivation and strategy described in Chapters 2 and 3, respectively, we want to verify the frequency of build and test conflicts, as also to understand the causes, resolution patterns and conflict fixer. In summary, we investigate the next research questions:

- **Research Question 1 (RQ1)** - *Frequency*: How frequently do build and test conflicts occur?
- **Research Question 2 (RQ2)** - *Causes*: What are the structure of the changes that cause build and test conflicts?
- **Research Question 3 (RQ3)** - *Resolution Patterns*: What are the resolution patterns adopted to fix build and test conflicts?
 - **(RQ3.1) - Fixer**: Who does fix build and test conflicts?

4.1.1 RQ1: How frequently do build and test conflicts occur?

To answer **RQ1**, we measure the rate of broken builds resulting from build and test conflicts. After selecting merge scenarios and discarding non conflicts (see Chapter 3), we found 125 build conflicts and 39 test conflicts (7th row in Table 4.1). Analysing the number of merge scenarios associated to build and test conflicts in comparison to merge scenarios presenting broken builds and superior parents status (473 and 864, for errored and failed status respectively, 6th row in Table 4.1), we have a better notion of the relevance of the incidence of conflicts during development, by computing the proportion of build and test problems caused by conflicts. Under such perspective, the percentage of build and test conflicts are **24,73%** and **0,81%**, respectively.

Table 4.1: Broken builds caused by build and test conflicts.

Number of Projects	529	
Analyzed Merge Scenarios (MS)	60991	
MS with Broken Builds	16076 (26.36%)	
	7506 (Errored, 12.30%)	8570 (Failed, 14.05%)
Broken MS / Superior parents status	1337 (2.19%)	
	473 (Errored, 0.78%)	864 (Failed, 1.42%)
MS with Conflicts	117 errored builds (125 build Conflicts)	7 failed builds (39 Test Conflicts)

All percentages are calculated considering the total of merge scenarios. Analyzed merge scenarios (MS) represent all merge scenario we analyzed in this study (second row). MS with broken builds group only merge scenarios associated with errored and failed builds (third row).

Fifth row presents the rate of broken builds from MS with their parents presenting superior status. Seventh row represents the merge scenarios with broken builds caused by build and test conflicts. Note that more than one conflict can be associated with a broken build.

So, roughly, 1 in 4 build problem is caused by a conflict in errored builds, whereas the others are caused by a single developer mistake, configuration issues, unresolvable dependencies or unavailability of external services finishes the build process. As these other causes quite often happen in our experience and are reported in other studies [2] [8], we can conclude the occurrence of build conflicts might be quite relevant.

Based on these finding, we notice the percentage of test conflicts is low. Such observation is motivated by not only the nature of the tests (projects without good test coverage) but also limitations of our scripts. For example, in failed builds, some failed tests when executed alone do not present errors leading us to conclude the environment circumstances are responsible for the failure (resource unavailability or parallel tests execution). In other failed builds, despite the failed status, the Travis log does not report the failed test case impairing our analysis.

Considering all merge scenarios (60991, second row in Table 4.1), the percentage of build and test conflicts decreases to **0,19%** and **0,01%**. We believe the low number of conflicts in our sample is due to the use of Continuous Integration (CI) and Maven. With automated build and testing processes in these projects, developers can easily build and test their contributions before sending them to the main repository. They are often, by project guidelines, required to locally merge their contributions with the current main repository contributions before submission for approval or final integration. This way, we assume most test and build conflicts are actually detected and resolved locally, in the contributor's private repository.

As our study analyzes only the main public repository, we do not have access to problematic merge scenarios that were locally amended before reaching the main repository. Consequently, ours numbers reflect the number of conflicts that reached main repositories, not the actual number of conflicts that happened and had to be resolved. This justifies the high percentage of scenarios with successful build process. This might be also a consequence of

developers practices such as committing early and often, which often results in an increased number of merge scenarios with small contributions that are unlikely to conflict and change build status.

As can be seen, more than one conflict can be associated with a merge scenario; in our results, 125 build conflicts for 117 errored builds and 39 test conflicts for 7 failed builds (seventh row in Table 4.1). In case of build conflicts, the build and compilation phase of a build process only stops the process in the end of the phase. It means, even if during the phase problems occur, the phase still continues to execute until its end. Only when the phases ends, the problems are reported, and the build process cannot continue (broken build). For test conflicts, a test suit is composed of many test classes. In the same way, the failure of one case does not impact the testing phase until it ends.

Although we use automatized scripts for the whole study, some scenarios of build conflicts could not be automatically evaluated due to lack of information or limitations in our scripts. For these special scenarios (42 cases, 33,6% of the analyzed scenarios), we perform a manual analysis. In the end, the conflicts were classified as Unavailable Symbol due to Project (22, all related to unavailable files), Unavailable Symbol due to Dependency (1), Unimplemented Method (3, all of related to Super Type), Incompatible Types (3) and Project Rules (13), represented in Table 4.3 with an asterisk (*). All build and test conflicts identified are available in Appendix E.

In test conflicts, we notice failed builds (18 cases of different merge scenarios) that should be errored process. Such problem happens when a project is composed of modules, and only some are involved during the build process. Therefore, when test cases, involving those modules not built yet, are executed, these modules are built occurring unexpected build conflicts. Despite such anomaly, we do consider such cases, counting them separately as build conflicts (numbers presented in Table 4.3 between brackets). All cases are available in Appendix E (Table E.2).

Thus, we conclude that build and test conflicts happens less in our sample than compared to related work. We believe these conflicts happen more than reported by our results, but they are fixed in developers workspace before they are sent to GitHub. Furthermore, more than one conflict might be responsible for breaking a build process of a merge scenario.

4.1.2 RQ2: What are the structure of the changes that cause build and test conflicts?

Based on conflict occurrence, we proceed with further analysis to answer **RQ2** identifying the associated causes. Table 4.2 shows the causes and their descriptions grouped by cause categories and type of conflicts (third and fourth column, respectively). Additionally, Tables 4.3 and 4.4 present the frequency for each cause associated with each kind of conflicts. The list of causes in the table corresponds to the causes identified in our sample. Enlarging the sample likely will not reveal new causes because, as explained previously, in open-source projects, developers might treat the conflict occurrence before sending contribution to the final central repository.

Table 4.2: Causes of build and test conflicts

Conflict Type	Cause	Name	Description
Build Conflict	Syntax	Malformed Program	Files do not present valid program
	Static Semantics	Unimplemented Method	Class does not implement a method associated with an interface
		Duplicated Declaration	Two or more elements declared with the same identifier
		Unavailable Symbol	Reference for a missing element in the source code
			Reference for an element not provided by the project dependencies
		Incomp. Method Signature	Reference for a method with wrong number of parameters or types
		Incomp. Types	Type mismatch between parameter and argument, or between result type and expected type
	Other Static Analysis	Project Rules	Files do not follow the projects guidelines
Test Conflict	Dynamic Semantic	Test Failures	Passed test cases fails due to contribution integration
	Other Analysis	Unachieved Coverage	Coverage result does not achieve the expected level for the project

Table 4.3: Distribution of build conflict causes by category and motivation

Changes Source		Build Conflict Causes											ASAT	
		Syntax	Static Semantics								Incom. Meth. Signature	Incom. Types		Project Rules
			Malformed Expressions	Unimplemented Method		Duplications	Unavailable Symbol		Project	Dependent				
Contributor	Pres.			Project	Dependent		3* (2.4%)	2 (1.6%)			28* (22.4%) [15]			5 (4.0%)
	Integrator	Unpres.						21 (16.8%)			1 (0.8%)	3* (2.4%)	13* (10.4%)	
Total		Pres.		2 (1.6%)		1 (0.8%)	9 (7.2%) [1]			5 (4.0%) [1]				
	Total	Unpres.	6 (4.8%)	4 (3.2%)		3 (2.4%)	8 (6.4%) [1]	1* (0.8%)	3 (2.4%)					
Total		Unpres.	6 (4.8%)	6 (4.8%)	3 (2.4%)	6 (4.8%)	66 (52.8%)	1 (0.8%)	14 (11.2%)	4 (3.2%)			19 (15.2%)	

The conflicts are grouped in two perspectives of causes: contributor and integrator changes. For each perspective, merge scenarios are classified in preserved (Pres.) and unpreserved (Unpres.) scenarios. * indicates some cases of such category were checked manually. Numbers between brackets ([]) indicate build conflicts from failed builds.

Most build conflicts are due to Unavailable Symbol

The most frequent cause of build conflicts is *Unavailable Symbol* (52,8% that represents a reference for a missing element in the source code). Such missing symbol can manifest in different ways, as also involving one or more classes. Among the *Unavailable Symbol* causes, the most recurrent missing symbols are classes taking 38,4% of all build conflicts. Unavailable Method (5,6%) and Variable (8,8%) are cases that the missing symbol and its associated reference can be in the same class as also in dependent ones. For example, in the Okhttp project, the build breaks because of changes to an available parameter.¹ Both parents change the method *read*. Left removes the parameter *deadline* and all references for it, while Right adds a new verification using this parameter. Since the changes are not in the same file area, no merge conflict happens. As result, the symbol *deadline* introduced by Right is neither declared as a parameter nor declared as a local variable, but is referenced in the method body breaking the build process.

The second most recurrent cause takes 15,2% of all build conflicts, and it is not motivated by a problem during the build and compilation phase but due to a nonconformity of *Project Rules*. This particular case do not involves direct or indirect dependencies among classes as the project style conventions must be followed for the whole project.

Duplicated declaration is another conflict cause involving only one class. For example, in Web Magic project, both parents add the *main* method declaration in the *HuxiuProcessor* class in the same file region. Despite the occurrence of a merge conflict, all parents contributions are preserved (two methods with the same signature) leading the build process to break.² A similar case happens in the Blueprints project, when two test cases are declared with the same signature (*testRemoveNonExistentVertexCausesException*) in the test class *GraphTestSuite*.³ Comparing the source code of the duplicated methods, in both cases they present the same structure allowing us to conclude one parent copied it from another resulting in the broken build when integrated.

Integrator changes do cause Build Conflicts

Although most build conflicts are caused by contributors changes (66,4%), integrator changes performed after the integration are responsible for 33,6% of all build conflicts. In this situation, the number of conflicts on unpreserved merge scenarios is higher (20%, 8th row in Table 4.3) than on preserved ones (13,6%, 7th row in Table 4.3). This frequency of build conflicts caused by integrator changes is an evidence that related work do not evaluate.

Besides to identify these new cases of conflicts, we also investigate the motivation behind them. Such cases occur when integrators change the merge integration result before committing it. These changes might be motivated by merge conflicts or even the addition

¹Build ID: square/okhttp/19399475 - Merge Commit: 9dfeda5

²Build ID: leusonmario/webmagic/243501513 - Merge Commit: a2fba8c

³Build ID: leusonmario/blueprints/267833702 - Merge Commit: 5a25e3a

of new contributions. This confirms our experience that fixing conflicts of contributions of other developers is often challenging and an error-prone activity. For example, in the CorfuDB project, the *AbstractViewTest* class has the method declaration *getDefaultRuntime* requiring no parameter.⁴ During parent's contributions, both parents change such method. Left changes the method body, while Right adds a parameter and introduces another method with a different signature but without requiring parameters. Additionally, Right also add a reference for the parametrized *getDefaultRuntime* of *AbstractViewTest* class in *ObjectsViewTest* class. Since the parents change the same file area of *AbstractViewTest* class, merge conflicts arise. To fix the conflicts, the integrator keeps only the Left contributions discarding Right changes for this specific class. However, all Right contributions for *ObjectsViewTest* class are preserved in the merge scenario as only Right changes this class. As result, the reference for parametrized method *getDefaultRuntime* in *ObjectsViewTest* class fails as the available method with such name in *AbstractViewTest* class has no parameter. In this case, the actual method cannot be applied representing a case of Incompatible Method Signature.

Concerning about the preservation of parent's contributions in merge scenarios with build conflicts, Table 4.3 presents the frequency of conflicts grouped under two perspectives: contributor and integrator changes. In our sample, 49,6% of all build conflicts happen in preserved merge scenarios (all parent contributions are preserved) than on unpreserved ones taking 50,4% of all cases (at least one parent contribution is discarded). Despite the percentage for each merge scenario perspective, it shows how often parent's contributions are discarding during merge integration. Most of these cases are motivated because of merge conflicts fixes, which lead integrator to introduce inconsistencies in the source code.

We also notice that more than one conflict causes might be associated with a single errored build. For example, in the Pac4J project, the build breaks because of two distinct causes: *Unimplemented Method* and *Unavailable Symbol*.⁵ In this case, the integrator changes are responsible for the failure. So the total numbers we present in the Table 4.3 are actually greater than the sum of errored builds associated with merge scenarios.

In the DSpace project, a similar case happens in a preserved merge scenario.⁶ This time, a merge commit build breaks because of three causes: *Incompatible Method Signature*, *Incompatible Types* and *Unavailable Symbol* (class). Nevertheless, only the first cause is motivated by the integrator changes, while contributor changes are responsible for the other causes. Interesting in this case is how the integrator introduces the inconsistencies. The attempt to use a method passing a different parameter list is introduced only in the merge commit. The class containing this reference is not even present in the merge and base commits. Curiously, the class holding the method reference is also introduced during the merge commit making clear the integrator confusion.

⁴Build ID: CorfuDB/CorfuDB/147270257 - Merge Commit: d4f1845

⁵Build ID: leusonmario/pac4j/291006785 - Merge Commit: 827b7d8

⁶Build ID: leusonmario/DSpace/263379307 - Merge Commit: 049eb50

Table 4.4: Distribution of test conflict causes by category

Changes on Methods	Test Conflict Causes			
	Failed Tests			Unachieved Coverage
	New	Old	Changed	
Same		1 (2.57%)		1 (2.57%)
Dependent			2 (5.12%)	
Both	8 (20.51%)	21 (53.85%)	6 (15.38%)	

Test conflicts caused by failed tests are classified as *new*, *old*, and *changed* test cases (second, third and fourth columns, respectively). For each failed test case, the files changed by the parents and involved in the test execution are classified based on the type of changes. Parents might changer *same* and *dependent* files, or *both* type of changes (fourth, fifth, and sixth rows, respectively).

Most Test Conflicts are motivated by Dependent Changes

As presented in Table 4.4, most test conflicts are caused by failed test cases (97,43%, 2nd-4th column). In such circumstances, almost 90% of the parent's contributions change same or dependent classes, which are involved in the failed test case execution. It means, the intersection of changes directly impact on the software behavior. Among the failed test cases, we observe difference related to the nature of the test cases. For example, most failed cases in test conflicts happen due to old test cases (56,42%, third column). In the other side, test conflicts are also caused by updated test cases (20,5%, fourth column) or even new ones introduced by one of the parents during the merge scenario (20,51%, second column). Such perspective makes clear how different contributions affect each other even for a new test case included in the project.

An example of a test conflict happens in the Wire project. Left parent updates the test case while Right updates some of the methods exercised by the test case.⁷ Although the changes are not in the same file, when integrated, they result in a test conflict. Another case happens in the Cloud Slang project when the parents change the same method.⁸ This case makes clear how the parent contributions worked without problems, but after the integration, the software behavior changes leading to a test conflict.

Additional analysis performed after the testing phase can also bring test conflicts. In this case, an *Unachieved Coverage* metric represents a conflict (2,57%, fifth column). It is an unexpected cause since it involves the results of the whole test execution phase. After the merge scenario integration, the metric coverage is bellow from the expected result.

Before commenting on specific test conflicts, it is important to mention many builds present failed tests cases but our scripts could not extract the information about the failures. Although we adopt only projects using Maven as build manager and instrument our script from Travis logs, the message patterns just like in build conflicts may change impairing our extraction.

⁷Build ID: square/wire/81124823 - Merge Commit: 6664824

⁸Build ID: CloudSlang/cloud-slang/75814949 - Merge Commit: 20bac30

However, the patterns we already identified, they can be applied to adopt other build managers in future experiments. Not necessarily the patterns could be reused, but they can guide the process of inclusion since common failed test cases are reported in a similar way (test class with its failed test case). In other cases, the problem is not in our scripts but in log report. For example, in the Clocker project, the log report informs an error happens during the testing phase but neither informs the file nor the associated test case.⁹ So our results are actually a rough lower bound on the number of test conflicts. For replications of our study, our scripts must be probably updated to compute new patterns of failed test case messages.

Besides this issue with log files, we also potentially miss test conflicts because, in a number of scenarios, we could not properly instrument the code to obtain coverage information. After instrumenting the code, we obtained compilation problems that break the build process. In the Jedis project, for example, 4 different scenarios could not be evaluated due to unexpected problems during the build and compilation phase. Since we remove some instructions of *travis.yml*, it is possible such removal can be responsible for preparing the environment before the build process start. However, after such removals, we had progress with the test conflicts that we present here. In other case, we could successfully generate the instrumented build, but none failure happens in the previous failed test case. Such cases brings evidences that depending on environment conditions and resource unavailability, test cases fails only because adversed situations (not an unexpected software behavior). For instance, we observed that in the Blue Food project.¹⁰

4.1.3 RQ3: What are the resolution patterns adopted on build and test conflicts fixes?

After finding the conflicts and their causes, we identify the commits and respective builds responsible for fixing them. Although all evaluated scenarios are from active projects, not all conflicts present a fix commit associated with a build in Travis. In fact, fix commits could be done, but following the fix, new contributions are performed. Thus, when the contributions are sent to the main repository, no build is started for the fix commit as Travis starts a build process only for the most recent commit of a *push*. Consequently, the last contributions are built on Travis instead of the fix commit.

As motivated previously, only 43,2% of all build conflicts (54) are fixed and present a build in Travis for the fix commit. These fix commit builds are important because they indicate whether a conflict is truly fixed. For example, only a fix commit associated with a passed build can be considered as a fix for a test conflict. For these commit fixes, our scripts analyze them to identify the resolution patterns adopted. In the end, 29,6% of the resolution patterns related to fix commits were identified automatically (16) by our scripts. The other resolution patterns (66,6%) were identified by manual analysis (36). In two cases of the Databind project we could not identify the fix pattern. In test conflicts, 57,14% of all cases are fixed, which represents 4

⁹Build ID: brooklyncentral/clocker/124410760 - Merge Commit: c13ed22

¹⁰Build ID: leusonmario/blueflood/315017416

Table 4.5: Resolution patterns for build and test conflicts

Conflict Type	Category	Name	Solution
Build Conflict	Syntax	Malformed Program	Syntax Update (100%)
	Static Semantics	Unimplemented Method	Unimplementable method implementation (50%)
			Interface method removal (25%)
			Interface method update signature (25%)
		Duplicated Duplication	Removal of one duplicated declaration (100%)
		Unavailable Symbol	Reference removal for missing element (37.04%)
			Reference identifier update (22.22%)
			Import update (22.22%)
			Dependency update (11.11%)
			Missing element reintroduction (7.40%)
		Incomp. Methods Signature	Method request update (50%)
			Method request removal (25%)
			Method signature update (12.5%)
			Unsupported method reintroduction (12.5%)
		Incomp. Types	Type adaptation update (100%)
	Other Static Analysis	Project Rules	Adjust for style convention (100%)
Test Conflict	Dynamic Semantic	Test Failures	Test case update (25%)
			Changed same methods update (50%)
			Undo all changes (25%)

cases also checked manually. We now present the adopted resolution patterns for build and test conflicts.

Resolution Patterns adapt the Project to its new state

The adopted resolution patterns basically adapt the project to the parent's contributions intention instead of discarding them and returning to an old project state (Table 4.5, fourth column). Such behaviour is present in 65,38% of all fixes. For example, in the Quickml project, the reference for the missing symbol *ignoreAttributeAtNodeProbability* in the *TreeBuilder* class was updated to the new method signature *attributeIgnoringStrategy*.¹¹ In the other hand, the fix adopted in the Cloud Slang project does not follow the same practice.¹² Instead of the *PreCompileValidatorImpl* class implementing the new method *validateResultName* from the

¹¹Build ID: sanity/quickml/53575229 - Fix Commit: f98e5eb

¹²Build ID: CloudSlang/cloud-slang/158196252 - Fix Commit: 8e9226f

PreCompileValidator interface, such method was removed from the interface.

In test conflicts, the changed methods associated with the test case failure execution are also changed bringing evidence these methods are genuinely involved with the failure. This happened in the Jedis project when the *BinaryJedis* class is changed.¹³ In another case, no additional change was performed to fix the conflict. The solution adopted was to come back to a previous commit (without test failures) discarding all changes.¹⁴ In the Cloud Slang project, the fix adopted was to update the test case instead of the parent's changes (changes on same methods).¹⁵ For test conflicts motivated by unachieved coverage metric, we do not identify any fix commit.

Most fixes are done by parent commit authors

Once the resolution patterns for build and test conflicts are identified, we investigate who was responsible for the fixes. Most fix commits are done by integrators, who are also one of the contributors involved in the merge scenario. For build conflicts, 88,46% of all fixes follows such pattern. Such finding makes us think the motivations behind such high numbers. We believe build and test conflicts are even worse when compared to merge conflicts forcing the contributor to deal with them.

For test conflicts, all 4 merge scenarios with test conflicts are fixed by developers involved in the associated merge scenarios. Such situation can be also motivated by the difficult behind dynamic semantic problems.

4.2 Discussion

In this section, we discuss our results and implications, how they can be applied in solutions, and also future work.

4.2.1 Conflicts are recurrent

Comparing our results with related studies, we conclude build and test conflicts happen less than reported in the literature (0,19% and 0,01%, respectively). Kasi and Sarma [2] show build and test conflicts occurrence ranges from 2% to 15%, while test ones range from 6% to 35%. In the same way, Brun et al. [8] present the occurrence of build and test conflicts together ranged around 33%. This discrepancy can be justified for some limitations and bias in the related studies. Related studies consider only merge commit builds to identify conflicts without also checking parent builds. They also try to build the merge commit locally, consequently, any change on environment configuration breaks build process leaving them to wrong conclusions. The threats of our work is discussed in Section 4.3.

¹³Build ID: xetorthio/jedis/44295865 - Fix Commit: 46a34f2

¹⁴Build ID: xetorthio/jedis/59489279 - Fix Commit: a52902b

¹⁵Build ID: CloudSlang/cloud-slang/75925280 - Fix Commit: d67e404

Our findings bring not only information of build and test conflicts frequency from clean merge scenarios but also unclean ones. Related studies consider only merge scenarios without merge conflicts. Most build conflicts, 66,4%, arise from contributor changes (36% and 30,4% for preserved and preserved merge scenario, respectively). In the other side, conflicts caused by integrator changes is also expressive (33,6%, 13,6% on preserved merge scenarios, and 20% on unpreserved). Build conflicts caused by integrator changes are due to merge conflict occurrence requiring human intervention. Consequently, the integrator tries to fix the merge conflict but introduces inconsistencies in the project.

Related to the build conflict causes, our results conform with the findings reported by Seo et al. [17]. Although our results focus on specific cases of build errors caused by build conflicts, Unavailable Symbol is the most recurrent cause of build errors in both studies. This reveals the common mistake of removing or updating declarations, but possibly not updating all references to the new identifiers. Assistive tools could be applied to anticipate the emerging conflicts, or even treat them directly. In this way, they could alert during the tasks development that a new reference for a symbol will fail during integration since the referenced symbol is removed by other contributor (in case of build conflicts due to contributor changes).

Only conflicts caused by contributor changes could be anticipated by analysis of developers workspaces. Such assumption has a direct relationship with the fact that most build conflicts involve different files (changes spread on dependent files), and new test cases may fail and lead to test conflicts. For example, Assistive tools like Palantír [28] that consider developers workspace as source information for predicting conflicts must be aware of dependent files independent of when changes are performed.

Build conflicts are not restricted to happen involving only dependent files (one file references another). Build conflicts of *Malformed Program* and *Duplicated Declaration* involve only one single file. Investigating manually some cases, we verify such conflicts arise from merge scenarios with merge conflicts. The occurrence of merge conflicts reflect how they indirectly motivated other types of conflict, as also the weakness of merge tools to deal with them (textual ones). The *Duplicated Declaration* cases could be actually detected by merge tools like S3M [9]. Teams using this tool reduce the number of merge conflicts that truly are false positive. For example, two different method declarations declared in the same file are reported as a conflict, while it is not. As commented previously, common merge tools bring many false positives leading the integrator to deal with them. Better merge tools could decrease the number of false positives, as also, avoid the unnecessary human intervention preventing build conflicts. We explain in details application of our results in the next sections.

Build conflicts from unpreserved merge scenarios bring evidence that approaches for automatically resolving conflicts could be applied independently of parent's contributions preservation in merge scenarios. For build conflicts, despite the distinct source of the causes, conflict fixes are similar. For test conflicts, our scripts already identify the methods and files involved in the failed test case. Thus, if these methods come from developers or integrator changes is

irrelevant since a support would be applied to this restricted set of changes.

4.2.2 Findings and Implications

Based on our results, we present here insights of how such knowledge could be used to not only avoid build conflicts but also treat them. In this way, we propose improvements and new features on existing assistive tools.

4.2.2.1 Awareness Tools

Awareness tools provide developers with information about the work of other developers in the same project [48]. Many studies have proposed tools aimed to achieve such goal differing each other in the way and applied awareness degree. Biehl et al. [49] propose a dashboard showing edits performed by each developer. Despite the simplicity, developers must regularly check the board because the report is shared on a screen for everyone. The idea is anyone can be able to identify an emerging conflict and contact the developers interested. Also in this way, Hattori and Lanza [50] present the Syde tool able to inform change and conflict information across developer's workspaces.

Even if these tools focus on merge conflicts and not specifically on build and test ones, they could also identify emerging build conflicts for contributions performed on the same file, especially in the same region. For example, *duplicated declaration* of variables with the same identifier on the same scope is an example of build conflict that could be identified unexpectedly by these tools. In such circumstances, build conflicts would be only verified whether developers actively used the tool.

Nevertheless, the causes associated with changes across dependent files could not be detected by such tools. In this way, Sarma et al. [28] propose a tool able to detect emerging build conflicts when developers edit, simultaneously, dependent files. Our findings could be used for covering new conflict causes not explored yet. For example, based on the *Incompatible Method Signature* cause, awareness tools could alert a developer the method referenced by her has been renamed by another developer. Thus, developers could already adapt their contributions or wait for when the modifications had been finished, and they could continue the work. In the same way, *Unimplemented Method* addition in interfaces could be supported requiring the implementation of a new method by its new implementing classes.

It must be taken into consideration that applying this tool for all causes of conflicts could distracting developers due to the high number of alerts and false positives. For the causes presented previously, the tool could be applied but presenting different intensity of alerts. For example, alerts for *Unimplemented Method* must be stronger than for *Incompatible Method Signature* since the reference for a method can be removed but the implementation of a new interface method must be implemented.

The support could also extend for dependent changes on the same file. For example, a

developer references a method/variable while another developer changes the type of the variable or one of the methods parameter.

4.2.2.2 Automatic Repair Tools

Automated programs repair tools propose and select the better patch to automatically fix failed programs without human effort [51]. The process works in three steps:

- Fault localization, the motivation behind the problem;
- Patches creation, different solutions can be applied to the same problem;
- Patch validation, a patch can be considered as valid if it can solve the fault.

Although many studies have explored different areas of semantic problems [52] [53], build conflict causes represent an opportunity for further studies, as also to support known tools for test failures. In the following sections, we present how our results could be applied in this area to deal with build and test conflicts. For build and test conflicts, our scripts identify the files involved in the conflict. This information reduces effort to the whole tool since the first step of an automatic repair tool is already done.

Build Conflicts

For build conflicts, an tool could explore two categories, related to the causes associated with the conflicts: (i) automatic and (ii) semi-automatic tools. Although automated program repair works without human intervention, in the second tool category the user decides the most appropriated patch to solve the problem. This constraint is essential once different solutions can involve different semantics. We now specify each category, and how some build conflicts would be treated by them.

Automatic tools represent genuine repair tools. Based on the causes, they could repair directly the conflict by changing the source code. For instance, for the *Unimplemented Method* cause (one implementing class does not implement a method introduced in its related interface), the tool could identify such method implementation and use this implementation. Since a new method is introduced in one interface, all associated classes with such interface must present an implementation for this method. Thus, the tool could identify this implementation and replicate it in the class that causes the build conflict.

In the same way, *Unavailable Symbol* causes could be treated but with some restrictions. If the missing symbol was renamed in the Left parent, the tool could rename references to the old name in the code of Right. This information our scripts already identify and could support this step, which reflect in less effort. The same approach could be adopted if the symbol were removed. In this case, the tool would reintroduce everything that has caused the build conflict (removed by one of the parents). Missing class declarations must have particular attention since it

is possible they had just moved the files. Instead of reintroducing the file, which would introduce inconsistencies in the project due to duplications, the tool would update the old import targeting to the new file location.

The semi-automatic tool is able to identify more than one solution and require human intervention to apply the solution. Although more than one solution could be applied to result in a successful build, the way each solution is implemented would be directly depending on the parent's contributions. For example, two parents can declare two methods with the same signature, but each one is responsible for different intentions. In this case, it does not make sense to remove one of the methods since it performs a required work.

Thus, for *Duplicated Declaration* of methods, the tool would present different options for resolutions:

- Removal of one duplicated method;
- Both implementations are kept, and a method signature update is done in one of the duplicated methods, as also its call;
- Attempt to merge both implementations into just one method like semi- and structured merge tools work.

Once the conflict is fixed, the tool would run the compilation and build phase from the build process aiming to ensure the solution adopted fixes the conflict. If such phase presents a successful status, it means the conflict is fixed and the changes can be saved in the local repository as a new commit.

Test Conflicts

Duong et al. [52] present a tool able to generate a valid repair for failed suit tests. In the same way, Tan and Roychoudhury [53] also propose an approach to repair software regression based on the concept of reconciling problematic changes. To identify the changes responsible for the fail, a first step is responsible for getting the diff of changes between two version files. Based on such information, they try to combine changes, eliminating others until to find the contributions responsible for the failure.

Our results about changed methods involved in the failed test case could support the process of fault localization of both tools. To identify the changes responsible for the failure, they consider the difference between two files. However, as we verified, not all changes involved in parent's contribution are directly associated with the a failed test. Our scripts could support this process informing only the files involved in the failure as also changed by the parent's contributions. We believe such approach implies in less effort since the scope of changes is smaller than considering all changes performed in two version files.

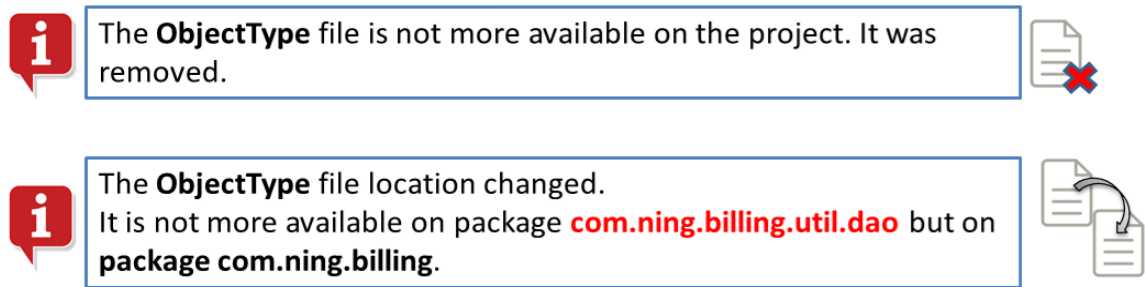


Figure 4.1: Improved messages for different unavailable symbol types

4.2.2.3 Better Guidelines Support for Developers

Build Conflicts

Travis build logs report how the build process works. However, for some errors, the messages are not clear enough to understand the causes of the broken builds, which could lead developers to spend more time than necessary. For instance, *Unavailable Symbol* file can be caused by the removal of a file or just a location change, but the log simply reports the file is not more available. Despite the information veracity, it can lead developers to think the file was removed, and not consider the file was moved to another directory. As a result, he could put the file on the old location introducing inconsistencies in the project.

Better support could report two different errors: *Unavailable Symbol* due to removal, and file location change. For the last one, it could also inform the new file location (Figure 4.1). Such improvement on the log report could be done by the build managers since they are responsible for presenting how the build process works, but also by Travis. For example, after a broken build process, Travis could internally evaluate the problem and present its own report for the broken build cause.

Test Conflicts

In failed builds caused by failures in test cases, Travis log presents the related test file and the set of associated failed test cases (see Figure 3.10). Like the report for compilation problems, the message is too superficial leaving the developer without any support to identify the causes of the failures. The common practice is debugging the code until the failure cause is identified. However, this approach can lead developers to spend time on unnecessary tasks since not all exercised methods during the execution are changed by parent's contributions.

Experienced developers could look only for the code changed during the merge scenario. Although this information can be accessed by git using refined commands, this process can be difficult for git beginners. On the other way, even if such improvement reduces the scope of work, we already verified not all source code changed in a merge scenario is involved in a failed

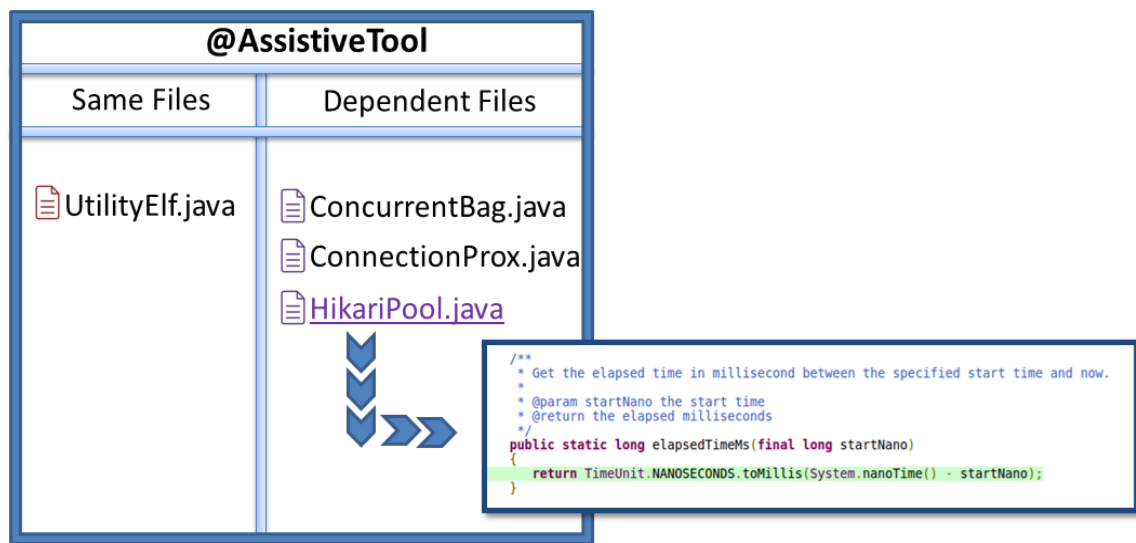


Figure 4.2: Prototype of assistive tool for test failures

test case. Therefore, developers would also be spending time on unnecessary tasks.

A tool could inform developers only the files (i) changed during the merge scenario and also (ii) exercised by the failed test case. Our scripts can support such improvement since we perform the analysis responsible for getting the required information. In the end, the changes would be presented based on same and dependent changes. The screen of such possible tool is presented in Figure 4.2. A click in a file would present the exercised methods associated with such file. A double click would open the file in the developer workspace.

For now, our scripts only present the information on spreadsheets. Thus, it is necessary to implement the user interface and the connection between the tool and the developer workspace making this process of failure identification more natural and easy to perform.

4.2.2.4 Better Merge Tools

Merge tools primordially work using text difference between two files (textual tools). KDiff3 [54] is one of the most used algorithm applied in merge tools. Although such approach has been largely used, it introduces many false positives of merge conflicts leading integrators to deal with them. If two different methods are introduced in the same file area, the merge tool would not merge the contributions requiring human intervention. As reported by our findings, the process of solving these conflicts can introduce build conflicts, and consequently, additional time spent on a problem that could be avoided.

Semi- and structured merge tools [9] [29] [7] are examples of tools that consider the source code elements as nodes and treat each one separately reducing the incidence of merge conflicts. Considering the previous example, these tools are able to merge the contributions preserving both methods. On the other hand, they could also anticipate the treatment for some build conflicts like *Duplicated Declaration*. In this way, duplicated methods on a class would be

identified already in the merge scenario forcing the integrator to treat the problem and avoiding an emerging build conflict. However, in some cases, only structured tools would be able to detect conflicts. For example, duplicated variables on the same scope are only taken by structured tools since semi-ones treat the internal code of an element as normal text (merging using a conventional merge algorithm, textual merge).

4.3 Threats to Validity

Our empirical study leaves a set of threats which we explain in detail.

4.3.1 Construct Validity

Few cases of build and test conflicts come to GitHub and Travis impacting negatively our results. Zhao et al. [55] identify such behavior. In their study, they verify a decreasing trend in builds with errors caused by missing files/dependencies (unavailable symbol) after projects adopt Travis. In this way, we believe more conflicts happen, but they are fixed before being sent to the main upstream repository. An experiment considering private repositories could bring more cases of build and test conflicts than compared to open-source since reputation and code accessible for everyone could reflect in a cleaner work. However, even in such situations conflicts could be locally fixed. Instead of using data from commits performed in the past (retroactive data), the best way to identify such conflicts would be having access to developers workspaces instantaneously evaluating the cases without any external influence (integrator changes).

Another threat of construct is related to our metric of frequency. In Travis, a build can be composed by a set of jobs; each job varying in environment configuration, or in the way the build process must be performed. Different jobs can be used to simulate the same project with different environment configurations. Therefore, it is possible to declare which jobs should not be considered for the final build status. In case a conflict happens in a non-valid job, we lose these scenarios. For future work, during the build process of commits not built yet, we could edit *travis.yml* file aiming to consider all jobs for the final build status. However, if the job is not considered as valid, we can assume non-valid jobs are used only to verify how the project behaves on a specific configuration, which leads us to conclude its result is not relevant enough for the project. Hence, problems in such jobs do not demand time of developers to fix them.

Moreover, our sample is composed of projects that use textual merge tools for dealing with merge conflicts. In this way, one can argue that using better merge tools would reduce the number of conflicts we found. However, not all kinds of merge conflicts can be treated by these improved tools leaving cases that would require human intervention also resulting in possible conflicts. Although they remove some false positive merge conflicts, they also introduce new false positive and false negative cases, which could lead integrator to deal with them, and possibly, introduce inconsistencies breaking the build process. To run this study with improved merge tools is a future work we plan to execute.

4.3.2 Internal Validity

A first internal threat is related to our study subjects since we decide to build some merge scenarios not previously built. This approach increases completeness but also introduces limitations. Despite building commits from the main upstream branch, some projects specify a list of exclusive branches for having permission to perform builds. Once all pushes done from our study are performed from the main branch, if such branch is not part of this restricted list, no build process can be started. If conflicts happen in such merge scenarios, we are losing them. In future work, we could check on the *travis.yml* file which branches have permission and use them for sending the *pushes*.

Like related work, we analyse Git projects that supports commands such as *rebase*, *squash* and *cherry-pick*, that rewrite project development history. Consequently, if a project adopts such practice, we may have lost merge scenarios with build and test conflicts. Specially build conflicts caused due to changes made by the integrator since merge conflicts that were treated by the integrator would not appear in project history [56]. Thus, our results are actually a lower bound for build and test conflicts.

Still in this context, project configuration is another issue because project dependencies might not be available leading the build to an errored status. If a project has a dependency for another GitHub project, and this last is not more available, the related build process will fail. If conflicts happen in these scenarios, we are also losing them. However, we selected active projects that makes us conclude they are evolving and also using dependencies, currently, available.

In a build process, one can specify failures in certain tests should impact the final build status. Consequently, if test conflicts happen in such scenarios, we are losing these cases because the first information we look is the final build status. If a build has *passed* status, we discard this scenario and do not perform any analysis in its Travis log report.

Another limitation arises when projects discard Travis CI, and after some time, they start to use it again. Since we filter merge scenarios from the date of the first build performed on Travis, if any conflict happens during this time of non-Travis use, no conflict will be identified. It happens because if we try to build them, all resulting builds will be broken even if Travis is using the default configuration for a build process. Thereby, we decide to build only commits with the presence of the *travis.yml* file in the commit contents.

Like related work, we analyse Git projects that supports commands such as *rebase*, *squash* and *cherry-pick*, commands that rewrite project development history. Consequently, if a project adopts such practice, we may have lost merge scenarios with build and test conflicts. Specially build conflicts caused due to changes made by the integrator since merge conflicts that were be treated by the integrator, and they do not appear in project history [56]. Thus, our results are actually a lower bound for build and test conflicts.

We analyse build logs looking for specific message patterns associated to problems that

cause the broken build. After a pattern is identified we perform additional analysis as explained in Chapter 3, to ensure the problem is caused by a conflict. During this analysis, if a problem is reported using a pattern our scripts do not know, we are losing these cases. However, for each message pattern we identified variations allowing us to increase completeness and eliminating eventual problems not supported.

We use GumTree diff to verify the contributions performed by developers and integrators. However, our approach has some limitations. For example, when we look for a specific method in the diff of a class, 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 signature. When we try to identify build conflicts caused by contributor changes, we verify the conflicting contributions looking for the method names in the syntactic diffs, which can lead us to a wrong conclusion. For example, if both parents introduce two methods with the same name but different signature, and in the merge commit there are two methods with the same signature, the conflict is motivated by the integrator changes and not by contributor ones. To verify such possibility, we manually evaluate all cases of *Duplicated Declaration* methods revealing all cases were well classified.

However, this problem is not a threat for *Unavailable Symbol* method. For instance, if a class has two methods with a different signature, but same names, and one of these methods are removed/updated, a reference for the removed/updated method would lead to *Incompatible Method Signature* instead of *Unavailable Symbol*. This case happens because the remaining method would be available in the class, and an attempt to use it would be performed. In the other hand, our approach for *Incompatible Method Signature* cause does not have this threat since we verify the set of methods with a specific name in each parent contribution. For each different method, we also verify the type of its parameters allowing us to consider different method signatures for the same method name.

Our approach for identifying *Unavailable Symbol* variables consider the whole source code of a class. For example, if a local variable is present in different methods, and one call for this variable is added in one method, while in another there is a removal or update for such variable. In this case, we consider this case as a conflict caused by contributor changes is identified, while it is caused by integrator changes. This case represents a false positive related to who causes the conflict since the contributions are not conflicting because they occur in different scopes. The same consideration is valid for *Duplicated Declaration* of variables since a local variable can be declared in different methods, and we compute this as a conflict caused by the contributor changes.

To identify conflicting contributions on test conflicts, we instrument *pom.xml* and *travis.yml* files with information of the failed tests for finding the code coverage associated. However, we only have access to this coverage if no problem happens in the build process. In some cases, the build process generates the coverage, but before deploying it, the build process ends due to unexpected errors. Aiming to simplify such problems, we instrument *travis.yml* file

only with necessary and common instructions. In these cases, if test conflicts happen, we are loosing them since our scripts must have the coverage result to perform our analysis.

During checking for interference on test conflicts, we evaluate changes performed by parent's contributions and coverage associated with the failed test case. Although the coverage represents the real exercised methods by the test case, we also consider general changes on file structure. Despite this kind of change, we do not explore their nature, like modification on attributes, update on class structure (interfaces, extensions, inheritance). Thus, if parent's contributions perform different changes on files structure, we compute this as changes on the same file. Such conclusion is not wrong, but a refinement about the changes structure could imply in a precise evaluation informing whether the changes were on the same or dependent element. In future work, we could improve our script for getting such refinement.

The threat associated to a wrong conclusion of who caused a build conflict due to the use of method names instead of its signature also represents a threat here. In this case, when we get the set of methods changed by parent's contributions, we only look for the method names; the same is done for executed methods during the test case execution. If two different methods have the same name, but only one of them is changed by the parent contributions, and the other is executed by the test case, we do not differ each one considering both methods as just one and consider them as changes on same method. In this case, we consider a conflicting contribution when it is not. However, investigating the fix commit for the conflicts, we confirm the same method is changed in the parent contribution and during the fix.

To identify the resolution patterns, we try to find the commits responsible for the fix. The assumption is the fix commit must have as its parents, the commit associated with the broken build, but in some cases, the commit fix has two parents. Since we do not check whether all parents contributions are preserved in such situations, our analysis is partial possibly excluding part of the resolution pattern. Consequently, the adopted resolution pattern does not reflect the truly fix. However, our analysis for resolution patterns in build conflicts is done based on the cause of the conflict, that is independent whether all parents contributions are preserved since they verify only files involved in the failure. In case our analysis for build conflicts cannot be performed or for test conflicts, we perform a manual analysis. Although such analysis was performed for one single person, the files (and methods when applied) were known guiding the analysis. Thus, the person verified only the changes in the files involved in the conflict.

Still in this context of resolution patterns, the commit fix identified must have its build available on Travis. In cases the builds are not found, we could build such commits aiming to check whether they are conflicts fixes, and then perform our analysis having more evidence of resolution patterns. However, we decide to use only data available on Travis since many processes failed due to external causes. Considering the number of commits having as parents the broken build commit, many builds could be necessary to be performed, and none could be the fix. In future work, we could determine a constraint time to select only commits under that interval time.

In case a broken commit has more than one fix commit, our analysis considers only the first fix commit identified. In future works, we could evaluate how other fix commits perform the fixes having more diversity of adopted resolution patterns.

4.3.3 External Validity

First, our sample is composed of open-source projects that cannot reflect how software is developed privately. This characteristic can bias our results related to the frequency of conflicts because developers tend to push commits without errors aiming to keep their reputations clean and associated with a good work.

Our sample contains only Java projects hosted on GitHub using Travis CI for CI and Maven as build manager. We analyze Java projects because of its popularity, and because Java is the second most used language with Travis on GitHub [15]. Moreover, the GumTree support for such language is very precise when compared to others. About the adoption of Travis CI, we decided to use it due to its popularity. Travis offers all build information through its API, differently from other CI services/tools. We only consider projects using Maven for build manager because its log report is very informative bringing enough information to identify conflicts automatically.

Our scripts were defined to support the whole experiment but with limitation related to technology. For example, to run the experiment in other languages, it would be necessary update the scripts responsible for identifying the conflicting contribution between parent's contributions. The same is applied for other build managers since each one can have its own message patterns related to problems. In a new experiment involving repositories hosted in GitHub and using Travis, the scripts responsible for the mining repositories can be completely reused. However, if a study requires other hosting services, Version Control Systems (VCS) or CI services, the mining scripts must be updated.

We analyze build logs looking for specific message patterns associated to problems that cause the broken build. After a pattern is identified, we perform additional analysis as explained in Chapter 3, to ensure the problem is caused by a conflict. For a new sample, different patterns can arise different impairing our scripts to treat them. Consequently, it is necessary to adapt our scripts for these new patterns.

5 CONCLUSION

Collaborative software development brings some challenges when developers do their tasks, separately and simultaneously. When different contributions are integrated, conflicts arise impairing productivity. Conflicts may arise in different development phases: during merging, when different contributions are integrated (merge conflicts); after integration, when building the integration result fails (build conflicts); or when testing, unexpected software behavior happens (test conflicts).

In this study, we perform an empirical study to verify the occurrence of build and test conflicts, their causes as also the adopted resolution patterns, and the person responsible for fixing the conflicts. Our goal is to provide a better understanding of this kind of build and test conflicts focusing on the circumstances these conflicts occur. We identify 125 and 39 build and test conflicts, respectively, associated with 117 and 7 merge scenarios, respectively. These conflicts arise from 60991 merge scenarios from 529 Java projects hosted on GitHub that use Travis CI for Continuous Integration (CI) and Maven as build manager.

Analysing the number of merge scenarios associated with conflicts in comparison with the number of merge scenarios presenting broken builds and superior parents status (473 and 864, for errored and failed status respectively), we verify the frequency of build and test conflicts are **24,73%** and **0,81%**, respectively. So, roughly, 1 in 4 build problem is caused by a conflict in errored builds, whereas the others are caused by a single developer mistake, configuration issues, unresolvable dependencies or unavailability of external services. Considering all scenarios (60991), we verify the frequencies of conflicts decreases to 0,19% and 0,01%.

Our results reveal build and test conflicts occur less than reported by previous studies. Brun et al. [8] reveal 33% of all evaluated merge scenarios (scenarios without merge conflicts) represent cases of build and test conflicts. In the same way, Kasi and Sarma [2] bring such evidence separately: build conflicts ranging from 2% to 15%, while test conflicts ranging from 6% to 35%. However, we eliminate threats, which related studies are not aware. For example, besides to check merge commit build status, we check the build process of merge parents. We also did not perform any build process locally as any change on environment configuration could lead the build process to fail. Instead of (re)build all commits locally, we consider only projects that adopt CI (Travis) obtaining the needed information from CI build process logs. For commits not build yet, we also use Travis service to build them.

Concerning about build conflicts, *Unavailable Symbol* is the most recurrent cause (52,8%,

that represents a reference for an element no longer available in the project). This problem can be motivated by unavailable elements from the project contents as also due to external dependencies. It can also involve dependent classes (one class reference an element from another) as also involving only one class (a reference for a missing element locally). This cause could be treated by an automatic repair tool without human intervention. Build Conflicts are not only caused by problem during build and compilation phase of the build process, but also during Automated Static Analysis (ASAT) phase. The conflicts identified in this category are caused by classes that do not following the style conventions defined for the project. Concerning about the way conflicts are fixed, most adopted patterns primarily consist of keeping the parent's contributions instead of undoing them and return to an old project state. For example, the update of old method reference to the new method signatures. Such behavior is present in 65,38% of all fixes for build conflicts.

Although build conflicts caused by contributor changes are more evident (66,4%), we have evidence that build conflicts can also be caused by changes done performed after the integration by **integrator** (33,6%). The interesting about this finding is the motivation of the inconsistencies. Integrators cause more build conflicts in scenarios, where at least one parent contribution is not preserved (59,52%). In these cases, merge conflicts might happens and the way integrators adopt to fix the merge conflicts are responsible for adding inconsistencies in the source code. The remaining 40,48% conflicts are from scenarios, where all parents contributions are preserved in the final merge commit. It means, even if merge conflicts happens, the way adopted by integrators to fix them preserved all parent's commits in the merge commit. However, integrators also introduce new contributions during the integration adding inconsistencies in the source code. Consequently, the build process fails. Better merge tools could eliminate the need for integrators lead with merge conflicts since such activity is error-prone.

For test conflicts, there are two types of conflicts identified. The first one is caused by failed test cases. Failed test case are not restricted to old test cases in the project, but also new or updated tests during the merge scenario. Changes on methods but also on class structure represent forms of how an dependent changes can lead a test case to fail. The second type of test conflict is resulted of analysis performed after the testing phase of the build process. The case we identified is motivated by unachieved coverage metric after the merge integration. Our scripts could support developers during the investigation of the causes of the failure. Thus, an assistive tool will be responsible for informing only the changes that are done by parents and are involved in the test failure. This information could also be used by automatic repair tools [52] [53]. Fixes for test conflicts commonly change the methods associated with the test case failure execution bringing evidence these methods are genuinely involved with the failure.

Our findings reinforce the evidence about build and test conflicts not only from preserved merge scenarios but also from unpreserved ones. According to the causes and motivations (contributor or integrator changes), our results and scripts can be used to improve assistive tools aiming to avoid or even treat conflicts. Insights of new tools are proposed to treat conflicts

without human effort. We now present in details the contributions of this thesis, related and future work.

5.1 Contributions

This study makes these contributions:

- Quantification, categorization, analysis and identification of resolution patterns of build and test conflicts. We believe the related frequencies can be higher on private projects, especially because conflicts can happen, and developers may treat them before sending to the central repository;
- A catalog of build and test conflicts causes. This catalog is an interesting contribution since previous work focus only in frequency of clean merge scenarios;
- Improvement applied in GumTree tool for presenting a better and more informative syntactic diff;
- Insights and improvement for assistive tools, which could be applied to avoid or treat conflicts;
- Infrastructure to support developers informing the causes and the origin of the conflicts, especially for test ones;
- Availability of all scripts used to perform this experiment encouraging new replications [20].

5.2 Related Work

5.2.1 Empirical Studies

Empirical studies about build and test conflicts provide evidence of their occurrence in practice. In this way, Kasi et al. [2] and Brunn et al. [8] perform studies verifying conflicts frequency. For doing it, they select open-source projects, 4 and 9, later reduced to 3, respectively, hosted on GitHub. To identify conflicts, they try to build locally the *clean merge scenarios* (merge scenarios without merge conflicts). Finally, if the build process fails, such merge scenario represents a conflict (build or test ones). Since they identify conflicts only based on the final 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. Consequently, the problem would be present in the resulting merge commit. Environment configuration also represents cases of false positives; any variation in the environment configuration or an unavailable dependency could contribute to the build process to break leading the authors to a wrong conclusion.

We also use open-source projects, but our sample is 132 and 176, respectively, times larger than adopted by the previous studies. In our study, we use information of merge scenarios already built on Travis. When some merge or parent commits are not built, we do not reproduce build process for it setting manually the project environment. First, we fork the project and use Travis service to build such cases using the specifications defined on the `.travis.yml` file of each project.

Another limitation is the sample adopted by the related work. They only consider *clean merges scenarios* as valid subjects, while we also consider scenarios resulting from merge conflicts (unclean scenarios). According to our results, the occurrence of build conflicts in this new perspective is also expressive. Although merge conflicts occurrence can indirectly influence for parents contributions not be preserved on the integration result, there are cases the preserved contributions are not conflicting, and the integrator changes cause the build conflicts (13,6%).

In the same way, Muylaert and De Roover [57] present a similar study using Travis and Git information. Differently from the previous studies, just like we do, they identify potential test conflicts verifying the merge and parent commits build status. They present 2,34% of merge scenarios present failed status while its parents successful ones. However, they do not validate the causes, which can introduce false positives on their results. For instance, a wrong environment configuration, a remote problem or time restrictions on Travis could lead the build to break, which does not represent a conflict.

Relying on the study subjects, they consider not only merge scenarios but also Pull Requests (PR) (analyzed in different perspective). We decide to not include PRs in our experiment since we would like to investigate the real impact of conflicts during the development. PRs, which are not accepted on the related projects, did not impact the development, and consequently, did not lead someone to spend time on fixing such problems. Therefore, PRs can introduce bias on results. For example, if one PR is accepted on GitHub project, independently of its status, two different builds with the same source code will be started on Travis, and possibly, present the same final status. It is clear the chances of passed PR to be accepted are more prominent than those presenting broken status. Consequently, the rate of build and test conflicts would decrease.

Although in the previous studies the authors verify the conflicts frequency, they do not attempt in investigating their causes. We go further and identify how a contribution affects others. Such information, as reported in Chapter 4, can be used on assistive tools aiming to avoid or treat conflicts occurrence.

Muylaert and De Roover [57] also attempt at verifying the effort spent on potential test conflicts. For that, they look for the first passed build after a broken one. We also adopt this approach for build and test conflicts. In case of build conflict, we consider a fix the first build to present failed or passed status. Investigating whether the fixed build status of a build conflict is passed or not represents another interesting area to be investigated. As part of evaluating effort, the authors also try to inform the files developers change to fix conflicts. However, their approach is imprecise since the causes of the conflicts are unknown. For instance, if the build fails due

to changes in a specific method, and there are changes in files not involved in the failure, their metric will consider all changes as part of the fix. Our approach is precise on informing only the changes patterns adopted for fixing the causes since we already knew them.

Beller et al. [39] perform a study focusing exclusively on the relation of tests and Travis. Their third research question specifically investigates the impact of tests execution in build in Travis. Although they do not consider only merge scenarios of Java projects but all general commits (non-merge commits), we can compare their results with ours. Since builds can present failed tests and still present a passed status (ignoring some partial results), they classify test failures based on this argument. In our study, we first seek for failed builds, and then we investigate the causes. If a build presents failed tests but a passed status, we do not investigate such case since such guideline is defined for the project, which it is aware of such possible failed tests. Based only on builds with failed final status, they conclude such problem represents 3,9% off all analyzed scenarios while our findings present 0,01% of all scenarios. Despite the discrepancy between the frequencies, they involve other aspects that we do not consider (failed test cases due to environment or external issues). For a better comparison, it would be important the related study knew the causes of tests failures filtering false positives as we already did.

5.2.2 Build Errors Diagnosis

Although build errors manifest during the build process, not so much is known about the causes that motivate them. Seo et al. [17] perform a study identifying cause categories and their related frequencies. Although they analyze millions of builds from different projects, such data could bias the results since they consider only projects of a single company. Aiming to achieve generality, we selected a sample of 529 different open-source projects. For each build, they analyze the build log trying to match keywords, which are associated with different error categories. However, their results gather build failures of non- and merge commits; they do not separate the findings based on such perspectives. Moreover, the proportions of non- and merge commits are probably different (more non-merge commits).

Another difference is how Seo's study treats the *Unavailable Symbol* cause. They consider every case of a reference to an unavailable element as a single cause. For example, an unavailable method on a class and an unresolvable dependency. Besides to categorize unavailable elements in different perspectives (dependency and project causes), we do further analysis for this last one. We identify the types of elements that are associated with unavailable symbol caused due to project contents (unavailable class files, methods, and variables).

Kerzazi et al. [58] also present a study aiming to understand the impact of broken builds. For that, they use quantitative and qualitative methods. Their second research question is very related to our second one, as well as some parts of the third. Just like our findings, the primary cause of broken builds is due to missing files (unavailable symbol). Such results also comply with the results of Seo's study.

5.3 Future Work

In this study, we investigate the main characteristics related to build and test conflicts in open-source projects. The first point of extension concerns on replications of this study considering different subjects characteristics. For example, private repositories (restricted to not only projects from companies but also from GitHub) might bring interesting results allowing us to compare with our findings. We could then evaluate whether the projects origin has influence on the occurrence and type of conflicts. Once our approach takes some manual analysis, a complete automated study could eliminate these steps making the experiment easier to replications. Therefore, we identify the following improvements:

- Investigation of the occurrence of build and test conflicts considering diversity related to subjects like other languages (Ruby, C#), CI services/tools (Jenkins) and build managers (Gradle, Ant);
- Although we try to accomplish all message patterns variations (messages used for getting the error causes and additional information), when we introduce new projects in our sample, new message patterns arise. A better and complete support of log information extraction could contribute to identify new conflicts;
- Better instrumentation of *travis.yml* file for the process of execution of failed test cases. Such improvement might avoid the occurrence of errored builds (build process breaks before starting to run the test case) as we already noted;
- For test conflicts, to identify conflicting contributions, our approach considers only the methods and general changes on files from the parent's contributions. Further analysis could be done considering the type of changes on class attributes as also addition or removal of elements that change the class structure, like interfaces, extensions;
- Approach for detecting conflicts not restricted to Java files, but also configuration ones, like *pom.xml* and *travis.yml*;

Other studies could investigate other characteristics that can be intrinsic to the occurrence of build and test conflicts. For example, Cataldo et al. [59] investigate social factors responsible for motivating failures on contributions integration. A possible factor in build and test conflict occurrence could be related to the experience level of a developer. Thus, we could evaluate whether conflicts are more likely to happen with new developers than compared with those experienced on the project [60].

Lessenich et al. [61] investigate technical indicators for predicting the number of merge conflicts in merge scenarios. Some of these indicators are the number of commits involved, substantial contributions and the number of same files changed in both parent commits. Investigating

such technical factors could bring insights for improvements in assistive tools. For example, if number of files were an indicator, assistive tools could alert the developers still during the tasks development.

In the same way, a survey and interviews can be done with development teams aiming to verify their view of the frequency of conflicts over the life cycle of a project. In the end, we could compare our results with this new information. Diversifying the subjects considering developers with different origins (for instance, developers from open-source and private projects). Such diversity could also bring information about how the environment contributes to the conflict occurrence.

Another interesting study could focus on executing the study with better merge tools, like S3M [9]. Therefore, it will be possible to verify whether improved merge tools could eliminate conflicts, especially for those conflicts caused by integrator changes resulted from merge conflicts (related to build conflicts).

Other studies can concentrate on applying the insights for assistive tools (improvements) described in Section 4.2.2.

An interesting finding happens when the commit fix of a build conflict results in a failed build. To investigate how build conflicts fixes can influence on test conflicts occurrence is an innovative area still untouched. Additionally, it will be possible to verify whether there are indicators associating changes for build and test conflicts.

REFERENCES

- [1] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, “Analyzing the state of static analysis: A large-scale evaluation in open source software,” in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 470–481.
- [2] B. K. Kasi and A. Sarma, “Cassandra: Proactive conflict minimization through optimized task scheduling,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 732–741.
- [3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Early detection of collaboration conflicts and risks,” *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, 2013.
- [4] T. Zimmermann, “Mining workspace updates in cvs,” in *Fourth International Workshop on Mining Software Repositories (MSR’07: ICSE Workshops 2007)*. IEEE, 2007, pp. 11–11.
- [5] P. Accioly, P. Borba, and G. Cavalcanti, “Understanding semi-structured merge conflict characteristics in open-source java projects,” *Empirical Software Engineering*, pp. 1–35, 2017.
- [6] G. Cavalcanti, P. Accioly, and P. Borba, “Assessing semistructured merge in version control systems: A replicated experiment,” in *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*. IEEE, 2015, pp. 1–10.
- [7] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, “Semistructured merge: rethinking merge in revision control systems,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 190–200.
- [8] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 168–178.
- [9] G. Cavalcanti, P. Borba, and P. Accioly, “Evaluating and improving semistructured merge,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 59, 2017.
- [10] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 2000.
- [11] P. M. Duvall, *Continuous integration*. Pearson Education India, 2007.
- [12] A. Miller, “A hundred days of continuous integration,” in *Agile, 2008. AGILE’08. Conference*. IEEE, 2008, pp. 289–293.
- [13] G. Dyke, “Which aspects of novice programmers’ usage of an ide predict learning outcomes,” in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 505–510.

- [14] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, “Quality and productivity outcomes relating to continuous integration in github,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 805–816.
- [15] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs, and benefits of continuous integration in open-source projects,” in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 426–437.
- [16] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand, “Continuous integration in a social-coding world: Empirical evidence from github,” in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 401–405.
- [17] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, “Programmers’ build errors: a case study (at google),” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 724–734.
- [18] C. Bird and T. Zimmermann, “Assessing the value of branches with what-if analysis,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 45.
- [19] G. Menezes, “On the nature of software merge conflicts,” Ph.D. dissertation, Federal Fluminense University, 2016, accessed: 2017-06-16.
- [20] Appendix, 2018, uRL: <https://github.com/leusonmario/TravisAnalysis//>.
- [21] R. Conradi and B. Westfechtel, “Version models for software configuration management,” *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 232–282, 1998.
- [22] Subversion, 2018, uRL: <https://subversion.apache.org/>.
- [23] CVS, 2018, uRL: <http://savannah.nongnu.org/projects/cvs/>.
- [24] GIT, 2018, uRL: <https://git-scm.com/>.
- [25] MERCURIAL, 2018, uRL: <https://www.mercurial-scm.org/>.
- [26] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, “How do centralized and distributed version control systems impact software changes?” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 322–333.
- [27] G. Gousios, M. Pinzger, and A. v. Deursen, “An exploratory study of the pull-based software development model,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 345–355.
- [28] A. Sarma, D. F. Redmiles, and A. Van Der Hoek, “Palantir: Early detection of development conflicts arising from parallel code changes,” *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 889–908, 2012.
- [29] S. Apel, O. Leßenich, and C. Lengauer, “Structured merge with auto-tuning: balancing precision and performance,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 120–129.
- [30] S. Chacon and B. Straub, *Pro git*. Apress, 2014.

- [31] Travis CI, 2017, uRL: <https://travis-ci.org/>.
- [32] Jenkins, 2017, uRL: <https://jenkins-ci.org/>.
- [33] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-wesley Reading, 2007, vol. 2.
- [34] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.
- [35] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 6–pp.
- [36] FindBugs, 2018, uRL : <http://findbugs.sourceforge.net/>.
- [37] CheckStyle, 2018, uRL : <http://checkstyle.sourceforge.net/>.
- [38] IntelliJIDEA, <https://www.jetbrains.com/idea/>, 2018.
- [39] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 356–367.
- [40] EclEmma, 2018, uRL : <http://www.eclemma.org/jacoco/>.
- [41] MAVEN, 2018, uRL: <https://maven.apache.org>.
- [42] Gradle, 2018, uRL: <https://gradle.org/>.
- [43] ANT, 2018, uRL: <http://ant.apache.org/>.
- [44] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.
- [45] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.
- [46] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.
- [47] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, 2013, pp. 466–476.
- [48] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*. ACM, 1992, pp. 107–114.
- [49] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, "Fastdash: a visual dashboard for fostering awareness in software teams," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 1313–1322.

- [50] L. Hattori and M. Lanza, “Syde: A tool for collaborative software development,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 235–238.
- [51] C. Le Goues, S. Forrest, and W. Weimer, “Current challenges in automatic software repair,” *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, 2013.
- [52] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 772–781.
- [53] S. H. Tan and A. Roychoudhury, “relifix: Automated repair of software regressions,” in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 471–482.
- [54] KDIFF3, 2018, uRL: <http://kdifff3.sourceforge.net/>.
- [55] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, “The impact of continuous integration on other software development practices: a large-scale empirical study,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 60–71.
- [56] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, “The promises and perils of mining git,” in *Mining Software Repositories, 2009. MSR’09. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 1–10.
- [57] W. Muylaert and C. De Roover, “Prevalence of botched code integrations,” in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 503–506.
- [58] N. Kerzazi, F. Khomh, and B. Adams, “Why do automated builds break? an empirical study,” in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 41–50.
- [59] M. Cataldo and J. D. Herbsleb, “Coordination breakdowns and their impact on development productivity and software failures,” *IEEE Transactions on Software Engineering*, vol. 39, no. 3, pp. 343–360, 2013.
- [60] M. Rebouças, R. O. Santos, G. Pinto, and F. Castor, “How does contributors’ involvement influence the build status of an open-source software project?” in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 475–478.
- [61] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, “Indicators for merge conflicts in the wild: survey and empirical study,” *Automated Software Engineering*, pp. 1–35, 2017.

APPENDIX

APPENDIX A - Build Conflicts Identification

In this Appendix, we continue to present the approach to identify build conflicts (see Section 3.4.1 in Chapter 3).

A.1 Syntax Conflicts

The problems related to syntax are restricted to malformed programs. Since the parent commits present successful compilation process, if the merge result has any syntax problem, we can assume it is introduced by the integrator changes. Thus, our approach to such problem is straightforward.

A.1.1 Malformed Program

This problem happens when the source code is syntactically incorrect. For example, in XChange project, the **build** process breaks because of a missing semi-coma (;) at the end of a line. Additionally, there is also the presence of a merge conflict header in the source code («««») (Figure A.1). To determine the motivator of the build conflict, we verify the occurrence of merge conflicts during the merge scenario. Thus, if none merge conflict happens, we assume such problem is caused by the **contributor changes**; otherwise, it is a problem motivated by the **integrators**.

A.2 Static Semantics Conflicts

This category of problems is related to source code syntactically correct but semantically wrong. Section 2.2.1.1 in Chapter 2 presents how these problems arise during the compilation process.

Figure A.1: Build log of broken build due to malformed program

```
Compilation failure
[ERROR] /home/[...]/xchange/[...]/OkCoinUtils.java:[43,1] illegal start of
expression
[ERROR] /home/[...]/xchange/[...]/OkCoinUtils.java:[44,9] ';' expected
```

Figure A.2: Build log of broken build due to unimplemented method

```

Compilation failure
[ERROR] /home/[...]/cloudslang/[...]/PreCompileValidatorImpl.java:[46,8]
is not abstract
and does not override abstract method validateResultName(java.lang.String)
in io.cloudslang.lang.compiler.validator.PreCompileValidator

```

A.2.1 Unimplemented Method (Super Type)

Unimplemented method cause happens when an implementing class does not present an implementation of a method defined in the related interface (or in some cases extends for inheritance). For instance, in Cloud Slang project, the `build` process breaks due to the missing method implementation (Figure A.2). In this case, the method `validateResultName` (unimplemented method) is introduced in the interface `PreCompileValidator` (secondary class). However, the implementing class `PreCompileValidatorImpl` (main class) does not present the associated new implementation.

A.2.1.1 Build Conflict Motivation

To verify if the conflict is caused by the parent's contributions or integrator changes, we adopt the following approach done in five steps:

- **Step 1:** Identification of the parent responsible for updating or introducing the *unimplemented method* in the *Secondary class* **base-parent-diff**.
- **Step 2:** In the other parent, we perform two checks. The first verifies whether the *unimplemented method* is added or updated in the *Secondary class* **parent-merge-diff** (ensuring the change in the *Secondary class* is preserved in the merge result).
- **Step 3:** The second check is performed on the *Main class* **parent-merge-diff** to ensure the *unimplemented method* is not added/updated in the implementing class.
- **Step 4:** If both checks can be done, we conclude a build conflict happens due to **contributor changes**.
- **Step 5:** If one of the previous checks cannot be done, we assume such build conflict results of **integrator changes**.

Unimplemented Method of Super Type works in the same way, although such super type is resulting from a dependency. To define the motivator of the conflict, we perform a manual analysis.

Figure A.3: Build log of broken build due to incompatible method signature

```

Compilation failure
[ERROR] /home/[...]/ontop/[...]/Quest.java:[569,58] no suitable constructor
found for
for SQLGenerator constructor is not applicable
(actual and formal argument lists differ in length)

```

A.2.2 Incompatible Method Signature

Different of *Unavailable Symbol Method*, in this case the required method is available but the list of parameters or the types expected are different of the received. For instance, in Ontop project, the `build` process breaks because the constructor (*incompatible method*) of *Quest* (*secondary class*) could not be used in the request of *SQLGenerator* (*main class*) since the parameters list differ in length (Figure A.3).

A.2.2.1 Build Conflict Motivation

To verify if the conflict is caused by the parent's contributions or integrator changes, we adopt the following approach done in three steps:

- **Step 1:** Identification of the parent responsible for updating/removing the *incompatible method* in the *Secondary class*. Since many methods can have the same name, differing of each other in the signature, we check the number of methods with the name of the *incompatible method* available in the *Secondary class* of each parent.
- **Step 2:** If each parent presents a different number of methods or parameter list size, we assume the *incompatible method* was available before the merge integration. Thus, one of the parents removed the method leading the build to break due to **contributor changes**.
- **Step 3:** If one of the previous checks fails, we assume such build conflict has resulted of **integrator changes**.

A.2.3 Duplicated Declaration

Different of *Incompatible Methods Signature*, in this case, two elements with the same identifier are declared in a class (variables, methods). Thus, when a request is done for such element, the program cannot decide which one to use. Blueprint project has a `build` process that breaks because of two declarations of method *testRemoveNonExistentVertexCausesException* (*duplicated method*) in the *GraphTestSuite* (*main class*) (Figure A.4).

Figure A.4: Build log of broken build due to duplicated declaration

```

Compilation failure
[ERROR] /home/[...]/blueprint/[...]/GraphTestSuite.java:[393,16] error:
method
testRemoveNonExistentVertexCausesException() is already defined in class
GraphTestSuite

```

Figure A.5: Build log of broken build due to incompatible types

```

Compilation failure:
[ERROR] /home/[...]/neo4j-timetree/[...]/SingleTimeTreeTest.java:[641,32]
error: incompatible types:
long cannot be converted to TimeInstant

```

A.2.3.1 Build Conflict Motivation

To verify if the conflict is caused by the parents contributions or integrator changes, we adopt the following approach done in three steps:

- **Step 1:** Checking of the introduction of the *duplicated method* in *Main class* **base-parent-diff** of both parents.
- **Step 2:** If such check is done, we conclude the build conflict is caused by the **contributor changes**.
- **Step 3:** If the previous check fails, we assume such build conflict is resulted due to **integrator changes**.

A.2.4 Incompatible Types

This problem happens when an unexpected type is received or sent at an operation. For instance, in Neo4J TimeTree project, the **build** process fails because of a parameter of type *long* is sent instead of *TimeInstant* in *getInstant* method of *TimeTree* class (Figure A.5). To define the motivator of the conflict, we seek for merge conflicts and changes done after the merge integration. Thus, if none of such actions happen, we assume the **contributor changes** motivates the conflict. Otherwise, we perform a manual analysis to determine the motivator.

A.3 Other Static Analysis Conflicts

Static analysis are done after the compilation process (see Section 2.2.1.2 in Chapter 2).

Figure A.6: Build log of broken build due to unfollowed project guideline

```
BUILD FAILURE
[ERROR] Failed to execute goal com.mycila:license-maven-
plugin:2.8:check (check-license)
Some files do not have the expected license header
```

A.3.1 Project Rules

This problem happens when the source code is syntactically and semantically correct, but it does not follow the conventions/guidelines defined to the project. For instance, in Java Driver project, a file does not have the *expected license header* resulting in a broken [build](#) process (Figure A.6). Like the approach adopted for *incompatible types*, if none merge conflicts or changes after the merge integration happens, we consider such case is motivated by the contributor changes. Otherwise, we perform a manual analysis to determine the motivator.

APPENDIX B - Build Conflict Fixes

In this Appendix, we continue to present the approach to identify the fixes patterns adopted on build conflicts (see Section 3.5.1 in Chapter 3). All resolution patterns are identified using the diff between the broken merge and fix commits **result-fix-diff**.

B.1 Syntax Conflicts

B.1.1 Malformed Program

Basically, fixes for malformed program problems involve adaptations of the source code to the associated syntax language depending on the problem. For example, source code with *merge conflict headers* are fixed eliminating them.

B.2 Static Semantics Conflicts

For almost all problems related to semantic conflicts are treated automatically for our scripts. In case, the automatize analysis fails, we also perform a manual analysis.

B.2.1 Unavailable Symbol

Besides the verifications presented in Section 3.5.1 in Chapter 3, here we present additional verifications exclusively for unavailable symbol *files*.

In case the missing file is caused by unavailable import, we perform two checks aiming trying to match the following patterns into the diffs:

- Import update using the pattern in the *Main* class diff: `Update QualifiedName : {missing_file_identifier}`
- Import removal using the pattern in the *Main* class diff: `Delete (QualifiedName | SimpleName | SimpleType) : {missing_file_identifier}`

B.2.2 Unimplemented Method (Super Type)

In this case, we verify whether the implementing class of an interface introduces a new implementation for the missing method. We do this looking for the following pattern into the

GumTree diff of the *Main* class:

```
Insert SimpleName: {missing_method_identifier} into  
MethodDeclaration.
```

B.2.3 Incompatible Method Signature

In this case, we look for the request removal for the missing method in the *Main* class diff trying to match the following patterns into the diff:

```
Delete SimpleName: {missing_method_identifier}.
```

B.2.4 Duplicated Declaration

In this case, we verify whether the class with the duplicated declaration removes one of the duplications. We do this trying to match the following pattern in the GumTree diff of the *Main* class:

```
Delete SimpleName: {duplicated_declaration_identifier}.
```

B.2.5 Incompatible Method Signature

Like the identification of conflicts of this category, the associated fixes are also done manually. Thus, we look for types changes in the request or in the operation that receives it. Additionally, we also consider the request removal of such operation.

B.3 Other Static Analysis Conflicts

Like syntax fixes, in this category, fixes can be checked looking the guidelines defined for the project. Thus, in our manual analysis we verify whether the set of files not following the project guidelines change their structure to adapt them for the styles conventions defined previously.

APPENDIX C - Travis Configuration File Instrumentation

This Appendix presents our approach to adapting *travis.yml* file to perform a build process in Travis generating the associated code coverage for a test case. Figure C.1 presents an example of an adapted *travis.yml* file.

- **language/jdk:** We extract such information from the *travis.yml* file keeping the same instruction.
- **script:** This instruction is completely changed. The *DTest* arguments are taken from the Travis log (Figure 3.10). The coverage command is specified in its documentation. Since the test case can fail, and such behavior is expected, we want the build process continues and does not stop. So, the argument *DfailIfNoTests* is set up to false.
- **before_deploy:** We zip the coverage result informing the path to such file. The path is default and defined by the own coverage tool.
- **deploy:** This instruction is composed of many arguments. By default, the argument *provider* must be set to *releases*. Many options can be used to deploy, but since we choose GitHub tags, the argument *tags* must be *true*. To ensure Travis will overwrite any information and not delete files created during the build, the arguments *overwrite* and *skip_cleanup* are set to *true*. The argument *file* indicates the file to be deployed (in our case, *coverage-result.tar.gz*). [Personal Access Token](#) does the authentication in GitHub. For each project, we locally encrypt the GitHub token using the indication of Travis [documentation](#).
- **sudo:** Since we want to deploy the code result in GitHub, it is necessary to give this kind of permission.

Figure C.1: Travis.yml file example for running failed test case

```
1. sudo: required
2.
3. language: java
4.
5. jdk:
6.   -oraclejdk8
7.
8. script:
9.   - mvn clean cobertura:cobertura -Dtest={TestFileName}#{TestCaseName}
   -DfailIfNoTests=false
10.
11. before_deploy:
12.   - tar -zcvf coverage-result.tar.gz /pathToDirectory/cobertura
13.
14. deploy:
15.   provider: releases
16.   api_key:
17.     secure: "encryptedKey"
18.   file: coverage-result.tar.gz
19.   file_glob: true
20.   overwrite: true
21.   skip_cleanup: true
22.   on:
23.     tags: true
```

APPENDIX D - Study Sample

This appendix presents the sample we used to perform our study. It is composed of 529 projects involving 60991 merge scenarios. Table [D.1](#) presents the general information of each project as also some demographic ones.

All scripts used to perform our experiment can be found in our [GitHub repository](#).

Table D.1: Sample

Project	MergeScenarios (MS)	Not Built MS	Errored MS	Failed MS
GoClipse/goclipse	426	356	8	49
taweili/ardublock	0	0	0	0
twitter/cloudhopper-smpp	23	3	7	1
RichardWarburton/lambda-behave	17	0	2	2
isaiah/jubilee	12	2	4	10
roundrop/facebook4j	63	31	3	0
brooklyncentral/clocker	216	29	29	22
reficio/soap-ws	2	0	0	0
pulse00/Twig-Eclipse-Plugin	8	4	1	4
hcoles/pitest	96	23	5	3
rhuss/docker-maven-plugin	0	0	0	0
kite-sdk/kite	80	3	42	17
javaee-samples/javaee7-samples	106	2	40	132
mpetazzoni/torrent	20	0	0	0
analytically/innerbuilder	9	4	4	0
linkedln/pinot	20	11	9	2
Athou/commafeed	56	1	9	0
cryptomator/cryptomator	120	39	6	14
spring-projects/spring-petclinic	25	6	0	1
sqshq/PiggyMetrics	10	4	2	1
apache/hive	65	34	0	29
VerbalExpressions/JavaVerbalExpressions	27	13	0	3
alibaba/java-dns-cache-manipulator	0	0	0	0
CloudSlang/score	154	2	3	8
CloudSlang/cloud-slang	818	247	51	60
native4j/java/BridJ	7	0	2	3
effektif/effektif	91	30	5	12
mrwilson/java-dirty	1	0	0	0
locationtech/udig-platform	34	1	25	9
apache/incubator-samoa	3	0	1	1
pac4j/spring-security-pac4j	34	7	4	0
CorfuDB/CorfuDB	594	87	48	30
martinpaljak/GlobalPlatformPro	5	0	0	0
jparsec/jparsec	36	6	9	0
spring-cloud/spring-cloud-consul	69	18	13	13
tbroyer/bullet	0	0	0	0
lookfirst/sardine	49	3	0	22
penberg/fjord	1	0	0	0
Esri/spatial-framework-for-hadoop	32	1	2	0
Esri/geometry-api-java	35	1	2	0
eXist-db/exist	1011	104	215	155
kongchen/swagger-maven-plugin	163	27	40	32
jamesdbloom/mockserver	64	48	3	12
trecloux/yeoman-maven-plugin	4	0	0	0
jsonld-java/jsonld-java	68	7	18	8
spotify/netty-zmtp	13	1	1	0
maxmind/GeoIP2-java	36	0	1	0
bitsofproof/supernode	28	10	6	2
f2prateek/progressbar	11	0	0	2
gresrun/jesque	24	0	1	4
damnhandy/Handy-URI-Templates	28	3	4	5
apache/accumulo	477	157	77	49
airlift/airlift	1	0	0	0
adamfisk/LittleProxy	139	11	16	22
caelum/caelum-stella	41	2	19	4
FasterXML/aalto-xml	2	0	0	0
mfornos/humanize	8	4	0	1
cosmin/IClosure	0	0	0	0
jhalterman/typetools	12	0	0	0
szegedi/dynalink	2	0	0	1
jooby-project/jooby	129	16	38	48
hansenji/pocketknife	3	0	0	0
immutable/immutable	133	13	0	31
HubSpot/jinjava	135	53	13	2
graphaware/neo4j-reco	14	2	3	1
SpoutDev/Vanilla	21	3	11	4
goodow/realtime-store	0	0	0	0
cloudfoundry/java-buildpack-auto-reconfiguration	50	11	2	8
kzwang/elasticsearch-image	2	0	0	2
jOOQ/jOOQ	53	26	32	2
4pr0n/ripme	42	23	11	56
resty-gwt/resty-gwt	93	6	9	14
yusuke/twitter4j	5	0	4	6
searls/jasmine-maven-plugin	51	10	1	2
brianfrankcooper/YCSB	279	11	11	20
mewson/couchdb-lucene	17	0	2	0
jhy/jsoup	59	27	3	3

sculptor/sculptor	37	4	12	20
OpenSmpp/opensmpp	5	0	0	1
mybatis/guice	75	10	2	2
javanna/elasticshell	12	2	1	1
owlake/genson	16	1	3	1
jqno/equalsverifier	253	196	1	2
etsy/statsd-jvm-profiler	16	7	0	0
square/javapoet	251	5	11	5
liaohuqiu/SimpleHashSet	0	0	0	0
google/google-oauth-java-client	8	0	2	0
google/google-http-java-client	3	0	2	0
javaparser/javaparser	789	337	7	2
naver/pinpoint	1453	16	360	120
digitalfondue/lavagna	87	9	14	2
druid-io/druid	1128	4	59	394
GoogleCloudPlatform/DataflowJavaSDK	264	4	20	0
relayrides/pushy	201	39	18	15
aerogear/aerogear-unifiedpush-server	524	9	106	20
language-tool-org/language-tool	232	130	2	17
yegor256/xembly	6	0	5	2
google/auto	182	4	2	35
mesos/hadoop	21	1	2	0
owls/owlapi	70	12	8	2
guokr/simbase	0	0	0	0
palantir/eclipse-typescript	32	2	1	0
nurkiewicz/async-retry	3	2	1	0
easymock/objenesis	7	1	3	0
graphaware/neo4j-framework	97	24	33	36
jeevatkm/digitalocean-api-java	16	1	3	0
evliotc/carbonite	0	0	0	0
sksamuel/elasticsearch-river-neo4j	12	5	4	4
ppat/storm-rabbitmq	20	10	0	0
julman99/gson-fire	9	1	0	0
StripesFramework/stripes	0	0	0	0
bujii/buji-pac4j	27	1	3	0
usc/wechat-mp-sdk	1	0	0	0
nurkiewicz/LazySeq	0	0	0	0
OpenSextant/SolrTextTagger	15	2	6	0
sps/mustache-spring-view	3	0	0	0
hdiv/hdiv	74	15	13	4
UISpec4J/UISpec4J	0	0	0	0
jhalterman/concurrentunit	2	0	0	0
stefanbirkner/system-rules	0	0	0	0
xebia/Xebium	31	6	3	6
buddycloud/buddycloud-server-java	192	52	9	10
togglez/togglez	56	1	12	4
JakeWharton/salvage	1	0	2	0
logstash/log4j-jsonevent-layout	11	1	0	2
mybatis/jpetstore-6	63	0	0	2
mybatis/generator	131	6	5	18
ralfstx/minimal-json	7	0	0	0
brianm/really-executable-jars-maven-plugin	3	0	0	0
jOOQ/jOOQ	6	2	7	0
FasterXML/jackson-dataformat-yaml	30	13	1	2
bkiers/Liqp	41	2	2	0
ning/maven-duplicate-finder-plugin	0	0	0	0
threerings/tripleplay	3	1	1	0
livingsocial/HiveSwarm	5	0	0	0
zeroturnaround/zt-zip	26	0	1	6
CloudifySource/cloudify	225	27	85	86
threerings/playn	13	0	8	0
basho/riak-java-client	386	213	86	57
goldmansachs/gs-collections	0	0	0	0
kevinsawicki/wishlist	1	0	2	0
FasterXML/jackson-databind	670	251	212	65
FasterXML/jackson-core	153	63	19	4
DSpace/DSpace	1226	80	47	134
adylu/jafka	3	1	0	0
ocpsolt/rewrite	28	0	8	10
asual/lesscss-engine	0	0	0	0
tinkerpop/blueprints	142	28	66	44
searchbox-io/Jest	186	17	26	88
tananaev/traccar	402	204	15	1
nodebox/nodebox	76	23	14	4
tinkerpop/gremlin	14	2	11	2
airlift/airline	0	0	0	0
mikera/clisk	9	2	0	0
killme2008/Metamorphosis	13	8	5	0
tinkerpop/rexster	65	12	39	0
twilio/twilio-java	238	82	19	13

Multiverse/Multiverse-Core	9	0	2	0
resthub/resthub-spring-stack	128	19	20	14
essentials/Essentials	133	38	1	73
spullara/mustache.java	58	12	1	5
springside/springside4	32	0	8	12
dynjs/dynjs	53	2	22	6
SpoutDev/Spout	64	29	17	14
salyh/elasticsearch-security-plugin	0	0	0	0
addthis/stream-lib	42	2	4	16
drbb/java-rust-example	0	0	0	0
Spoutcraft/Spoutcraft	14	1	16	2
reficio/p2-maven-plugin	22	10	2	0
structr/structr	1125	776	71	303
hstaudacher/osgi-jax-rs-connector	23	0	0	8
gwt-maven-plugin/gwt-maven-plugin	7	0	0	0
madeye/proxydroid	0	0	0	0
yegor256/s3auth	46	12	3	58
MoriTanosuke/glacieruploader	15	3	1	0
jeluard/semantic-versioning	26	2	2	0
resthub/springmvc-router	29	5	2	3
jOOQ/jOOOR	17	2	4	2
DiUS/java-faker	48	10	16	9
pac4j/pac4j	473	3	23	28
pedrovgs/Algorithms	13	6	0	1
Graphify/graphify	3	1	0	4
wix/petri	17	2	1	5
google/guava	1	0	0	0
sstone/akka-amqp-proxies	1	0	0	0
toomasr/skype-bot	1	0	1	0
koraktor/steam-condenser-java	19	6	1	0
bbeck/token-bucket	4	0	0	0
Findwise/Hydra	114	1	8	27
mikaelhg/urlbuilder	13	1	1	4
organicveggie/metrics-statsd	0	0	0	0
tumblr/jumbl	33	6	3	0
mybatis/mybatis-3	304	7	15	11
undera/jmeter-plugins	183	38	44	37
mybatis/spring	104	7	6	1
roboguice/roboguice	103	14	58	21
openpnp/openpnp	437	159	18	51
CamelCookbook/camel-cookbook-examples	19	14	1	2
zhangkaitao/es	11	0	13	4
wstrange/GoogleAuth	18	6	9	0
jmxtrans/jmxtrans	287	10	3	78
OpenGrok/OpenGrok	395	16	6	4
wuman/JReadability	1	0	0	0
doanduyhai/Achilles	47	10	11	2
Yubico/ykneo-openpgp	8	2	0	4
lvggiano/owner	36	1	7	0
perwendel/spark	226	29	3	16
apache/jackrabbit-oak	0	0	0	0
jbehave/jbehave-core	4	1	1	0
webbit/webbit	9	1	4	4
qos-ch/slf4j	43	4	0	3
magro/kryo-serializers	22	0	2	0
opensagres/xdcreport	42	21	0	2
play/play-android	0	0	0	0
Asquera/elasticsearch-http-basic	12	0	0	0
maxmind/geoip-api-java	17	0	0	3
dain/leveldb	2	0	2	0
before/uuidetector	0	0	0	0
woorea/openstack-java-sdk	76	3	14	0
caskdata/tephra	94	1	5	2
jwt/jjwt	77	8	0	16
square/burst	18	0	2	0
spring-cloud/spring-cloud-config	138	53	41	4
sd4324530/fastweixin	54	47	0	0
zafarkhaja/jsemver	0	0	0	0
davidmoten/rtr	27	2	5	0
GlowstoneMC/Glowstone	104	28	38	26
OfficeDev/ews-java-api	115	5	3	0
NLPchina/elasticsearch-sql	127	33	28	24
spring-projects/spring-security-oauth	10	8	0	1
gephi/gephi	92	35	18	12
yasserg/crawler4j	45	5	0	0
socketio/socket.io-client-java	27	1	0	8
pagehelper/Mybatis-PageHelper	1	1	0	0
aws/aws-sdk-java	163	6	16	108
bytedeco/javacpp	0	0	0	0
OpenFeign/feign	79	1	1	10

bonnyfone/vectalign	0	0	0	0
vavr-io/vavr	1137	68	31	21
otale/tale	47	3	2	0
bytedeco/javacv	0	0	0	0
jhalterman/failsafe	13	1	2	2
chewiebug/GCViewer	19	5	0	9
ninjaframework/ninja	96	9	4	5
hs-web/hsweb-framework	74	20	13	16
arturmkrtychyan/iban4j	8	0	1	0
siom79/japicmp	68	11	17	1
mgodave/barge	47	7	27	12
torakiki/sejda	23	7	7	0
corydissinger/raw4j	15	2	1	6
justinsb/jetcd	3	0	0	3
microg/NetworkLocation	2	0	2	0
aws/aws-dynamodb-session-tomcat	1	0	0	2
jcgay/maven-color	0	0	0	0
brettwooldridge/NuProcess	24	1	5	0
orienttechnologies/spring-data-orientdb	0	0	0	0
davidmoten/rxjava-jdbc	24	0	2	2
redwarp/9-Patch-Resizer	7	0	0	0
la-team/light-admin	15	0	2	1
larsga/Duke	0	0	0	0
dropwizard/metrics	228	15	6	24
IDPF/epubcheck	81	1	2	4
jhalterman/lyra	14	1	0	0
jpjpush/jpush-api-java-client	80	4	16	1
caelum/vraptor4	566	17	14	58
coverity/coverity-security-library	0	0	0	0
metamx/druid	29	23	4	12
adylu/zkclient	3	1	0	1
jmock-developers/jmock-library	19	2	1	0
mikera/vectorz	138	31	9	0
FasterXML/jackson-module-jsonSchema	36	12	2	6
oltpbenchmark/oltpbench	68	10	1	7
JodaOrg/joda-beans	6	1	4	0
reidarsollid/RustyCage	3	1	0	0
ajermakovics/eclipse-instasearch	7	0	0	0
jzy3d/jzy3d-api	20	3	4	9
pierre/meteo	1	0	2	0
jruby/joni	17	8	0	10
ganglia/jmxetrc	14	1	10	0
sanity/LastCalc	6	1	0	0
jhalterman/expiringmap	8	0	0	1
FasterXML/jackson-datatype-joda	29	13	4	3
ThreeTen/threeten-extra	27	0	2	0
Simmetries/simmetries	36	6	18	0
brettwooldridge/SansOrm	10	0	0	0
olap4j/olap4j	4	4	0	0
tbroyer/gwt-maven-plugin	0	0	0	0
wuman/AndroidImageLoader	2	0	4	0
zenobase/geocluster-facet	0	0	0	0
checkstyle/checkstyle	12	0	0	0
google/compile-testing	32	1	1	0
FasterXML/jackson-datatype-hibernate	17	9	1	2
square/wire	504	1	10	70
l0rdn1kk0n/wicket-bootstrap	336	82	262	60
torodb/stampede	407	263	9	19
chanjarster/weixin-java-tools	64	26	0	0
poetix/protonpack	20	0	0	2
sanity/quickml	221	53	80	24
iluwatar/java-design-patterns	318	75	50	18
xerial/sqlite-jdbc	102	0	9	6
OryxProject/oryx	100	1	0	6
socketqwe/fragmentargs	18	9	2	14
google/guice	70	0	4	6
spotify/docker-maven-plugin	100	72	4	2
ObeoNetwork/UML-Designer	3	0	0	0
google/truth	94	15	2	16
databricks/learning-spark	0	0	0	0
spotify/helios	846	750	29	32
spotify/docker-client	450	183	52	116
xtreemfs/xtreemfs	372	93	31	78
TannerPerrien/picasso-transformations	0	0	0	0
knightliao/disconf	55	7	4	0
tuenti/SmsRadar	2	0	0	0
google/closure-compiler	274	2	26	8
querydsl/querydsl	729	27	92	113
cglib/cglib	30	0	0	12
caelum/mamute	122	5	2	28

square/okio	140	0	1	2
igniterealtime/Openfire	626	8	0	36
devnied/EMV-NFC-Paycard-Enrollment	5	2	0	1
oxo42/stateless4j	18	4	1	0
jirutka/spring-rest-exception-handler	0	0	0	0
SonarSource/sonarqube	312	70	4	120
stephanenicolas/robospice	50	6	39	16
psi-probe/psi-probe	437	3	20	20
square/moshi	134	0	7	0
square/keywhiz	270	1	5	4
objectify/objectify	14	1	0	1
vbauer/jackdaw	2	0	0	0
vbauer/caesar	0	0	0	0
mono/sharpen	0	0	0	0
timmolter/XChange	1343	175	96	129
cloudfoundry/cf-java-client	1350	1150	42	102
ktoso/maven-git-commit-id-plugin	120	1	7	2
ryantenney/metrics-spring	43	4	2	2
rtyley/roboguice-sherlock	0	0	0	0
Hidendra/LWC	13	0	0	1
jOOQ/jOOQ	113	10	64	8
jsimone/webapp-runner	16	9	11	4
julianhyde/linq4j	0	0	0	0
square/retrofit	526	3	10	20
geotools/geotools	1242	203	178	510
sematext/HBaseHUT	0	0	0	0
JodaOrg/joda-time	72	1	0	0
tcurdt/jdeb	60	5	9	8
mitreid-connect/OpenID-Connect-Java-Spring-Server	60	28	3	4
jknack/handlebars.java	75	7	23	8
scobal/seyren	135	15	6	10
zxing/zxing	75	50	5	1
weld/core	0	0	0	0
karussell/snacktory	17	9	0	18
kevinsawicki/http-request	3	0	0	0
caelum/vraptor	86	11	15	4
notnoop/java-apns	42	3	7	8
gwtbootstrap/gwt-bootstrap	95	7	24	16
super-csv/super-csv	30	2	0	2
pac4j/play-pac4j	88	6	14	3
asciidocfx/AsciidocFX	156	14	48	2
spotify/cassandra-reaper	139	44	9	4
expectedbehavior/gauges-android	0	0	0	0
apache/drill	58	58	0	1
dkunzler/esperandro	16	4	5	1
trautonen/coveralls-maven-plugin	24	3	8	0
nurkiewicz/spring-data-jdbc-repository	6	2	5	2
dianping/cat	1273	575	133	175
datastax/java-driver	422	92	50	35
paulhoule/infovore	24	12	8	0
Comcast/cmb	4	0	6	0
olivergierke/spring-restbucks	0	0	0	0
klout/brickhouse	9	0	0	0
andorm/andorm	3	2	1	2
Comcast/jrugged	23	0	0	12
yammer/tenacity	52	27	7	2
google/jimfs	9	1	3	0
protostuff/protostuff	105	5	6	0
twitter/hpack	4	1	0	0
WhisperSystems/BitHub	9	4	0	0
gwtbootstrap3/gwtbootstrap3	156	5	3	25
plantuml/plantuml	1	0	0	0
ronmamo/reflections	28	1	0	3
guari/eclipse-ui-theme	10	0	0	0
code4craft/webmagic	106	18	31	10
TheHolyWaffle/TeamSpeak-3-Java-API	7	0	0	0
awslabs/route53-infima	0	0	0	0
jcabi/jcabi-github	225	43	56	107
jreijn/spring-comparing-template-engines	11	1	0	2
uaihebert/uiacontacts	0	0	0	0
ripple/ripple-lib-java	10	1	5	0
Spoutcraft/LegacyLauncher	6	2	0	0
square/seismic	6	0	0	2
ArcBees/GWTP	69	11	60	0
jacoco/jacoco	86	8	22	6
FasterXML/jackson-dataformat-csv	45	21	1	7
square/pollexor	21	0	0	0
restlet/restlet-framework-java	157	38	4	103
square/okhttp	1591	248	260	219
neuland/jade4j	41	20	6	8

pulse00/Symfony-2-Eclipse-Plugin	19	2	11	1
neophob/PixelController	39	13	4	0
smooks/smooks	12	0	1	10
julianhyde/optiq	8	2	2	0
fabiomaffioletti/jsondoc	22	0	7	1
cereda/arara	15	7	0	0
pgjdbc/pgjdbc	179	6	6	21
FasterXML/jackson-dataformat-xml	40	21	8	2
lemire/javaawah	36	19	0	2
tinkerpop/frames	34	7	12	0
samskivert/jmustache	5	0	0	0
mpatric/mp3agic	26	2	2	2
timmolter/XChart	76	2	10	18
redline-smalltalk/redline-smalltalk	72	5	6	35
graphaware/neo4j-timetree	34	8	11	7
thelinmichael/spotify-web-api-java	30	12	10	2
shilad/wikibrain	258	104	124	14
codecentric/spring-boot-starter-batch-web	19	5	2	5
clemp6r/futuroid	1	0	0	0
apache/tajo	157	80	23	35
FenixEdu/fenixedu-academic	1005	181	52	16
welovecoding/editorconfig-netbeans	32	18	0	0
greenmail-mail-test/greenmail	53	38	0	0
OpenSOC/opensoc-streaming	25	1	3	0
caskdata/coopr	836	395	234	117
paukiatwee/budgetapp	10	6	2	0
jcabi/jcabi-ssh	6	0	0	40
OrbitzWorldwide/consul-client	8	0	1	0
jmozanec/cron-utils	194	31	19	80
vdenotar/spring-boot-security-saml-sample	0	0	0	0
mesos/storm	112	3	0	0
Kixeye/chassis	6	0	4	0
matlux/jvm-breakglass	5	2	0	0
Tillerino/Tillerinobot	18	1	0	0
matlux/jvm-breakglass	5	2	0	0
Tillerino/Tillerinobot	18	1	0	0
alibaba/cobarclient	8	2	1	0
shyiko/mysql-binlog-connector-java	21	0	6	2
SpigotMC/BungeeCord	24	3	2	0
atermenji/IonicDroid	2	0	4	0
davidmoten/geo	9	0	3	0
alexxyang/shiro-redis	3	1	1	0
groupon/Selenium-Grid-Extras	0	0	0	0
robolectric/deckard-maven	9	0	2	2
forge/roaster	22	5	2	0
jcabi/jcabi-http	36	1	10	61
jprante/elasticsearch-gatherer	0	0	0	0
smola/galimatias	7	0	1	0
headissue/cache2k	5	0	0	0
HolmesNL/kafka-spout	11	0	1	0
recommenders/rival	26	3	1	0
aaberg/sql2o	18	0	0	0
kenglxn/QRGen	44	4	1	7
jmgreen/morphia	25	0	1	8
sarxos/webcam-capture	27	13	14	0
thinkarelius/faunus	31	9	12	1
linkedin/parseq	50	0	0	5
square/dagger	359	145	13	21
Berico-Technologies/CLAVIN	36	4	20	9
lemire/JavaFastPFOR	20	1	1	2
ActiveJpa/activejpa	11	0	4	2
marshall/memoryfilesystem	29	2	1	8
thrau/jarchivelib	3	0	0	2
jberkel/pay-me	1	0	0	0
AdoptOpenJDK/lambda-tutorial	2	0	2	0
caspark/eclipse-multicursor	3	0	0	0
Slim3/slim3	5	0	3	0
ontop/ontop	1259	800	81	142
forge/core	37	37	0	0
alecgorge/jsonapi	38	6	18	12
MilkBowl/Vault	7	0	10	0
square/otto	71	0	81	9
vvakame/JsonPullParser	32	2	8	17
qos-ch/logback	62	3	0	23
write2munish/Akka-Essentials	1	0	2	0
jOOQ/jOOX	10	4	4	0
myabc/markdownj	3	0	0	0
serso/android-common	0	0	0	0
tinkerpop/pipes	7	3	0	0
maxcom/lorsource	343	25	31	13

eclipse-color-theme/eclipse-color-theme	14	7	0	0
sachin-handiekar/jInstagram	77	38	12	1
FasterXML/jackson-annotations	25	8	1	0
JakeWharton/DiskLruCache	21	0	0	4
SomMeri/less4j	23	3	1	6
winterstein/Eclipse-Markdown-Editor-Plugin	16	7	0	0
stratosphere/stratosphere	50	12	16	0
JavaMoney/jsr354-api	26	1	4	0
eirslett/frontend-maven-plugin	128	4	3	7
kuujo/vertigo	2	0	2	2
VCNC/haeinsa	19	2	4	6
Adobe-Consulting-Services/acs-aem-commons	605	153	57	12
indeedeng/proctor	65	6	19	4
iipc/openwayback	244	129	28	13
torakiki/pdfsam	10	1	4	0
knowitall/reverb	0	0	0	0
dropwizard/dropwizard	736	66	15	49
YannBrrd/elasticsearch-entity-resolution	21	5	1	1
jcabi/jcabi-aspects	58	3	9	33
threerings/getdown	22	0	6	0
aled/jsi	4	0	4	0
sanity/tahrir	20	7	13	14
rest-driver/rest-driver	43	4	0	6
helun/Ektorp	86	4	7	0
apache/pdfbox	1	0	1	0
xetorthio/jedis	256	40	2	114
netty/netty	17	17	0	0
google/j2objc	72	19	7	13
apache/storm	2044	1385	330	435
swagger-api/swagger-core	773	106	116	81
eclipse/che	1176	524	0	652
scribejava/scribejava	87	6	1	33
neo4j/neo4j	0	0	0	0
AsyncHttpClient/async-http-client	27	0	0	14
weibocom/motan	87	11	7	7
java-native-access/jna	266	23	0	25
NLPchina/ansj_seg	40	7	26	4
orienttechnologies/orientdb	2132	1802	344	117
apache/zeppelin	0	0	0	0
Atmosphere/atmosphere	269	49	13	178
b3log/solo	300	20	58	0
antlr/antlr4	707	203	360	60
jankotek/mapdb	60	1	0	24
medcl/elasticsearch-analysis-ik	10	1	0	0
apache/flink	3	0	2	2
codecentric/spring-boot-admin	50	15	1	0
Activiti/Activiti	37	26	18	4
twitter/distributedlog	10	1	3	10
rest-assured/rest-assured	35	1	0	6
FasterXML/jackson-jr	21	16	7	6
rombert/desktop-maven-notifier	4	1	0	0
mp911de/logstash-gelf	14	8	0	2

APPENDIX E - Build and Test Conflicts

In this appendix, we present the build and test conflicts identified during our study. Table E.1 presents the build conflicts resulting of errored builds, while Table E.2 those cases resulting of failed builds. The test conflicts are presented in Table E.3.

To access any build, the URL associated to the build is composed by the identifier of the repository owner and the project name. For example, for build *184169929* of CorfuDB, the valid URL is *<https://travis-ci.org/CorfuDB/CorfuDB/builds/184169929>*. In case, it does not work, change the the repository owner identifier for *leusonmario* (commits built during the study).

Table E.1: Build conflicts from errored builds

Project	Build Conflict	BuildID	ParentBuild	ParentBuild	Preserved MS	Motivator
CorfuDB/CorfuDB	"unavailableSymbolFileSpecialCase"	184169929	182952082	184053786	true	contributor
CorfuDB/CorfuDB	"incompatibleMethodSignature"	147270257	146731270	147268879	false	integrator
CloudSlang/cloud-slang	"projectRules"	167353985	167337851	167031283	true	contributor
CloudSlang/cloud-slang	"unimplementedMethod"	158194754	157904090	157651571	true	integrator
CloudSlang/cloud-slang	"unavailableSymbolVariable"	108796563	108795960	108776026	false	integrator
CloudSlang/cloud-slang	"unavailableSymbolVariable"	75952745	75948732	75923848	false	integrator
CloudSlang/cloud-slang	"incompatibleMethodSignature"	52708270	52624715	52621434	true	integrator
CloudSlang/cloud-slang	"incompatibleMethodSignature"	48623977	48452656	47642475	false	integrator
jparsec/jparsec	"unavailableSymbolFileSpecialCase"	298321301	298321674	44292456	false	contributor
spotify/netty-zmtp	"unavailableSymbolFile"	64970950	49887268	64969308	false	contributor
damnhandy/Handy-URI-Templates	"unavailableSymbolMethod"	294178999	3279767	8985944	false	contributor
graphaware/neo4j-reco	"unavailableSymbolFileSpecialCase"	291082366	67313466	291082952	true	contributor
square/javapoet	"incompatibleMethodSignature"	48276050	48264414	48249131	true	contributor
javaparser/javaparser	"malformedProgram"	266734913	266735395	232077865	false	integrator
graphaware/neo4j-framework	"unavailableSymbolFileSpecialCase"	300000699	300001083	300001574	false	contributor
graphaware/neo4j-framework	"unavailableSymbolFileSpecialCase"	299999324	299999578	300000232	false	contributor
hdiv/hdiv	"projectRules"	126140680	126121157	126010037	false	contributor
CloudifySource/cloudify	"malformedProgram"	14913281	14912159	14913175	false	integrator
FasterXML/jackson-databind	"incompatibleMethodSignature"	261011849	176606678	176914535	true	contributor
FasterXML/jackson-databind	"unavailableSymbolFile"	261011849	176606678	176914535	true	contributor
FasterXML/jackson-databind	"unavailableSymbolFile"	243316175	178982350	178982403	true	contributor
FasterXML/jackson-databind	"unavailableSymbolFile"	243309781	214119107	214497429	false	contributor
FasterXML/jackson-databind	"unavailableSymbolFile"	242509582	214851840	215034489	false	contributor
FasterXML/jackson-databind	"unavailableSymbolFile"	231752920	216921134	216949214	true	contributor
searchbox-io/Jest	"incompatibleTypes"	266666842	7348623	266667301	false	contributor
searchbox-io/Jest	"unavailableSymbolFileSpecialCase"	266666842	7348623	266667301	false	contributor
searchbox-io/Jest	"incompatibleMethodSignature"	266665859	266666132	266666455	false	contributor
searchbox-io/Jest	"unimplementedMethod"	266665859	266666132	266666455	false	integrator
FasterXML/jackson-core	"duplicatedDeclaration"	49277833	49277342	49277694	false	integrator
tinkerpop/blueprints	"unavailableSymbolFileSpecialCase"	267838950	7251425	7275268	true	contributor
tinkerpop/blueprints	"duplicatedDeclaration"	267833702	10006381	10033212	true	contributor
tananaev/traccar	"unavailableSymbolFile"	248845938	248846728	127767037	false	integrator
tananaev/traccar	"unavailableSymbolMethod"	232042524	93468836	93456576	true	integrator
tananaev/traccar	"projectRules"	232041300	232041477	145785480	true	contributor
DSpace/DSpace	"incompatibleMethodSignature"	263379307	18006409	17867814	true	integrator
DSpace/DSpace	"incompatibleTypes"	263379307	18006409	17867814	true	contributor
DSpace/DSpace	"unavailableSymbolFileSpecialCase"	263379307	18006409	17867814	true	contributor
yegor256/s3auth	"unavailableSymbolFile"	294115389	294116142	27476470	true	integrator
pac4j/pac4j	"unimplementedMethodSuperType"	291027337	291028699	111151558	true	contributor
pac4j/pac4j	"unavailableSymbolFile"	291027337	291028699	111151558	true	integrator
pac4j/pac4j	"unimplementedMethodSuperType"	291006785	291007948	291008489	true	contributor
pac4j/pac4j	"unavailableSymbolVariable"	291006785	291007948	291008489	true	integrator
pac4j/pac4j	"unavailableSymbolFile"	291001187	291001416	81744123	true	integrator
tumblr/jumblr	"unavailableSymbolFileSpecialCase"	15597349	15448857	13923231	true	contributor
tumblr/jumblr	"unavailableSymbolFile"	15597349	15448857	13923231	true	integrator
OpenGrok/OpenGrok	"unavailableSymbolFile"	260884854	34401647	260885658	false	contributor
perwendel/spark	"incompatibleMethodSignature"	229805514	229805715	181120662	true	integrator
NLPchina/elasticsearch-sql	"incompatibleTypes"	99402562	99401600	98781128	false	contributor
NLPchina/elasticsearch-sql	"unavailableSymbolFile"	260447690	260448041	260448503	false	contributor
NLPchina/elasticsearch-sql	"unavailableSymbolFile"	238652770	238653883	238654712	true	contributor
vavr-io/vavr	"unavailableSymbolMethod"	279623010	279623946	73460929	true	contributor
square/wire	"unavailableSymbolFile"	157352214	151088900	149644087	true	contributor
l0rdrn1kk0n/wicket-bootstrap	"unavailableSymbolFileSpecialCase"	33711220	32203441	33700856	false	integrator
l0rdrn1kk0n/wicket-bootstrap	"unavailableSymbolFileSpecialCase"	11775898	11766897	11753209	false	integrator
square/okio	"unavailableSymbolVariable"	53590047	53512697	51230329	true	contributor
querydsl/querydsl	"unavailableSymbolFileSpecialCase"	120089896	119843675	119841177	true	contributor
querydsl/querydsl	"unavailableSymbolFileSpecialCase"	68942596	68893274	68518956	true	contributor
querydsl/querydsl	"unavailableSymbolFileSpecialCase"	32894479	32644663	32870444	true	contributor
querydsl/querydsl	"unavailableSymbolFileSpecialCase"	28099524	25635609	27786127	false	contributor
querydsl/querydsl	"unavailableSymbolFileSpecialCase"	26703313	26604028	26689825	true	contributor
querydsl/querydsl	"unavailableSymbolFileSpecialCase"	25098248	24127367	24926864	true	contributor
querydsl/querydsl	"unavailableSymbolFileSpecialCase"	245246051	43024946	44344574	true	contributor
oxo42/stateless4j	"malformedProgram"	291334455	291335307	291335657	false	integrator
sanity/quickml	"unavailableSymbolMethod"	53571613	53552160	53495563	false	contributor
sanity/quickml	"unimplementedMethod"	53078374	51295424	52786842	false	integrator
sanity/quickml	"duplicatedDeclaration"	25549352	25544853	25548698	false	integrator
sanity/quickml	"unavailableSymbolVariable"	25549352	25544853	25548698	false	contributor
sanity/quickml	"unavailableSymbolMethod"	261094547	32365230	32339061	false	contributor
google/truth	"unavailableSymbolFileSpecialCase"	25020324	25016734	25016452	true	contributor
timmolter/XChange	"unavailableSymbolFile"	265366487	265367118	265367674	false	integrator
timmolter/XChange	"malformedProgram"	232202095	232202606	232202943	false	integrator
datastax/java-driver	"unavailableSymbolMethod"	144600040	144584703	144580240	false	contributor
datastax/java-driver	"incompatibleMethodSignature"	129572210	128966369	129571864	false	integrator
datastax/java-driver	"unimplementedMethod"	100402057	100359289	100316921	false	integrator
datastax/java-driver	"unavailableSymbolMethod"	70541417	69694110	70541409	false	contributor

datastax/java-driver	"projectRules"	264197134	66828290	67548498	false	contributor
datastax/java-driver	"projectRules"	262590772	66828086	67551312	false	contributor
datastax/java-driver	"projectRules"	262588978	94195206	93340876	true	contributor
datastax/java-driver	"projectRules"	262588129	94448188	95393602	false	contributor
datastax/java-driver	"projectRules"	261827428	66039328	66375528	false	contributor
datastax/java-driver	"projectRules"	261826973	66851336	67551575	false	contributor
datastax/java-driver	"projectRules"	261826640	68517283	68544192	false	contributor
datastax/java-driver	"projectRules"	261825993	91450994	91448081	true	contributor
datastax/java-driver	"projectRules"	261824238	95643827	95849340	false	contributor
datastax/java-driver	"projectRules"	261076398	61156227	61168500	true	contributor
datastax/java-driver	"projectRules"	261075895	61504578	61743199	true	contributor
datastax/java-driver	"projectRules"	261075340	66038248	66380170	false	contributor
datastax/java-driver	"projectRules"	261073520	68544200	68546467	false	contributor
datastax/java-driver	"projectRules"	261072842	80879915	80895179	false	contributor
datastax/java-driver	"projectRules"	261072181	90114055	90354703	false	contributor
datastax/java-driver	"projectRules"	261069264	95528461	95854619	false	contributor
datastax/java-driver	"unavailableSymbolMethod"	201229321	201229321	202731595	false	contributor
dianping/cat	"incompatibleMethodSignature"	262523939	262524153	262524583	true	integrator
protostuff/protostuff	"unavailableSymbolFile"	262560987	262561921	54938495	false	contributor
code4craft/webmagic	"duplicatedDeclaration"	258998355	128642244	258998698	false	integrator
code4craft/webmagic	"duplicatedDeclaration"	243501513	243501975	11121696	true	contributor
code4craft/webmagic	"unavailableSymbolMethod"	237768118	216587617	237243090	false	contributor
square/okhttp	"unavailableSymbolVariable"	19399475	19399433	19398501	true	contributor
graphaware/neo4j-timetre	"incompatibleTypes"	298701803	31937574	31697048	false	contributor
ontop/ontop	"incompatibleMethodSignature"	113479426	111166422	112943276	true	contributor
ontop/ontop	"unimplementedMethodSuperType"	113479426	111166422	112943276	true	contributor
ontop/ontop	"unavailableSymbolMethod"	109369904	101107434	107247494	true	contributor
ontop/ontop	"unavailableSymbolFile"	103065874	93966682	103064869	true	integrator
ontop/ontop	"incompatibleMethodSignature"	101107160	90696395	100866210	true	integrator
ontop/ontop	"unavailableSymbolFile"	101107160	90696395	100866210	true	contributor
ontop/ontop	"unavailableSymbolFileSpecialCase"	90071710	89465997	89465245	true	integrator
ontop/ontop	"incompatibleMethodSignature"	81066148	80620738	80621067	true	contributor
ontop/ontop	"unavailableSymbolFile"	81066148	80620738	80621067	true	contributor
ontop/ontop	"incompatibleMethodSignature"	54693559	51374440	54604897	true	contributor
ontop/ontop	"unavailableSymbolFile"	54693559	51374440	54604897	true	contributor
ontop/ontop	"unimplementedMethodSuperType"	50878615	50833440	43860836	true	integrator
ontop/ontop	"unavailableSymbolFile"	50878615	50833440	43860836	true	integrator
ontop/ontop	"unavailableSymbolMethod"	41598073	39266122	41344362	true	contributor
ontop/ontop	"duplicatedDeclaration"	41367169	41347143	41344362	true	integrator
ontop/ontop	"unavailableSymbolMethod"	30971217	26749094	30728790	true	contributor
ontop/ontop	"unavailableSymbolFile"	26120796	26119005	25772946	true	contributor
Adobe-Consulting-Services/acs-aem-commons	"unavailableSymbolFileSpecialCase"	292436253	292436909	292437960	false	integrator
Adobe-Consulting-Services/acs-aem-commons	"unavailableSymbolFileSpecialCase"	292419256	292425982	292421561	true	integrator
swagger-api/swagger-core	"unavailableSymbolFile"	65300089	65086520	65298512	false	integrator
swagger-api/swagger-core	"malformedProgram"	65291397	64436476	65265579	false	integrator
swagger-api/swagger-core	"unavailableSymbolVariable"	65086450	64955895	65083676	false	contributor
swagger-api/swagger-core	"unavailableSymbolFile"	232521477	230247465	155041327	true	contributor
swagger-api/swagger-core	"malformedProgram"	232516725	185372812	230242288	false	integrator
scribejava/scribejava	"unimplementedMethod"	230252127	181964599	230253435	true	integrator
b3log/solo	"unavailableSymbolFile"	260314193	260315206	134652682	false	contributor

Table E.2: Build conflicts from failed builds

Project	Build Conflict	BuildID	ParentBuild	ParentBuild	Preserved MS	Motivator
apache-hive	"unavailableSymbolFile"-DEP	199377770	197512044	199361292	true	contributor
immutable-immutable	"unavailableSymbolVariable"	240160856	80898938	85470778	false	integrator
google-auto	"methodParameterListSize"	74592079	74524329	237729413	true	integrator
buddycloud-buddycloud-server-java	"unavailableSymbolFileSpecialCase"	33486099	32341190	33482310	false	integrator
DSpace-DSpace	"unavailableSymbolFile"	105741793	105654456	104737262	true	integrator
xetorthio-jedis	"unavailableSymbolVariable"	56165793	56163781	53911782	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	54450658	54371410	54371065	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	54372813	54371441	54366634	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	54366753	54012638	54351283	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	54225499	53823365	53755742	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	50977825	50826294	50922338	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	50512668	50511079	49953864	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	50510220	50385522	50257458	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	49706083	49638114	48271040	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	48995815	48990115	48272327	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	45787398	45787289	44690452	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	44600207	44599135	44346495	true	contributor
querydsl-querydsl	"unavailableSymbolFile"	43494722	43103063	36291679	true	contributor

Table E.3: Test conflicts from failed builds

Project	BuildID	ParentOne	ParentTwo	Type	File Name	Test Case Name	Test Case	Same Methods	Dependent Methods
brettwooldridge/HikariCP	280062604	271260846	280058560	Metric coverage					
	62022698	61160937	61885622	Failed Test	TestMetric	testMetricUsage	updated (Right)	true	true - true
	81124823	81124811	81125648	Failed Test	SerializableTest	decodeGolden	updated (Right)	false	true - false
	229830967	21859057	4378568	Failed Test	ObjectCommandsTest	objectEncoding	new (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	renameOldAndNewAreTheSame	new (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testClusterCountKeysInSlot	new (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testRedisClusterMaxRedirections	new (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testAskResponse	updated (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testDiscoverNodesAutomatically	updated (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testMigrateToNewNode	new (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testCloseable	new (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testThrowExceptionWithoutKey	updated (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testCalculateConnectionPerSlot	updated (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testMigrate	new (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testStablesSlotWhenMigratingNodeOrImportingNodesNotSpecified	updated (Right)	true	false - true
	229830967	21859057	4378568	Failed Test	JedisClusterTest	testRecalculateSlotsWhenMoved	updated (Right)	false	true - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testJedisClusterRunsWithMultithreaded	old	true	false - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testClusterCountKeysInSlot	old	true	false - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testDiscoverNodesAutomatically	old	true	false - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testMigrateToNewNode	old	true	false - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testCloseable	old	true	false - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testCalculateConnectionPerSlot	old	true	false - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testMigrate	old	true	false - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testStablesSlotWhenMigratingNodeOrImportingNodesNotSpecified	old	true	false - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testRecalculateSlotsWhenMoved	old	true	false - true
	229827269	52895148	59431389	Failed Test	JedisClusterTest	testClusterCountKeysInSlot	old	true	false - true
xeorathio/jedis	229826096	74497496	74499479	Failed Test	JedisClusterTest	testRedisClusterMaxRedirections	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testAskResponse	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testThreadPoolConfigAppliesToClusterPools	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testDiscoverNodesAutomatically	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testMigrateToNewNode	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testCloseable	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testCalculateConnectionPerSlot	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testMigrate	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testReturnConnectionOnJedisConnectionException	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testStablesSlotWhenMigratingNodeOrImportingNodesNotSpecified	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testReturnConnectionOnRedirection	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testRecalculateSlotsWhenMoved	old	true	true - true
	229826096	74497496	74499479	Failed Test	JedisClusterTest	testSubscribeOnAllEventsWithListener	old	true	false - false
	75814949	75428682	75811079	Failed Test	SlangImplTest		old	true	
CloudSlang/cloud-slang									