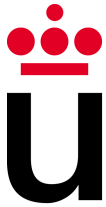


Universidad
Rey Juan Carlos

TESIS DOCTORAL

Towards an Empirical Model to Identify When Bugs are Introduced

Autor :

Gema Rodríguez Pérez

Co-directores:

Dr. Jesús M. González Barahona

Dr. Gregorio Robles

Programa de Doctorado en Tecnologías de la Información y las Comunicaciones

Escuela Internacional de Doctorado

2018

To my family and friends

for their patience and support

II

Acknowledgements

This thesis is the result of three years of work where I have met incredible people from all around the world who have inspired me. Thus, I apologize in advance if I forgot to mention someone important; if we have met and talked, you are also a part of this. I do not even know how to start or how I should start, but let's try...

First of all, I would like to start by thanking my supervisors Jesús and Gregorio. They are great professors and great people who have provided me with a lot of guidance and knowledge, and I am very privileged to have been their Ph.D. student. They have taught me what it means to be a researcher and I have had the invaluable opportunity to learn from their passion, ideas, and professionalism. Their continuous support during my thesis has been a valuable part in my Ph.D., and their advice as well as their assistance in conferences has helped me to establish my own network. They have provided me with the means to continue growing, and I would like to once again express my gratitude. After two Summer stays in Canada and one year visiting the Netherlands, you will finally get rid of me :).

I would like to give special mention to Alicia and Dorealda who have always been there to help. I would also like to give special mention to all the Bitergians; they have made my daily work more orange and funny, and in times of mining more easy with their marvelous tools (advertising!).

From a more personal side, I would also like to say some words in Spanish; it will be easier to understand for my family. Me gustaría das las gracias sobre todo a mis padres, su paciencia, entendimiento y apoyo constante siempre han sido incondicionales. Durante los últimos años hemos aprendido a vivir separados, con las ventajas y desventajas que eso conlleva. Pero sin embargo, a pesar de la distancia, ellos siempre han sabido cómo demostrarme y hacerme llegar su apoyo, tanto en los mejores momentos como en los peores. Desde que me fui del pueblo, siempre han sabido estar cerca en los días más difíciles. ¡Gracias papás! De

la misma manera, me gustaría dar las gracias a mis abuelos, a pesar de que los términos *Doctorado* y *Software* les suenen un poco abstractos, siempre han sabido defender (y en ciertas ocasiones inventarse...) lo que hago. Durante un tiempo, mi abuela explicaba que mi trabajo consistía en encontrar a la gente "mala" que estropeaba el Internet, incluso en ocasiones llegué a ser *hacker* profesional según sus explicaciones, jajaja. También quiero agradecer a mi hermano, y mis amigos, en especial a Maca y Sol, por todo su apoyo, entender la vida de una estudiante de doctorado no es nada fácil, ¡imagínate que incluso trabajamos los fines de semana! Por útimo, quiero dedicar una línea a mi mejor entrenador, siempre ha sabido como motivarme, y si no es por su empuje inicial, quizá no estaría escribiendo esto ahora, así que ¡gracias Casta!

I also would like to thank Alexander Serebrenik and Andy Zaidman. They hosted me during my last year, giving me the opportunity to be part of their research group. My experience in TuDelft and TuE has been personally and professionally enriching. Along with Daniel M. German and Ali Mesbah, they have helped me in improving my language, my knowledge, my research and communication skills along with the lessons I have learned from other cultures. I do not have any doubt that these periods have been the best part in my Ph.D. I would like to thank them for their patience in trying to understand what (sometimes) I wanted to express, and for their tips. During these periods out of my research lab, I had the opportunity to meet so many Ph.D. students and post-docs students. I would like to mention Wesley Torres, Mauricio Verano, Danna Zhang, Andrea Soto, Saba Alimadadi and Quinn Hanam who were my lab mates and with whom I had a lot of fun.

This last paragraph is to summarize my experience. I do not have any clue what is going to happen after finishing my Ph.D, probably a would try to find a post-doc but who knows... However, one of the lessons that I have learned is that academic life is very difficult, you have to be able to handle stress, deadlines, proposals, students, family, friends, and of course, never give up. It is also both challenging and exciting where you start building an academic family to share ideas, research works, projects, travels and fun. In the academia, you are continuously learning and teaching others your work and this is the part I very much enjoy. I do not know if my future would be as a professor but, in this case, I will always remember the lessons and talks that Gregorio has given me, I have had the best role models to follow.

Abstract

Finding when bugs are introduced into the source code is important, because it is the first step to understand how code becomes buggy. This understanding is an essential factor to improve other areas related to software bugs, such as bug detection, bug prevention, software quality or software maintenance. However, finding when bugs are introduced is a difficult and tedious process that requires a significant amount of time and effort, to the point that it is not even clear how to define *"when"* a bug is introduced.

All the bugs are not caused in the same way, and they do not present the same symptoms. Thus, they cannot be treated as equal when locating the bug introduction moment. Some bugs are not directly introduced into the system, and it is essential to distinguish the fact of introducing a bug in a system and when a bug is manifested itself in the system. The first case refers to the moment when the error is introduced into the project, whereas the second case refers to the first moment when the bug manifests itself in the system due to other reasons different from the insertion of buggy code. For instance, when the source code is using, calling external APIs that changed without any previous notification, causing the manifestation of the bug in some parts of the source code.

To distinguish between these moments, this dissertation proposes a model to determine how bugs appear in software products. This model has been proven useful for clearly defining the code change that introduced a bug, when it exists, and to find the reasons that lead to the appearance of bugs. The model is based on the concept of when bugs manifest themselves for the first time, and how that can be determined by running a test; it also proposes a specialized terminology which helps to analyze formally the process. The validity of the model has been explored with a careful, manual analysis of a number of bugs in two different open source systems. The analysis starts with changes that fixed a bug, from which a test to determine whether or not the bug is present is defined. The results of the analysis have

demonstrated that bugs are not always introduced in the source code, and this phenomenon should be further investigated to improve other disciplines of software engineering. Furthermore, the model has also been put in the context of current literature about the introduction of bugs in source code. An interesting specific result of the model is that it provides a clear condition to determine if a given algorithm for identifying the change introducing a bug is correct or not when performing the identification. This allows (i) to compute the *"real"* performance of algorithms based on backtracking the modified lines that fixed a bug, and (ii) a sound evaluation of those algorithms.

Resumen

Es importante encontrar cuándo se introducen los errores en el código fuente, porque éste es el primer paso para comprender cómo el código se vuelve defectuoso. Sin embargo, encontrar realmente cuándo se introducen los errores es un proceso difícil y tedioso que requiere una gran cantidad de tiempo y esfuerzo, hasta el punto de que ni siquiera está claro cómo definir *"cuándo"* se introduce un error en el sistema.

Los errores no se causan de la misma manera, y no presentan los mismos síntomas. Por lo tanto, no pueden tratarse de la misma forma al identificar el momento de introducción del error. Además, algunos errores no se introducen directamente en el sistema, y es esencial distinguir el hecho de introducir un error en un sistema, del hecho de que un error se manifieste en el sistema. El primer caso se refiere al momento en que se introduce el error en el proyecto, mientras que el segundo caso se refiere al primer momento en que el error se manifiesta en el sistema debido a otras razones diferentes de la inserción del código erróneo. Por ejemplo, cuando se usa el código fuente, o se llama a APIs externas que fueron modificadas sin notificación previa, lo que provocó la manifestación del error en algunas partes del código fuente.

Para distinguir entre estos momentos, esta tesis propone un modelo para determinar cómo aparecen los errores en los sistemas de software. Este modelo ha demostrado ser útil para definir claramente el cambio de código que introdujo un error, cuando éste existe, y para descubrir otras razones que conducen a la aparición de errores. El modelo se basa en el concepto de identificar cuándo se manifiesta el error por primera vez, y cómo se puede determinar ese momento a través de ejecutar una prueba de test; también propone una terminología especializada que ayuda a analizar formalmente el proceso. La validez del modelo ha sido explorada mediante el cuidadoso análisis manual de una serie de errores en dos sistemas de código abierto. El análisis comienza con cambios que corrigieron un error, a partir del cual se define

una prueba para determinar si el error está presente o no. Los resultados del análisis han demostrado que los errores no siempre se insertan en el código fuente y que hay diferentes motivos para ello, por lo que este fenómeno debería investigarse más a fondo para mejorar otras disciplinas de ingeniería de software. Además, el modelo también se ha puesto en el contexto de la literatura actual sobre la introducción de errores en el código fuente. Un resultado interesante y expecífico extraído del modelo es que proporciona una condición clara para determinar si dado un algoritmo para identificar el cambio que introduce un error es correcto o no en la identificación. Esto permite calcular (i) el rendimiento *"real"* de los algoritmos basados en el seguimiento de las líneas que han sido modificadas para corregir un error, y (ii) evaluar esos algoritmos.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software evolution is a very active field of research in software engineering [Mens, 2008]. The term *software evolution* lacks a standard definition [Bennett and Rajlich, 2000], and it is often used as synonym of the term *software maintenance*. The basic operations of software maintenance are software modifications in order to, e.g., fix bugs, adapt when external components change or add features. In 1974, Brooks states that "over 90% of the costs of a typical system arise in the maintenance phase, and that any successful piece of software will inevitably be maintained" [Brooks, 1974]. Almost 30 years later this statement has not changed, [Erlikh, 2000] reports that software maintenance activities consume approximately 90% of the costs of a software system.

Finding the software bugs[1] is one of the software maintenance activities that has occupied and will occupy much of the daily development and maintenance tasks of software developers. Software systems have always contained bugs, and the industry average bugs rate when it comes to creating software is around 1 to 25 bugs for every 1,000 lines of code [McConnell, 2004]. Software bugs have an impact direct and indirect in the costs of a company such as the customer loyalty, brand reputation, wasted time in development and maintenance phases, etc. According to the report form the National Institute of Standards and Technology in 2002, software bugs cost around $59.5 billion to the U.S. economy annually [Planning, 2002]. This reveals the necessity to understand how bugs are introduced into the source code to minimize the chance of defects being introduced.

The bugs have been studied since the early 70's in software reliability [Jelinski and Moranda, 1972].

[1] Likewise Li *et al.* [Li et al., 2006], we use the words "bug" and "error" interchangeably

Software reliability researchers developed models for predicting software reliability based on the observation of software product failures. However these models ignore information regarding the developments of the software, the environmental factors or the method of failure detection [Pham, 2000].

A common practice in Empirical Software Engineering *(ESE)* to understand how bugs are introduced, is based on the study of their reports, and fixes. Specifically, the study of the changes that fix a bug is an interesting practice which allows researchers to understand the importance of locating when a bug was inserted since it has many implications in different areas of Software Engineering and come up with new techniques that practitioners can used. For example, determining why and how a bug is introduced may help to identify potential software modifications that introduce bugs [Śliwerski et al., 2005b], [Kim et al., 2006c], [Zimmermann et al., 2006], [Thung et al., 2013], [Sinha et al., 2010]. This identification could lead to the discovery of methods to avoid software bugs [Nagappan et al., 2006], [Hassan, 2009], [Zimmermann et al., 2007], [Hassan and Holt, 2005], [Kim et al., 2007], or it may help to identify who is responsible for inserting the bug. Identifying the responsable for inserting a bug has the potential to propagate self-learning and peer-assessment processes [Izquierdo et al., 2011], [da Costa et al., 2014], [Ell, 2013]. This study also help to understand how long a bug is present in a code, thereby enabling researchers and developers quantify the software quality and avoiding the misleading [Chen et al., 2014], [Weiss et al., 2007]. Due to these reasons, the study of software bugs has been very active during the last decades.

Despite empirical investigations into software bugs of systems [Chou et al., 2001], [Sahoo et al., 2010], [Kamei et al., 2013], [Chen et al., 2014], [McIntosh et al., 2016], [Wan et al., 2017], and the development of a handful of techniques and tools to detect and prevent software errors [Śliwerski et al., 2005a], [Hovemeyer and Pugh, 2004], [Thung et al., 2014], [Kim et al., 2007], understanding what line or lines of the source code introduced the bug is still a challenge. The researchers are merely adopting a popular assumption where it is established that the modified lines to fix the failure are likely the cause of the bug. Many researchers in ESE start with this assumption to conduct studies on identifying the bug introduction change through navigating back the modified lines of the change that fixed the bug.

However, there are factors that can cause a bug to be fixed in many different ways. For instance, a bug fixed before release and a bug fixed during the planning phase are likely to be

fixed in a different way [Murphy-Hill et al., 2015]. Thus, it is possible that the lines modified to fix the bug were not contained the bug and backtracking them lead to erroneous results. As consequence, studies in areas such as prediction and localization may differ their results. Moreover, it is not always clear whether these studies and tools are defining what a *" true "* bug is, *"when"* a bug is introduced, and what *"'to introduce"* a bug means. Kim *et al.* argues that the fact of introducing a bug depends on the definition of bug, and the future work should verify whether the introduction of bug meets a given definition of bug [Kim et al., 2006c]. But as fas as we know, nobody has conducted such verification. Moreover, Mens *et al.* claims that *"it is necessary to develop new theories and mathematical models to increase understanding of software evolution, and to invest in research that tries to bridge the gap between the what (i.e., understanding) of software evolution and the how (i.e., control and support) of software evolution"* [Mens et al., 2005].

In this thesis, we develop such a new theory envisioned by Mens *et al.* [Mens et al., 2005] and Kim *et al.* [Kim et al., 2006c] to investigate and better understand how bugs appear in software products. This theory describes a model of bug introduction and defines what a *"TRUE"* bug is. The model is based on the concept of when bugs manifest themselves for the first time, and how that can be determined by running a test. Furthermore, this model provide developers with means of validating whether the line identified as cause of the bug was in fact inserting the bug. For that, we use the information on how failures are reported, discussed and managed in issue tracking systems. Although useful information to understand the bug may be missed or not documented in these systems [Aranda and Venolia, 2009], Panichella *et al* observed that very often in open source projects, developers tend to use these issue tracking systems and mailing lists to discuss and communicate the reasons and fixes of the bug with each other [Panichella et al., 2014a],[Panichella et al., 2014b]. Thus, these systems may provide a textual description regarding the *symptoms* of the bug that helps researchers to understand whether the bug was inserted, and how it was inserted. Then, this information can be linked to the source code system that provides the *treatment* through changing the source code to fix the error, that helps researchers to understand how the bug was fixed and whether this code caused the bug.

1.1 Context

This section aims to contextualize this thesis. There are two subsections that describe the bug fixing process, and the well-known SZZ algorithm used to identify the changes that introduced bugs. Then, we detail the research goals, and enumerate the contributions.

1.1.1 Bug Introducing and Bug Fixing

Bug introducing activity encompasses the process when a developer unintentionally introduces an error while describing the problem or developing software artifacts. This error may result in an non-desired event from the users' point of view or a unexpected state in the system which will be reported in an issue-tracking system such as GitHub, BugZilla or Jira, and fixed in the software. On the other hand, the bug fixing process encompasses the process when someone discovers a bug and creates the bug report that describes the malfunction. Then, the bug report is assigned to a developer in order to produce a Bug-Fixing Commit (*BFC*) that fixes the bug. The *BFC* might be seen as a possible way to determine why a software component was behaving erroneously. Finally, once the bug is resolved, another developer verifies the *BFC* and closes the bug report. Thus, the bug fixing process may help to identify the part of the component that caused the erroneous behavior. When a *BFC* is completed, it is a common practice to record this change in a Version Control System (*VCS*) and add it to a repository.

Figure 1.1 is a bug report of ElasticSearch that uses GitHub as issue tracking system. On the left side, the image *(a)* describes the bug, there is the label *"Bug"* which implies that the issue should be a bug report. On the right side, the image *(b)* shows all the events that transpired between the time when the bug was reported and when the bug was closed. There is a comment explaining the root cause of the bug as well as the identifier of the commit that fixed the bug.

Figure 1.2 *(a)* shows the message of the *BFC #28316*[2], this information is accompanied by the diff of the files that were modified to fix the bug in GitHub. Figure 1.2 *(b)* shows the log entry of this *BFC* from the git repository of ElasticSearch project. This commit log can be obtained using the command *git show* and the number of the commit. Git's log entry includes

[2] `https://github.com/elastic/elasticsearch/commit/ba5b5832039b59`

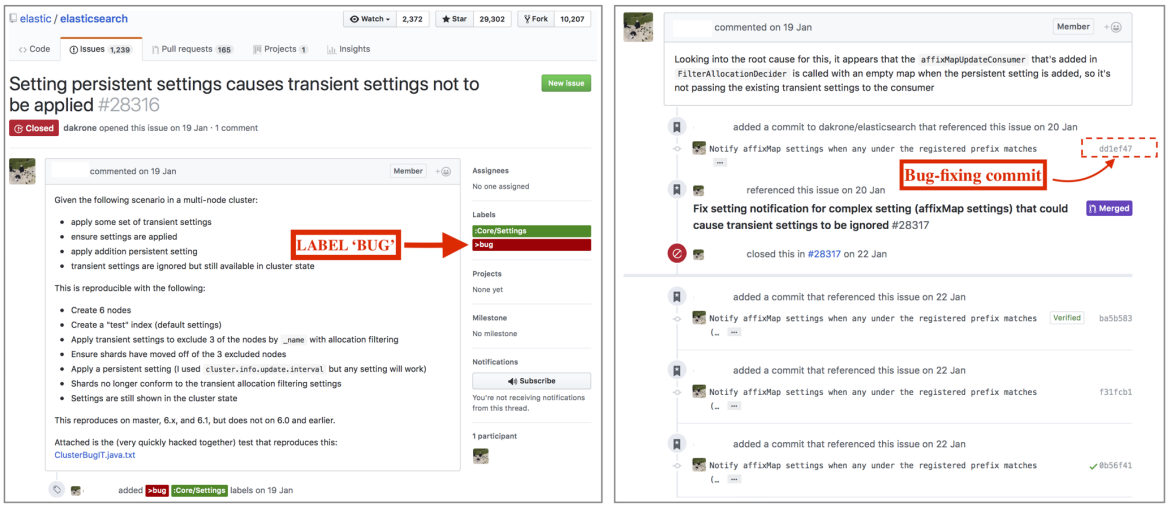


Figure 1.1: Bug Report in ElasticSearch

the author, date, and files involved in the change. Additionally, a text message describing the change is also recorded. This message is the same as the message on GitHub.

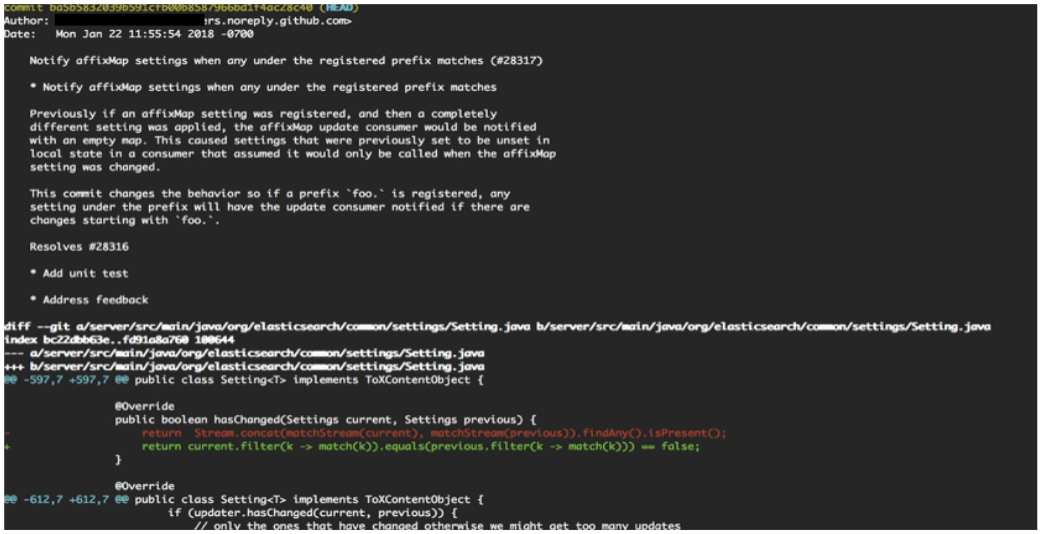
However, being able to identify the change in the code that first caused a bug is another story, which, when analyzed in detail, proves to be in many cases difficult to tell. Researches have made a lot of effort to understand and locate the changes that introduced bugs, but this process is not easy because there are many factors that prevent the success of this process. For instance:

1. Software systems and their architecture are continuously evolving and becoming more complicated over time. This leads to problems that creep into a system and manifest themselves as bugs [Le, 2016]. It also leads to the manifestation of failures in unchanged parts [German et al., 2009].

2. The use of component-oriented development model leads to the implementation of software products that are an assemblage of small components from many different sources. This makes estimating the behavior of a complete system tedious when a component behaves erroneously [Duraes and Madeira, 2006].

3. The accuracy of current approaches rely heavily on manual analysis where only experts can judge whether the identified changes are responsible for inserting the bug. In most of the cases, this analysis is impractical and there is no established way to know exactly

(a) Bug-fixing commit in GitHub



(b) Bug-fixing commit entry in Git

Figure 1.2: Bug-Fixing Commit (*ba5b5832039b591*) in ElasticSearch

which line introduced the bug [da Costa et al., 2016].

Many studies in the area of software maintenance and evolution assume that "modified lines in a *BFC* are likely the ones that introduced the bug" [Zeller, 2003], [Śliwerski et al., 2005b]. Although the software engineering community has suspected that this assumption is not always true, it is frequently found in the state-of-the-art literature. However there is not enough empirical evidence supporting it and there still are little proofs to help researchers and practitioners understand under what circumstances this assumption is not held. Recent studies have demonstrated that there are limitations when flagging potential changes to be the bug

introducing change [da Costa et al., 2016].

The consequences of the lack of understanding on which line or lines introduced a bug are manifold. For instance, bug prediction and bug localization may not capture the true cause of failures as these models are based on how developers have fixed bugs in the past. In addition the evaluation may not be accurate when computing various metrics to evaluate the software quality based on how many bugs practitioners fix, or how many bugs developers introduce according to their experience and activities in the project. Thus, a complete understanding of how, when and why bugs are introduced allows to improve advanced techniques that harness information to learn patterns of these changes, to motivate the design and development of better mechanisms, or to help in the automation of bug predictors that can estimate the likelihood of fault introduction.

Fortunately, in modern software development, many traces can be retrieved on how code changes, and how bugs are fixed. As a result, this trace information is at our disposal allowing us to better understand the reasons why a change was required to fix a bug from the issue tracking systems and VCS. The issue tracking system is used to record such issues as bug reports, feature requests, maintenance tickets and even design discussions. Therefore, when researchers want to conduct studies on bug introduction, they need to first identify the bug reports from other kind of issues [Herzig et al., 2013], since analyzing all of them together may lead to biased results [Bird et al., 2009a]. A bug report stores all comments and actions on the issue tracking system, for example, discussions on a fix in the code review system, or in some cases, information regarding the final *BFC* that closes the bug report. Depending on the policy of each project and the bug tracking system used in the project, developers can tag each issue with different labels[3] in order to distinguish bug reports from other kinds of issues report. In this way, the researchers may conduct more reliable researches when identifying *BFCs*, as they can use the expertise criteria of developers to distinguish bugs from other issues. For example, *ElasticSearch* has the policy to label the reports describing a bug with the tag *"Bug"*, accelerating and ensuring that the researcher's decision process is much more reliable. However, when projects do not use these tags or the bug tracking system does not allow them, researchers have to use regular expressions to locate the bug reports [Śliwerski et al., 2005b]

[3]there are different uses for labels, the distinction between bugs and features is just an example [Trockman, 2018]

or to use automatic classification systems [Antoniol et al., 2008]. Nevertheless, the vocabulary that describes the cause of the reports varies from project to project making it difficult to establish an unequivocal method to distinguish them.

The VSC Source code management systems also store the source code and their differences across different versions of the source code. They store metadata such as user-IDs, timestamps, and commit comments. This metadata explains who, how, and when the source code changed.

Empowered with all these information, the history of a software component can be navigated to identify when a failure was occurring for the first time. This thesis uses these information to cover the process of finding out what malfunction caused the change, and help to identify what caused the bug in software products, and finally how it was fixed.

1.1.2 Brief Introduction to the SZZ Algorithm

In Software Engineering research, the SZZ algorithm [Śliwerski et al., 2005b] is a popular algorithm for identifying the origin of a bug [da Costa et al., 2016]. It was proposed by Śliwersky, Zimmermann, and Zeller in 2005 to identify the suspicious change to induce the later fix. The algorithm identifies the Bug-Fixing Commit (*BFC*), then uses a diff tool to compare the lines that differ between two revisions of the same file. In the SZZ, the authors assume that the lines that have been removed or modified in the *BFC* are the ones containing the bug. For this reason, SZZ traces back the lines through the code history (by means of the annotate/blame[4] command), to find when the changed code was introduced.

Even though the algorithm addresses two different problems, it can be split into two main parts. The first part is related to the problem of linking to the VCS and the issue tracking system to identify the *BFC*. In this part, the algorithm identifies -by means of a set of heuristics- *BFCs* through employing a technique that matches commits with bug reports labeled as fixed in the bug tracking system. Therefore, the algorithm uses regular expressions to identify bug numbers and keywords in the commit messages that are likely to point out a real *BFC*. The second part addresses the problem of identifying the change that introduced the bug, the Bug-Introducing Commit (*BIC*). In this part, the algorithm employs the diff functionality implemented in the source code management systems to determine the lines

[4] Annotate is used in SVN and blame is used in Git

that have been changed (to fix the bug) between the fixed version and its previous version. Then, using annotate/blame functionality, SZZ is able to locate which change modified or deleted those lines for the last time in previous commit(s). These change(s) are flagged as suspicious of being the *BICs*.

Despite being a fundamental algorithm in the community to locate the *BICs*, the results obtained after its application are limited. Firstly, there is not enough empirical evidence supporting the assumptions suggested by the SZZ, and the current evaluations are limited; Secondly, this algorithm fails in identifying the *BIC* in some scenarios, i.e, when new lines in the *BFCs* cannot be traced back, these types of commits are removed from the analysis. Thirdly, despite there are some studies facing challenges with this algorithm [Kim et al., 2006c], [Williams and Spacco, 2008], [da Costa et al., 2016], all of them have the same assumption: " The lines changed to fix a bug are the ones containing the malfunction".

1.2 Research Goals

Finding the cause of bugs has been an important topic during the last decades. The high importance and impact of this topic is an essential factor to understand and improve other areas related to bugs, such as bug detection, bug prevention, bug analysis and bug statistics. Many questions related to solving the problem of identifying the *BIC* are proposed in this dissertation. All the bugs are not caused in the same way, and they do not present the same symptoms. Thus, they cannot be treated as equal when locating the cause of the bug.

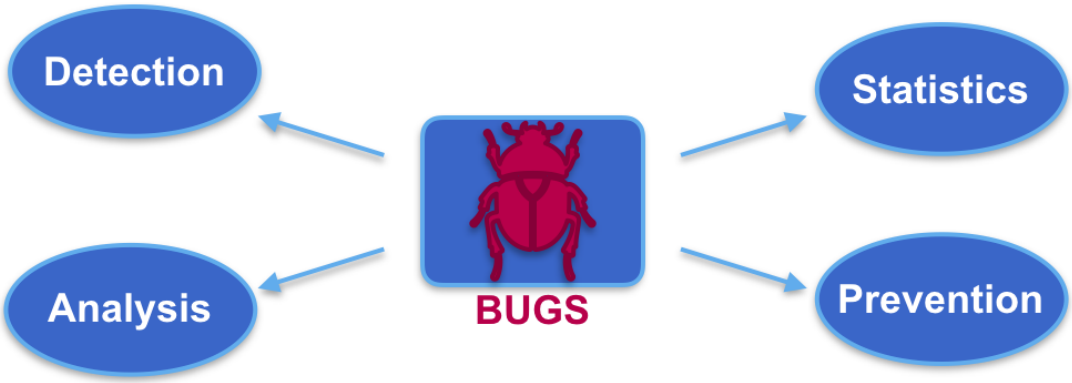


Figure 1.3: Affected areas of studying the cause of a bug

There are many studies and approaches based on backtracking the lines of a *BFC* to locate the origin of bugs. Nevertheless, these approaches are not using any meaningful model that researchers might use to validate the "real" performance of the current state-of-the-art algorithms. These algorithms attempt to locate the line that "contains the bug". And researchers cannot be sure about the meaning of "injecting a bug", because any previous study states the fact of introducing a bug, or the moment of when it was introduced. Consequently, researchers are not sure whether the bug was introduced in the moment of inserting the lines into the source code. For that reason, in this dissertation, we distinguish between bug manifestation moment (*FFM*) and the bug introduction moment (*BIC*). It is importan to establish the distinction between both moments because although they can be the same, it is possible that they may differ. For example, Alice inserted a line to the project that opens the html code of a website, one weeks later Bob reported that the website was different from what users expected and he fixed this line. 1) The bug introduction occurs whether the *URL* that Alice wrote is incorrect, at this moment the error was introduced by Alice and it manifested itself in the project, although it was not notified until one weeks later. 2) The bug introduction does not exist whether the *URL* that Alice wrote is correct and due to other reasons, the website has been removed or suspended by the server administrator some days before. In this scenario there is no bug introduction moment when Alice wrote the line, but there is only a first failing moment in which the bug manifested itself in the project. This first failing moment is a software change done in the project after the website had been removed.

To address the lack of definitions and the need to validate the current algorithms, this thesis proposes a theoretical model to define which changes introduced bugs into the source code. This model assumes a perfect test which can be run indefinitely in the history of the project to find out when the failure was introduced and whether the change was responsible for the bug or it was something external. Furthermore, this model may be used as a framework to validate the performance of other approaches and researchers can compare the effectiveness of different algorithms. Setting this framework is one of the principal values of the thesis because the previous literature is not able to compute the "real" precision and recall of the current algorithms.

1.3 Contributions

The main four contributions of this thesis are outlined below.

1. **A Systematic Literature Review *(SLR)* on the Use of the SZZ algorithm:** We have carried out a study to analyze reproducibility and credibility in Empirical Software Engineering (*ESE*) with the SZZ algorithm as case of study. The aim of the SLR is to obtain an overview of how authors have addressed the reproducibility and credibility in the studies where they have used the SZZ algorithm. This SLR has been published in the Information and Software Technology Journal in March, 2018 [Rodríguez-Pérez et al., 2018] and the goals of the study are described below:

   (a) **An overview on the impact that the SZZ has had so far in ESE:** The SZZ algorithm has been shown to be a key factor to locate when a change introduced a bug. Furthermore, to provide insight of how widespread the use of SZZ is, we also addressed the maturity and diversity of the publications where SZZ has been used in order to understand its audience.

   (b) **An overview of how studies that use the SZZ algorithm address the reproducibility in their research work:** Reproducibility is a crucial aspect of a credible study in ESE [González-Barahona and Robles, 2012]. Piwowar et al. state that reproducibility improves the impact of research [Piwowar et al., 2007]. In addition, when a research work incorporates reproducibility, it is more likely to be replicated. However, there is evidence in the ESE literature that replicable studies are not common [Robles, 2010]. By providing a replication package, the authors facilitate others to replicate or to reproduce their experiment, which increases the credibility of their results [Juristo and Vegas, 2009].

   (c) **An analysis of how these studies manage the limitations of the SZZ algorithm:** Limitations of SZZ are well-known in the research literature, and we would like to analyze how many papers report any of the limitations or how they address them. Therefore, we study whether authors are aware of that and mention that some limitations of SZZ may affect their findings, be it in the description of the method, in the threats to validity or in the discussion.

2. **A Theoretical Model to Identify the Bug-Introduction Changes:** This dissertation describes a comprehensive model to identify how bugs are introduced into the source code. This model includes a set of definitions which formally helps to analyze the process. Furthermore, it also includes an exploratory taxonomy that helps in understanding how a bug is introduced and manifested into the source code.

   The goals of the model are:

   (a) **A detailed definition of the Bug-Introducing Change and the First-Failing Moment:** The model introduces a general method to determine, unequivocally and falsifiability, the first time that the software fails in relation with the bug-fixing change, identifying the *BIC* when it exists.

   (b) **The criterion to apply the model:** The model relies on the existence of a hypothetical test that can be run indefinitely in each past version of a project to check whether or not the code was buggy at this point. Since the hypothetical test is not automatized, the criterion describes how it should be run and the possible outcomes after applying it.

3. **Empirical Study on the Application of the Proposed Model:** This dissertation presents an empirical study of the proposed theoretical model. This study analyses the publicity of available data sources from two open source software projects, Nova and ElasticSearch and describes a model to identify the bug introduction commit or to determine whether it exists given a *BFC*. The goals of the empirical study are:

   (a) **The frequency of *BFC* induced by *BIC* in Nova and ElasticSearch:** The empirical study manually identifies whether a *BFC* was induced by a *BIC*, or whether other reasons may explain the cause of the failure. Thus, the frequency of a *BIC* being induced by a *BIC* in the two cases of study can be computed.

   (b) **The "gold standard" dataset:** The proposed model enables to define the "gold standard" to be defined where commits in the repositories are *BIC*. This dataset favors the comparison with the performance of some state-of-the-art approaches to computes the "real" false positives and false negatives.

4. **Quantification of the SZZ Algorithm:** This thesis studies and contextualizes the current problem of identifying a *BIC* given a *BFC* using algorithms based on backtracking the lines that were modified to fix the bug. Furthermore, there is quantification of the possible sources of error when identifying the *BICs* using the SZZ algorithm or some variants of it. Since the thesis defines the "gold standard", we can compute the "real" performance of the SZZ algorithm when identifying the *BICs* because we are sure of the "real" true positives and true negatives in their results.

1.4 Structure

The remainder of this thesis is organized as follows.

Chapter 2 provides a detailed description of the current state of the art to the reader. Then, Section 2.2 mentions some studies in the filed of bug seeding and bug localization. Finally, Section 2.3 details some bibliography related to the SZZ algorithm.

Chapter 3 addresses the current problem to identify the first moment when the system fails by providing motivating examples and explaining the reasons why algorithms sometimes fail when locating the Bug-Introducing Commits (*BICs*). Furthermore, this Chapter explains the role of the VCS in the bug seeding activity.

Chapter 4 discusses the reproducibility and credibility of the studies that used the SZZ in the ESE. This Chapter draws an overview of the impact that this algorithm has in the ESE community and how researchers use it in different fields to identify the *BIC*.

Chapter 5 contains a detailed description of our proposed approach to deal with the inaccurate algorithms to locate the moment when the bug manifest the failure for the first time. This theoretical model allows for a better framing of the comparison of automatic methods to find *BICs*. This model may distinguish between the bug introduction moment and bug manifestation moment.

Chapter 6 applies the proposed theoretical model into two cases of study: ElasticSearch and Nova. Both projects are open source projects with many thousand of actives developers. Section 6.2 describes the methodology used in the studies to explain how manually identify the bugs that were injected into the source code by navigating back into the lines of code that were modified in the Bug-Fixing Commit (*BFC*) to fix the bug. Finally, Section 6.3 presents

the results of the empirical study after applying the theory of bug introduction.

Chapter 7 details the threats to validity of this dissertation in Section 7.1 and then it discusses each of the results obtained in this thesis in Section 7.2.

Finally, Chapter 8 draws some conclusions and concludes with the potential further work to be done.

Chapter 2

State of the art

This chapter outlines an overall picture of the bug life cycle, bug seeding and bug localization process in the Software Engineering *(SE)*. The information from the bug seeding process may be used in other areas of *SE* such as bug prediction, bug triage or software evolution. This chapter describes the related work necessary to understand how this thesis fits into the current literature. It explains the life cycle of a bug and the research works that other authors have carried out to locate the origin of bugs.

2.1 Bug Life Cycle

In this section the main focus is on understanding the bug's life cycle. Sommerville [Sommerville, 2010] claims that it is not possible to avoid the unintentional introduction of bugs in the source code because it is inherited in the process of software making. To help with this process, several tools and products have been developed in order to reduce the number of bugs and improve the software development process.

Generally, the open source projects leave the management of issues to specific tools such as the issue tracking system. The open source systems studied in this dissertation use Launchpad and GitHub as their issue tracking systems, although the best-known issue tracking system is Bugzilla[1]. For example, in Bugzilla the life of a bug starts when a developer or user detects a wrong behavior and reports it to the system. The initial status of the report is UNCONFIRMED. Developers legitimatize the status by reproducing the symptoms described

[1] ww.bugzilla.org/

in the report where it is confirmed, meaning that the bug is real. The status is changed to NEW, and the bug is considered open from here onwards. An open bug's status is changed to ASSIGNED once it is assigned to a developer for fixing. The status of a bug changes to RESOLVED when its resolution is either: FIXED, DUPLICATE, WONT-FIX, WORKS-FORME, INVALID, REMIND, LATER. Next, a Quality Assurance (QA) person might verify the resolution by either accepting it or rejecting it, turning the outcome to either VERIFIED or REOPEN. Finally, when a bug is labeled as VERIFIED it can be marked as CLOSED which concludes the bug resolution process. Although these steps describe the common process of a bug, there are other possible paths in the life cycle of a bug.

During this cycle, the developers and users might discuss about the possible cause of the bug or the reason why the project manifests the failure. Thus, the greater the understanding of a bug's life cycle, the greater the explanation of it origin.

2.1.1 Characteristics of bugs

With this respect, previous works have studied bug characteristics in large software systems [Chou et al., 2001], [Gu et al., 2003], [Ostrand and Weyuker, 1984], [Ostrand and Weyuker, 2002], [Podgurski et al., 2003], [Sullivan and Chillarege, 1992], [Chen et al., 2014], [Li et al., 2006], [Beizer, 2003], [Tan et al., 2014]. Recent papers have performed empirical studies to characterize and classify the bugs in open source software depending on the different challenges that arise in the bug finding process. Lu *et al,.* classified bugs in three categories depending on their root cause: Semantic, Concurrency, and Memory [Lu et al., 2005]. Then, Li Tan *et al,.* extended the previous root cause classification by adding more cases and introducing two more dimensions, Impact and Software Component [Tan et al., 2014]. Table 2.1 shows the relationships between the root cause and the fault, the impact and the failure, and the component and the location of the bug [Li et al., 2006]. Finally, Chen et al., included additional sub-categories after manually studying the impact of dormant bugs in software quality; dormant bugs refers to errors that are introduced in a version of the software system, but they are not found until much later [Chen et al., 2014]. Moreover, table 2.2 shows the classification of the root cause of failures defined by [Asadollah et al., 2015], [Chen et al., 2014], [Lu et al., 2005].

Many authors have used this classification to deal with different purposes regarding au-

Table 2.1: Bug categories of the three dimensions: Root Cause, Impact, and Component

Dimension	**Subcategory**	**Description**
Root Cause	Memory	Improper handling of memory objects.
	Concurrency	Synchronization problems
	Semantic	Inconsistencies with requirements or programmers' intention
Impact	Hang	Program keeps running but does not respond.
	Crash	Program halts abnormally.
	Data Corruption	Mistakenly change user data.
	Perfor. Degradation	Functions correctly but runs/responds slowly.
	Incorrect functionality	Not behaving as expected.
	Other	other impacts.
Software Component	Core	Related to core functionality implementations.
	GUI	Related to graphical user inter- faces.
	Network	Related to network environment and communication.
	I/O	Related to I/O handling.

Table 2.2: Categories of root causes of bugs found in [Asadollah et al., 2015], [Chen et al., 2014], [Lu et al., 2005]

Category	**Subcategory**	**Description**
Concurrency	Data race	Two or more threats access to write the same data
	Atomicity-related	A concurrent overlapping execution between two sequences
	Deadlock	A process depend by another process to proceed
	Order Violation	Violation of the desired order
	Livelock	A thread is waiting for an unavailable resource
	Starvation	A process indefinitely delayed
	Suspension	A calling thread waits for a long time
Memory	NULL Pointer Dereference	Dereference of a null pointer
	Memory leak	Failures to release unused memory
	Uninitialised Memory Read	Read memory data before it is initialized
	Dangling Pointer	Pointers still keep freed memory addresses
	Overflow	Illegal access beyond the buffer boundary
	Double Free	One memory location is freed twice
Semantic	Missing Cases	A case in a functionality that is not implemented
	Missing Features	A feature that is not implemented
	Corner cases	Incorrect or ignored boundary cases
	Wrong control flow	Incorrect implementation of sequences of function calls
	Exception handling	Do not have proper exception handling
	Processing	Incorrect Evaluation of expressions/equations
	Typo	Typographical mistakes
	Design Issue	Incorrect Design or API/function
	Incorrect Documentation	Incorrect/inconsistent documentation
	Other	Any other semantic bug

tomatic approaches for software bug classification, as well as empirical and methodological studies on the case of software errors, taxonomic studies for software bugs and the study of bug characteristics in the OSS. Vahabzadeh *et al,.* attempted to understand the characteristics of bugs in test code. In this study the authors also described the *Environment* category when referring to tests that pass or fail depending on the operating system and the incompatibilities between different versions of JDK. They discovered that incorrect and missing assertions are the main root cause of dormant bugs [Vahabzadeh et al., 2015]. Jeffrey *et al,.* developed a technique to automatically isolate the root cause of memory-related bugs; their approach is effective in finding root causes when memory corruption propagates during execution until a failure crash occurs [Jeffrey et al., 2008]. Wan *et al,.* studied 1,108 bug reports in order to understand the nature of the bug, they also introduced other categories such as *Security*, *Environment and Configuration*, *Build*, *Compatibility* and *Hard Fork*. Their findings indicated that security bugs took the longest median time to be fixed, and also that the environment and configuration bugs were one of the major types of bugs along with the semantic bugs [Wan et al., 2017].

On the other hand, some authors have also considered using other characteristics to classify bugs. Sahoo *et al,.* used the reproducibility of a bug, the observed symptoms and the number of inputs needed to trigger the symptom and distinguish between deterministic or non-deterministic bugs [Sahoo et al., 2010]. Chandra and Chen distinguished environment-dependent from environment-independent bugs in the Apache, GNOME, and MySQL [Chandra and Chen, 2000]. Zhang *et al,.* computed the bug fixing time and identified factors that influence it on three open source software applications. They found the assigned severity, the bug description, and the number of methods and changes in the code as impacting factors.

2.2 Bug Localization

To locate the origin of bugs, researchers have proposed two different procedures. Some researchers use techniques starting with a Bug-Fixing Commit (*BFC*) in order to identify the most recent change(s) that was modified to fix the bug. Other researchers use techniques to locate root causes of software failures by analyzing program traces. These approaches do

not rely on identifying the *BFC*, instead they attempt to find an association between program failures and the execution of program elements.

In this section discusses the state of the art procedures to identify *BFCs* and the changes that introduced the bug, also called Bug-Introducing Commits (*BIC*). Although this thesis is mainly motivated by the techniques that use a *BFC* to identify the commit that caused the bug, other existing and popular techniques are also discussed.

2.2.1 Identifying the Bug-Fixing Commit (*BFC*)

Previous works have attempted to identify *BFCs* in version archives. Mockus and Votta performed an analysis to identify the reasons for software changes using historic databases; they used the textual description in the log of a commit to understand why the change was performed [Mockus and Votta, 2000]. Cubranić and Murphy recommended a practice that uses a bug report number in the comment when the practitioners fix a bug report, thereby linking the changes with the bugs [Čubranić and Murphy, 2003]. Finally, Śliwersky *et al.* proposed the *SZZ* algorithm that links a version archive to a bug database in order to automatically identify and analyze fix-inducing changes [Śliwerski et al., 2005b]; the authors made use of the previous work mentioned before, and also benefited from Fischer's *et al.* work in which they proposed a technique to identify references to bug databases in log messages, then used these references to infer links from VCS archives to BUGZILLA databases [Fischer et al., 2003a], [Fischer et al., 2003b]. In summary, the SZZ automatically links change logs and bug reports using some heuristics which search for specific keywords (such a " Fixed" or "Bug") and bug IDs (such a "#1234") in change logs [Bachmann and Bernstein, 2009], [Mockus and Votta, 2000], [Schröter et al., 2006], [Zimmermann et al., 2007]. This heuristic relies on " developers leaving hints or links regarding bug fixes in the change logs" [Wu et al., 2011].

However, the quality of the change logs are not ensured as they may incorrectly link with the *BFC* from the version archives with issue reports that are not bug reports from the issue tracking system. Bird *et al.* discovered that the absence of bug references in change logs affected the number of missing links leading to biased defect information, thereby affecting defect prediction performance [Bird et al., 2009a]. Researchers have further investigated this misclassification during the past years as the heuristics might yield biased data, Bird and Bachmann confirmed this problem and reported that 54% of bug reports are not linked to

BFCs [Bachmann et al., 2010], [Bird et al., 2009a]. Bettenburg *et al.* noticed that the issue reports often present incomplete and incorrect information [Bettenburg et al., 2008]. Antoniol *et al.* noticed that many of the issues in issues tracking systems did not describe bug reports [Antoniol et al., 2008]. Herzig *et al.* found that around one third of the bug reports that they manually analyzed were not describing a bug [Herzig et al., 2013]. Nguyen *et al.* showed that even in a near-ideal dataset, the biases exists [Nguyen et al., 2010].

On the other hand, some researchers have attempted to mitigate the limitations and shortcomings of linking bug reports with *BFCs* [Herzig et al., 2013],[Tan et al., 2015]. Thus, practice, tools and algorithms have been developed in order to mitigate this problem. For instance, GitHub supports linkage by automatically closing issues whether the commit message contains the *#numberOfIssue*, many Free/Open Source Software projects have adopted as a good practice to use keywords in their commit comments such as "# fix-bug -" when they are fixing a bug, as it has been reported for the Apache HTTP web server7 in [Bachmann et al., 2010], and for VTK8, and ITK9 in [McIntosh et al., 2016]. In addition, several authors have suggested the used of semantic heuristics [Schröter et al., 2006], [Čubranić and Murphy, 2003], [Zimmermann et al., 2007], while others have proposed solutions that rely on feature extraction from bugs and issue tracking system metadata. For Instance, Wu *et al.* developed ReLink to automatically link bugs reports and commits based on the similarity between the texts in both [Wu et al., 2011]. Bird *et al.* suggested manual inspections by developers in order to identify missing links. They proposed the tool LINKSTER that helps developer to locate possible links by providing query interfaces to the data [Bird et al., 2010]. Nguyen *et al.* proposed the Mlink tool to mitigate some of the problems found in ReLink, for instance some code changes in the commit are excluded and both issue reports and commit are used as plain texts [Nguyen et al., 2012]. Le *et al.* continued working on the shortcomings of the previous tools and develop RCLinker, which enriches commit logs. This approach extracts textual and metadata features from issues and commits [Le et al., 2015]. As a consequence of these efforts, the linkage problem has been addressed and its accuracy has drastically increased. For example, FRlink, an existing state-of-the-art bug linking approach, has improved the performance of missing link recovery compared to existing approaches, and it outperforms the previous one by 40.75% (in F-Measure) when achieving the highest recall [Sun et al., 2017].

2.2.2 Identifying the Bug-Introducing Commit (*BIC*)

Failure-inducing[2] changes were first addressed by Ness and Ngo in 1997 [Ness and Ngo, 1997]. They described how to identify a single failure-inducing change using simple linear and binary search. Their goal lies in isolating failure-inducing changes by applying chronological changes to a program until the fixed version presents the same wrong behavior as the next version of the program. Despite this, the technique is able to reduced the computational cost of testing each combination of changes introduced in the faulty version to locate the failure-inducing changes. However, this technique fails when instead of one change, a set of changes cause the failure. To deal with the issue, Zeller proposed the automated delta debugging technique, which can determine the minimal set of failure-inducing changes by gradually increasing granularity to identify the differences (that is, the deltas) between a passing and a failing subset [Zeller, 1999].

Purushothaman and Perry measured the likelihood for small changes, particularly one-line changes, to introduce errors. They refer to fix-inducing changes as *dependencies* which are changes to lines of code that were changed by an earlier commit. They assume that if the latter change was a *BFC*, the original change was erroneous. The study concludes that the probability of a one-line causing a bug to be less than 4% [Purushothaman and Perry, 2004]. Baker and Eick also used a similar concept when referring to fix-inducing changes,*fix-on-fix changes*. However, this concept requires both changes to be fixes. The paper describes a graphical technique for displaying large volumes of software where directories and subdirectories with high fix-on-fix rates were identified [Baker and Eick, 1994].

When a Bug-Fixing Commit exists: As previously mentioned, Śliwersky *et al.* proposed the SZZ algorithm for the first time, it locates fix-inducing changes in version archives [Śliwerski et al., 2005b]. Although the SZZ algorithm provides a technique to identify possible *BICs*, it has to deal with their incorrect identification. For this reason, many efforts have been made to suggest improvements to the SZZ algorithm. First, Kim *et al.* developed an algorithm that automatically identifies *BICs*; the algorithm is based on improvements of the SZZ algorithm that remove false positives and negatives by using the *annotation graph* technique instead of using VCS *annotate* to locate the lines changed in the bug-fixes. With

[2]Notice that some authors use failure-inducing as the concept for Bug-Introducing Commit

this modification, the SZZ may avoid some false positives by not considering non-semantic source code changes, (i.e. blank spaces, changes in the format or changes in the comments) and by ignoring outlier fixes. After a manual validation, the new version of SZZ can remove about 38%-51% of false positives and 14%-15% of false negatives [Kim et al., 2006c]. Secondly, Williams and Spacco proposed another enhancement of the SZZ algorithm. The authors suggested to use a mapping algorithm instead of the annotation graphs because they are more precise when facing larger blocks of modified code in bug fixes; this new approach uses weights to map the evolution of a unique source line and ignores comments and formatting changes in the source code with the help of `DiffJ`, a Java-specific tool. The authors also verified how often the *BICs* were the true source of a bug in a small sample size, where 33 of 43 lines mapped to a bug fix showed evidence of a bug being introduced [Williams and Spacco, 2008]. Then, Da Costa *et al.* realized that during the last ten years there was not much research conducted to evaluate the results of the SZZ, they proposed a framework that evaluates the results of the different SZZ implementations based on a set of criteria such as the earliest bug appearance, the future impact of changes, and the realism of bug introduction. Their findings suggest that the previous SZZ enhancements tend to inflate the number of incorrectly identified *BICs*, and by using this framework, the practitioners can evaluate the data generated by a given SZZ implementation and they might eliminate unlikely *BICs* from their outcome [da Costa et al., 2016]. Campos Neto *et al.* worked on improving the SZZ algorithm by disregarding refactoring changes as the *BICs* because they do not change the system behavior. The authors empirically investigated the impact of such refactoring in both changes, bug-fixing and bug-introducing. Their results indicate that their approach can reduce 20.8% of the incorrect *BICs* when compared to the first SZZ approach [Neto et al., 2018].

Other authors have created new approaches based on the same concept as SZZ algorithm, by attempting to mitigate the incorrect identification of *BICs*. Kawrykow and Robillard developed *DiffCat*, a tool-supported technique to detect and remove non-essential changes in the revision histories of projects. Their findings showed that up to 15.5% of system's method updates consisted entirely of non-essential modifications, [Kawrykow and Robillard, 2011]. Ferdian *et al.* looked at the root cause of the bugs by applying a combination of machine learning and code analysis techniques, then verifying their approach through comparing the

results with their manual analysis of 200 bug reports. This approach identifies the erroneous lines of code that cause a chain of erroneous behavior in the program leading to the failure; it has a precision of 76.42%, and a recall of 71.88% [Thung et al., 2013]. Servant and Jones introduced the fuzzy history graph; this technique helps to represent the code lineage as a continuous metric providing a balance of precision and recall. This technique performs better over the evolution of the code when compared to other models [Servant and Jones, 2017].

Other techniques are related to dependence techniques. These approaches also attempt to locate *BICs*; they address some of the shortcomings in the text-based approaches by examining the behavior of the changes by using a program dependence graph (PDG). Dependence-based techniques compare the PDG for the *BFC* to the PDG for the previous version. First, PDG only identifies removed dependencies, it only examines added dependencies when no dependencies were removed and returns only the most recent version involved. Sinha *et al.* introduced this technique in order to identify the *BIC* by analyzing the effects of *BFC* on program dependencies. This is a significant improvement over the text approach used by the SZZ algorithm, as this approach takes into account semantics of code changes, similar to previous work [Horwitz, 1990], [Binkley, 1992]. This make the approach to be more accurate and applicable to a wider class of *BFCs* (i.e., changes that involve addition of statements). Their results increased the precision and the recall of the fixes by 19% and 15% when compared to the text approach [Sinha et al., 2010]. After this technique was introduced, many additions and refinements were made. Davies *et al.* compared text-based and dependence-based techniques for identifying bug origins. The authors suggested detailed improvements to identify bug origins by combining both techniques [Davies et al., 2014].

When a Bug-Fixing Commit do not exist: Contrary to the previous authors, some authors have studied the origin of bugs without identifying *BFCs*; and they have developed different methods such as Delta Debugging [Zeller, 2002], [Zeller and Hildebrandt, 2002], [Cleve and Zeller, 2005], [Misherghi and Su, 2006]; Spectrum Based Fault Location [Reps et al., 1997], [Janssen et al., 2009], [Harrold et al., 2000], [Tiwari et al., 2011], [Abreu et al., 2007], [Jones et al., 2002]; or Nearest Neighbor [Renieres and Reiss, 2003], [Jones et al., 2002], [Abreu et al., 2007]. A brief description of each method is given as follows.

Delta Debugging Zeller and Hildebrandt described the Delta Debugging algorithm for the

first time in [Zeller and Hildebrandt, 2002]. This algorithm compares the program states of a failing and passing run, while using binary search with iterative runs. The iterations stop when the smallest state change that caused the original failure is identified. In other words, this technique defines a method to automatize the process of making different hypotheses about how changes affect output to locate failure causes. Gupta *et al.* used Delta Debugging combined with dynamic slicing to identify the set of statements that is likely to contain a faulty code [Gupta et al., 2005]. Cleve and Zeller [Cleve and Zeller, 2005] presented the Cause Transitions technique and compared it to the Nearest-Neighbour technique. Their results suggest that, on the same set of subjects, Cause Transitions technique performs better than Nearest Neighbour.

Spectrum Based Fault Location: A method used to locate faults from the identification of the statements involved in failures was first introduced in [Reps et al., 1997]. This technique usually takes as inputs two sets of spectra, one for successful executions and the other for failed executions. It reports candidate locations where causes of program failures occur (e.g., lines, blocks, methods, etc.), which may be presented to debuggers. There are many spectra such as node spectra, edge-pair spectra, edge spectra and block spectra. Jones *et al.* developed the Tarantula system which provides a way to rank statements in terms of their likelihood of being faulty. Furthermore, it has a graphical user interface that specifies a color for each statement in the program depending on the suspiciousness of being buggy [Jones et al., 2002]. Abreu *et al.* investigate the diagnostic accuracy of the spectrum-based fault localization as a function of several parameters using the Siemens Set benchmark. Their results indicate that the superior performance of a particular coefficient is largely independent on test case design [Abreu et al., 2007].

Nearest Neighbour. Renieres and Reiss used Nearest Neighbour queries to locate the fault [Renieres and Reiss, 2003]. This technique contrasts a failed test with a successful test which in terms of distance, is more similar to the failed test. It uses these two test cases to remove the set of statements executed by the passed test case from ones executed by the failed test case. A bug is then located whether it is in the difference set between the failed run, and its most similar successful run. In case the bug is not contained in the difference set, this technique continues constructing a program dependence graph, while adding and checking adjacent un-checked nodes in the graph until the bug is located. This method is

easily applicable since it only requires a classification of the runs as either incorrect or faulty from the users.

Tools for locating bugs: A lot of effort have been made to develop practical tools that assist in locating and detecting the bugs. Without trying to be exhaustive, a brief description of some tools that have been used to find bugs in the source code is given.

FindBugs[3] is an automatic detector for bugs with the same pattern in Java source code. The user experience of this tool showed that it was helpful and most of the warnings fixed in a specific organization were "hashcode/equals problems, serialization issues, unchecked method return values, unused fields and constants, mutable static data, and null pointer dereferences" [Hovemeyer and Pugh, 2004]. HATARI is a plugin for Eclipse that determines how risky is a change depending on the area of source code [Śliwerski et al., 2005a]. PMD[4] tool is a static code analyzer that checks the source code of a project in order to find possible bugs, dead code, suboptimal code or overcomplicated expressions. It checks for patterns in the abstract syntax tree of parsed sources files [Copeland, 2005]. Jlint[5] is a tool that checks for bugs, inconsistencies and synchronization problems in Java code [Artho, 2001]. FixCache has a similar purpose, files and methods are saved and maintained in the cache; when a bug is fixed, these elements are updated and the cause of the bug is identified. The cache can be used to predict how likely a change in an area might cause another bug [Kim et al., 2007]. BugMem identifies project-specific bugs and suggest corresponding fixes, it uses a learning process of bug patterns of project-specific bugs [Kim et al., 2006a]. OpenJML[6] is a compile-time checker tool that warns against potential runtime errors and inconsistencies between the design decision recorded and the actual code. It is the successor of ESC/Java. The feedback of users using this tool supports that it can detect real software defects [Flanagan et al., 2013]. BugLocalizer[7] is implemented as a Bugzilla extension, and it uses information retrieval (IR) based bug localization that computes similarities of the bug report with source code files with the aim of locating the buggy files [Thung et al., 2014]. Commit Guru[8] is a tool that

[3] http://findbugs.sourceforge.net

[4] http://pmd.sourceforge.net/snapshot/

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

[6] http://jmlspecs.sourceforge.net/

[7] https://github.com/SoftwareEngineeringToolDemos/FSE-2014-BugLocalizer

[8] http://commit.guru

identifies and predicts suspicious buggy commits. The tool also provides downloadable analytics to users on the likelihood of a recent commit to introduce a bug in the future, the percentage of possible commits that may have introduced buggy code in their projects, among others [Rosen et al., 2015].

2.2.3 Software Testing to Locate Faults

Software testing is conducted to provide information about the quality of the software or service under test [Kaner, 2006]. The test techniques include the process of executing an application with the purpose of finding software bugs before the software product is released.

Much work on software testing seeks to ensure the minimum human intervention by automatizing as much as possible the process, to make testing faster cheaper and more reliable. Without trying to be exhaustive, this work can be understood in the following categories:

Automatic software repair: It is challenging because of its difficulty. It is focused on two main fields, behavioral repair, and state repair. Behavioral repair address the issue of test-suite based repair that states " given a program and its test suite with at least one failing test case, create a patch that makes the whole test suite passing" [Monperrus, 2014] and which has been explored by the Genprog. Genprog is a seminal and archetypal test-suite based repair system developed at the University of Virginia [Weimer et al., 2009], [Forrest et al., 2009] whose evaluation in a later study claims that 55 out of 105 bugs can be fixed by Genprog [Le Goues et al., 2012]. Currently, people are still working on improving the core repair operators. On the other hand, the large research field of state repair address the issue of *recovery* that focuses on "a system state that contains one or more errors and (possibly) faults into a state without detected errors [Laprie, 1985].

Emulation of software faults: It is used to evaluate fault tolerance procedures, and to assess the impact that a bug will have in the system. Durães and Madeira created a new fault injection technique (G-SWFIT) after observing that a large percentage of faults can be characterized with high accuracy, and that allows to use a small set of emulation operators instead emulate the software faults with a big set, allowing accurate emulation of software faults through a small set of emulation operators [Duraes and Madeira, 2006].

2.3 Studies on Identifying the Origin of a Bug

The foundational role of locating a bug origin in Software Engineering resides in its transversality. Once the origin of a bug is located, many different areas in Software Engineering can benefit from the knowledge. They can deliver different studies with different outcomes where practitioners can better learn practices to continue improving the current state of the art about bug seeding and software engineering. Without attempting to be exhaustive in the description, we offer several examples where authors have analyzed the origins of bugs with different general purposes. Five different categories were selected based on the important role of correctly identifying the *BIC*: bug prediction, bug localization, bug classification, bug fix and software evolution.

Bug prediction: Bug prediction is aimed at supporting developers to identify whether a change will be buggy or not. This area studies bug seeding and bug fixing activity as a potential source of prediction for further issues. For instance, Feng *et al.* collected the defect data and attempted to build a universal defect prediction model for a large set of projects from various contexts [Zhang et al., 2014]. Jiang *et al.* proposed a novel technique that produces a personalized model for each developer; this model is used to predict bugs on future data [Jiang et al., 2013]. Hata *et al.* developed a fine-grained version control system for Java in order to conduct fine-grained prediction [Hata et al., 2012]. Kim *et al.* analyzed the version history of seven projects to predict the most fault prone entities and files; they identified the *BICs* at the file and entity level [Kim et al., 2007]. Zimmermann *et al.* predicted bugs in large software systems such as Eclipse [Zimmermann et al., 2007]. Nagapan *et al.* associated metrics with post-release defects to build a regression model that predicts the likelihood of post-release defects for new entities [Nagappan et al., 2006]. Yang *et al.* studied what kind of *BICs* are likely to become a great threat after being marked as *BFCs* [Yang et al., 2014]. Rosen *et al.* developed a prediction tool base that identifies and predicts risky software commits [Rosen et al., 2015]. Kamei *et al.* used just-in-time *(JIT)* quality assurance concept to build a model that predicts whether a change is likely to be a *BIC* [Kamei et al., 2013]. Fukushima *et al.* empirically evaluated the performance of defect prediction models based on Just-In-Time cross-project [Fukushima et al., 2014].

Bug Classification: Bug classification is aimed at supporting developers in classifying whether a change is buggy or not. For example, Pan *et al.* described a program slicing metrics to classify changes as buggy or bug-free. They use SZZ to mark files that have *BICs* [Pan et al., 2006]. Kim *et al.* showed how to classify file changes as buggy or clean using change information features and source code terms [Kim et al., 2008]. Thomas *et al.* introduced a framework for combining multiple classifier configurations that improves the performance of the best classifier [Thomas et al., 2013]. Ferzund *et al.* presented a technique to classify software changes as buggy or buggy-free based on hunk metrics [Ferzund et al., 2009]. Kim and Ernst proposed a history-based warning prioritization algorithm that helps to improve the prioritization of bug-finding tools [Kim and Ernst, 2007]. Nguyen and Fabio Massacci conducted an empirical study to validate the reliability of the NVD vulnerable version data [Nguyen and Massacci, 2013].

Bug Localization: Bug localization is aimed at supporting developers in identifying where a bug resides. Asaduzzaman *et al.* applied the SZZ algorithm on Android to identify the *BICs*, they then used this information to look for problems during the maintenance upkeep of the project [Asaduzzaman et al., 2012]. Schröter *et al.* built a data set that contains the mapping between the bug reports and their *BICs* in the Eclipse project [Schröter et al., 2006]. Kim *et al.* developed a tool to find bugs using the bug fix memories, which also focused on the knowledge of changes that fix bugs. The tool uses statistical-analysis to learn project-specific bug patterns by analyzing the history of the project and then suggest corrections [Kim et al., 2006a]. Wen *et al.* proposed the use of LOCUS, an IR-based bug localization tool based on the analysis of software changes and contextual clues for bug-fixing. The performance of LOCUS at source file level have significantly improved, on average around 20.1% and 20.5%, as demonstrated in the results of MAP and MRR techniques [Wen et al., 2016]. Youm *et al.* developed BLIA, a statically integrated analysis tool of IR-based bug localization that uses information from bug reports, source files and source code changes histories. The authors claimed that this tool achieves better results than BugLocator, BLUiR, BRTracer and AmaLgam [Youm et al., 2015].

Bug Fix: Bug fix is aimed at supporting developers in improving the bug fixing process. Researchers have studied who should fix a certain bug report [Kagdi et al., 2008], [Anvik et al., 2006] based on previous changes of the same file. Another approach used by Guo *et al.* predicts whether a bug report will be fixed. This approach is based on the study of different characteristics and factors that affect the fix of bug reports in Windows Vista and Windows 7. This study found that bugs were more likely fixed when they were reported by people with higher reputation, or when they were handled by people on the same team [Guo et al., 2010]. Cincarini and Sillitti extended the previous study in an open source environment and confirmed the results found by Guo *et al.* [Ciancarini and Sillitti, 2016]. Other authors have dealt with assigning bug reports to individual developers, Baysal *et al.* developed an approach that uses developer's expertise, current workload and preferences to assign the appropriate developer to fix a bug [Baysal et al., 2009]. Another common practice during the bug fixing process is to compute the time required to fix a bug after it was introduced into the source code. Kim and Whitehead computed the time to fix a bug in files of ArgoUML and PostgreSQL project. Their results indicated that the median was about 200 days [Kim and Whitehead Jr, 2006]. Zang *et al.* developed a Markov-based method for predicting how many bugs will be fixed in the future and the time required to fix them [Zhang et al., 2013]. Some authors have conducted empirical studies to understand the usefulness of social platforms such as Twitter in the bug fixing process [El Mezouar et al., 2017]. Finally, other authors, have studied the bug fixing patterns by using the SCM systems, they focused on the semantics of the source code [Pan et al., 2009].

Software Evolution: Software evolution is aimed at supporting developers in understanding how a software evolves and which characteristics (authorship, time of commit, developers' interaction...) or patterns are implicated in the bug proneness. Kim and Whitehead computed the time to fix a bug after it was introduced into the source code in ArgoUML and PostgreSQL [Kim and Whitehead Jr, 2006]. Kim *et al.* also studied the properties and evolution patterns of signature changes in seven software systems written in C, using SZZ to identify the *BICs* [Kim et al., 2006b]. Eyolfson studied whether the time of the day and developer experience affects the probability of a commit to introduce a bug [Kamei et al., 2010]. Izquierdo *et al.* researched whether developers fixed their own bugs [Izquierdo et al., 2011]; they also

studied the relationships between experience and the bug introduction ratio using the Mozilla community as case of study [Izquierdo-Cortázar et al., 2012]. Rahman and Devanbu attempted to understand some factors that have a big impact on software quality such as ownership, experience, organizational structure, and geographic distribution [Rahman and Devanbu, 2011]. Posnett *et al.* studied the effect of artifact ownership and developer focus on software quality. They discovered that more focused developers introduce fewer defects than less focused developers[Posnett et al., 2013]. Bavota *et al.,* carried out an empirical study in three Java systems to investigate the extent of refactoring activities in introducing a bug [Bavota et al., 2012].

Chapter 3

Context of the problem

This chapter attempts to explain in detail the whole context of the current problem with identifying the precise moment when a bug was introduced into the source code of a software system. It then describes the many reasons why the current state-of-the-art approaches are not successful in correctly identifying Bug-Introducing Commits (*BICs*). Finally, this chapter gives motivating examples to demonstrate the necessity for practitioners and researchers to search for other more accurate methods to identify the moment when a bug is introduced into the source code.

In the previous chapters of this thesis we have described the important role of correctly identifying *BIC*. There are many reasons for the explanation of this special interest. From the economic point of view, software bugs are costly to fix [Lerner, 1994] and they are highly time-consuming [LaToza et al., 2006]. In 2009, the US National Institute of Standards and Technology (NIST) estimated that the US economy earmarks $59.5 billion annually to fix software defects and to reinstall systems that have been infected. Zhivich and Cunningham reported that software developers use approximately 80% of the total 59.5 billion to identify and correct defects [Zhivich and Cunningham, 2009]. From the research point of view, the knowledge of where the bug has been first introduced has important implications in software engineering disciplines as we have explained in Chapter 1. However, specific characteristics of software evolution and maintenance complicate the correct identification process of the *BICs*. For example:

- The complexity of software products: software systems and their architecture are continuously evolving and becoming more complicated over time; this may lead to prob-

lems that creep into the system and manifesting as bugs [Le, 2016]. It may also lead to the manifestation of failures in unchanged parts [German et al., 2009].

- The dependency of the source code with external artifacts: the component-oriented development model leads to the development of software products that are an assemblage of small components from many different sources. In this scenario, it will be tedious to estimate the behavior of the whole system when one of its components behaves erroneously [Duraes and Madeira, 2006];

- The accuracy: the current approaches rely heavily on manual analysis to evaluate the results with the restriction that only the experts can judge if the changes identified using such methods are the real causes of the bugs. Unfortunately, this analysis will be impractical in most of the cases and there is no existing framework to correctly evaluate the results obtained after using different approaches to locate the bug-introduction moment [da Costa et al., 2016].

Thus, without a clear methodology to know exactly what line or lines created a bug, many studies in the area of software maintenance and evolution start with the implicit assumption that the line (or lines) that is being replaced in a bug fix is likely the cause for introducing the bug. This assumption can be frequently found in the research literature, for instance in:

- "We assume that the last change before the fixing change was the change that introduced the defect" [Cao, 2015].

- "A fix-inducing is a change that later gets undone by a fix" [Shippey, 2015].

- "The SZZ observes each hunk in the bug-fix and assumes that the deleted or modified lines are the cause" [Shivaji, 2013].

- "The defect was caused in one of the artifacts that was later edited to correct the defect" [VanHilst et al., 2011].

- "The lines that have been removed of modified in the Bug-Fixing Commit are the ones where the bug was located" [Izquierdo et al., 2011]

- "We assume that the person who injected defects into a file is the person who changed it" [Ando et al., 2015].

- "We assume that faults are reported just after they are injected in the software" [Yamada and Mizuno, 2014].
- "To trace backwards through the version history to identify for each of these lines the last commit that has changed the line" [Prechelt and Pepper, 2014].
- "[The] [b]lame feature of Git is used to identify the file revision where the last change to that line occurred" [Bavota and Russo, 2015].
- "We determine the defect-inducing change as the change that is closest and before" [Wehaibi et al., 2016].
- "A line that is deleted or changed by a bug-fixing change is a faulty line" [Tan et al., 2015].
- "We mark those hunks as bug introducing in which we find the source code involved" [Ferzund et al., 2009].

One of the key problems with the approaches when identifying the *BIC* is the assumption that "the modified line in a Bug-Fixing Commit (*BFC*) is likely the one that introduced the bug". Although this assumption may appear reasonable at first glance given its frequent presence in the research literature, there is not enough empirical evidence supporting it. Furthermore, recent studies have demonstrated the many limitations of this assumption [da Costa et al., 2016] when flagging potential changes as *BICs*. In addition, the Chapter 4 of this dissertation details that even when the researchers were aware of using this assumption and knowing full well of its limitations, they still use it in their studies. For example, some research studies have commented on the threat when a fixing commit only adds new lines, or when the line has been modified several times since its introduction, or when some lines that fix the bug are not related with it:

- "It is possible that previously in the history of the inspected line a large addition of lines has introduced this error, thus confusing the history of the line." [Williams and Spacco, 2008].
- "There are some bugs introduced in one place, but fixed in another place" [Yuan et al., 2013].
- "Fix locations may inflate the warning false positive rate. Additionally, adding new code may fix an existing warning" [Kim and Ernst, 2007].

- "The SZZ algorithm used to identify Bug-Introducing Commits has limitations: it cannot find Bug-Introducing Commit for bug fixes that only involve addition of source code. It also cannot identify Bug-Introducing Commits caused by a change made to a file different from the one being analyzed." [Shivaji et al., 2013].

- "It is extremely hard to automatically understand the root of vulnerabilities." [Nguyen and Massacci, 2013].

- "In a fix of a crash-related bug, not all of the changes are aimed to address defects. Some lines may be added because of a refactoring or an addition of a new feature. These changes are hard to identify with an automatic approach." [Jongyindee et al., 2011].

- "Additionally, it is not necessary that a bug may have been introduced in the most recent CVS transaction that changed the relevant lines in the file." [Abreu and Premraj, 2009].

However, researchers cannot be sure whether a *BIC* is the actual moment when a bug has been introduced in the system, as the term bug is undefined, thereby making it impossible to understand what is meant by the introduction of a bug. When the approaches place blame on a line that is suspicious to contain the bug, this line should not be isolated from the whole context of when it was introduced the first time. This is because based on this context, researchers can understand whether the line that introduced the bug occurred at that particular moment, or on the contrary, the line was correct in that moment but defected later due to changes in other parts of the code causing it to manifest in that specific line. For example, when a source code is using a third-party code, and it changes something in the API without a previous notification, it could be possible that at some point the lines of the source code manifests a bug. This however does not mean that the lines of the source are responsible for inserting the bug, as they could be perfectly clean when they were first written into the source code. As such, in this example, the bug has not been introduced in the previous lines blamed for some approaches such as the SZZ, the bug has been caused by the evolution of a third-party code.

The main problem lies in the current literature and how the act of introducing a bug is defined as well as how the moment of its introduction is defined. It is possible that both moments are the same, but it is also possible that they are different. The latter case has

not been addressed in the actual state-of-the-art literature. Thus, the current heuristics and approaches need to be extended; there is the necessity to build a model that contemplates all the different scenarios in order to re-define the theory of bug introduction. Fortunately, in modern software development many traces can be retrieved on how code changes, and how bugs are fixed. Thanks to that, a lot of information is at disposal where when analyzed, provides the means to understand the reasons why a change was required to fix a bug. Thus, before building a model, we can use information from the source code management system, the issue tracking system and the code review system to first understand the malfunction that instigated change, then identify what problems in the source code were causing it, and finally how it was fixed.

3.1 The Effectiveness of Current Approaches

Nowadays, approaches based on backtracking the lines of a *BFC* are not using any meaningful model that researchers or practitioners can employ to validate the algorithms. The lack of definition on what should be validated makes it difficult for researchers to describe what a false positive, true positive, false negative and true negative is. These algorithms attempt to find which commit introduced the bug, but there is no differentiation between which commits introduced the bug and which commit did not introduced the bug. The current algorithms also do not distinguish between the moment of introduction and the moment of manifestation. For these reasons, researchers cannot be sure about what it means to introduce a bug. To measure the accuracy of their approaches, many of the researchers use the concepts of false/true positives and false/true negatives without taking into account the real meaning of these concepts. Nowadays, the approaches compute the precision and the recall using the following definitions:

- ***True positive***: Given a commit identified by applying one of the approaches in a *BFC*, a true positive is when this commit last modified the buggy line(s) changed in the *BFC*. This definition of true positive means that this commit is likely to be the cause of the failure, but researchers cannot be sure whether or not it introduced the bug into the system when the suspicious buggy line was first instroduced.

- ***False positive***: Given a commit identified by applying one of the approaches in a *BFC*, a false positive is when this commit last modified the lines changed in the *BFC*, but it is likely that the change did not insert a bug, although the algorithm flags it as a Bug-Introducing Commit. For example, whether the commit introduced blank lines that changed the format of a function by moving a bracket or adding a tabulation, added or modified comment lines, or when it renamed a variable, this definition of false positive implies that the commit is not the cause of a bug.

- ***False negative***: Researchers have different perceptions of what a false negative is; Da Costa *et al.* defined it as "a Bug-Introducing Commit that is not flagged as such by SZZ" [da Costa et al., 2016]. Kim *et al.* assumed that their improved version of SZZ is more accurate than the original, and computed the false negatives as $(\frac{|K-S|}{K})$, where K is the set of *BICs* detected by their algorithm, and S as the set of *BICs* detected by SZZ [Kim et al., 2006c]. Davies *et al.* defined it as "commits that introduced bugs but which are not identified by the approaches", and surprisingly, they did not find any false negative in their study [Davies et al., 2014].

- ***True negative***: Any commit that was not identified by the approaches and is not responsible for introducing the bug.

However, as mentioned earlier, these definitions of true and false positive are not completely correct, because they are not based on understanding whether the identified lines introduced the error in the project at this location and time stamp. Thus, it is imperative to establish proper definitions for the concepts regarding false positive, false negative, true positive and true negative, as can be seen below:

- ***True positive***: Given a commit identified by applying one of the approaches in a *BFC*, a true positive is when this commit, that belongs to the repository, modified the source code of a project and introduced the error which caused the later *BFC*.

- ***False positive***: Given a commit identified by applying one of the approaches in a *BFC*, a false positive is when this commit modified the source code of a project but did not insert the error at this time. This means that in this moment, the lines were clean and as

a consequence, this commit did not caused the *BFC*. Some examples of false positives are described through the next paragraphs of this section.

- ***False negative***: A false negative is when a commit introduced the bug in the source code of a system and caused the *BFC* but the current algorithms cannot identify it by means of their heuristics.

- ***True negative*** A true negative is when a commit that did not insert the bug in the source code of a system and did not caused the later *BFC* is not identified by the current algorithms.

Nevertheless, the approaches built on backtracking the lines of a *BFC*, search for the last commit that touched the line(s) that was modified to fix a bug, the *previous commit(s)*. After applying these approaches to a *BFC*, there is a set of previous commits that can contain one or more different previous commits. Thus, researchers have to decide which previous commit from the previous commit set is causing the bug. However, it can be possible that the *BIC* is one or none of the previous commit, and it is possible that none of them caused the bug because a bug could have been either introduced in new lines in other parts of the code, or a bug could have been introduced by an external artifact or because of a change in the environment of the project.

The Systematic Literature Review described in Chapter 4 quantifies the limitations of the current approaches to identify the *BIC*. These reasons are the following:

1. **Identification of more than one previous commit:** A *BFC* may have more than one line edited (deleted, modified or added), and in cases where a developer changed more than one line to fix a bug, it is possible that the previous commit of these lines were different. Thus, when the approaches to find the suspicious *BIC* are applied, there may be different previous commits to blame for the cause of the bug. The main problem in this case is that the current state-of-the-art approaches do not provide any guidelines on how practitioners or researchers should behave in these situations. Moreover, there has been no clear explanation about the heuristics that have been used in the cases where the scenarios come up. This might be a big source of false positives.

2. **When only new lines are used to fix the bug:** There are some bugs that are fixed

by simply adding new lines to the source code. This may occur when some ancestor commit forgets to add lines to the source code. As a result, when the system fails, the developers need to fix the bug by adding new lines to the source code. For example, a commit may miss to add a null-pointer dereference. The fix adds a null-check line, while the lines of the commit that have missed to add the code are not changed. However, the current approaches remove these cases from their analysis because the new lines cannot be tracked back. As a consequence, these approaches are not able to identify the *BIC* which causes the presence of false negatives in these scenarios.

3. **Changes in the environment or configuration:** There are some bugs that manifest themselves before a change in the environment or during the configuration. These kinds of defects correspond to the bugs that lie in third-party libraries, underlying operating systems, or non-code parts (e.g., configuration files). Thus, when one of the parts changes without any previous notification to the developers, the system experiences the failure, and the developers are required to change the lines that are affected by the ecosystem or the configuration of the project to fix the project. The issue here is that when the current approaches are applied, the previous commits are identified as the *BICs* when in fact, they did not introduce the bug. This is because when the lines are introduced the first time, they were initially correct for the ecosystem at that point in the time. In these cases, the approaches are introducing false positives.

4. **Multiple modifications of a line:** The evolution of the source code of a project affects the identification of *BICs* due to the necessity of the implementation of new requirements. When the source code of a new requirement is committed, the code might alter the identity of the previous commit in a line. For instance, when the commit *123aa* is introduced into the function *func*, and later the commit *456bb* inserts a new functionality into the project that modifies the function *func* by adding a new argument to it. In these cases, the last commit that touched the line of the function *func* is the commit *456bb*, and it may occur multiple times in the same line. The problem lies in whether the commit *123aa* was responsible for inserting the bug. In this scenario, the current approaches cannot identify it, pushing the blame to the false positive commits as the *BICs*, because they were the last commits that modified the line that is failing.

5. **Weak semantic level:** A key factor in the correct identification of *BICs* is the comprehension of the changes made in each commit. The modifications of lines are addressed for different purposes; some modifications are made to optimize the code while the same behavior remains in the code, other modifications are made to rename variables or functions or to remove dead code, while other modifications simply copy and paste lines from other commits. All of these modifications are semantic, which causes the approaches to identify false positive *BICs*. On the one hand, the false positive may occur because the real change that introduced the buggy behavior to the source code was before the modifications, and that the semantic changes are hinting towards the real cause. On the other hand, the false positive may occur because the approaches might identify numerous, suspicious commits, when in fact they are simply semantic changes and should instead be removed from the analysis.

6. **A Bug-Fixing Commit that fixed more than one bug:** Although it is not common for a *BFC* to fix more than one bug, sometimes due to the close relationship between the bugs or the dependency between them, researchers may find that a *BFC* closed more than one bug report. This causes the approaches used to identify the Bug-Introducing Commit to identify false positive commits even though the two bugs addressed in the same Bug-Fixing Commit have been introduced in different commits.

7. **Compatibility bugs:** These kinds of cases correspond to the bugs that make a system fail or pass depending on the particular CPU architecture, operating system, or Web browser used. For example, user *A* never experiences a bug when using the project under macOS, but user *B* who is using Windows has experienced the bug due to the failure of the project using this OS. Thus, with these kinds of bugs it will not be fair to lay the blame on a previous change or an ancestor change as the *BIC*, because the developers' intentions and the circumstances of the moment cannot be known for certain whether it will later fail when they submit the source code. In these cases, the current approaches also identify false positives *BICs*.

8. **Dormant bugs:** Tse-Hsun *et al.* have investigated this type of bugs in depth. The characteristics of the bugs is that when they are introduced in an earlier version of the system, they are not reported until much later. Thus, when the current approaches

are applied to the bugs to find the *BIC*, the result are false positive. Sometimes, lines fixed in the *BFC* are not responsible for introducing the bug. This could be due to the natural evolution of the source code where other developers may have modified some parts of the dormant buggy code. The approaches identified as a result identifies the modifications as the *BIC*.

It is clear that the effectiveness of the current approaches is not up to standard for practitioners and researchers. This problem lies mainly on how these approaches work. The approaches identify the chronologically latest modification of a fixed line as the bug-introducing version, but it is now apparent in many scenarios, such as the explained before, that these approaches fail because of this reason. This thesis explores a new concept of the first failing commit, based on the hypothetical idea that there is a perfect test and it can be run forever in the past. Let's take V_n as a version and t as the perfect test with a coverage of 100%. A fault can be revealed given a fixed version V_{n+1}. Assuming that t is an oracle that tests how the behavior of the fixed lines should be in V_n and previous versions of V_n, it is possible to identify the buggy version regardless of whether the test fails. For instance, as the test knows what the bug is and what should be the correct behavior of the source code at that point, if the test passes, it means that this version did not introduce the bug, and it continues to backtrack the version history of the source code. The approach starts with the version V_{n-1}, where it executes the test in each version to identify the version that fails. This version will be the *FFM* and also the *BIC* depending on other factors. More details are found in Chapter 5.

3.2 Motivating Examples

In this section we present real motivating examples found in the projects that were analyzed. These examples clearly describe the necessity to distinguish the difference between the *BIC* and the *FFM*. There are different kind of bugs, some of them were directly introduced but others have manifested themselves in the system without the necessity of being introduced. Thus, the nature of bugs provide a guide to find other approaches to identify the first change that manifested the malfunction, enabling the understanding of whether the change also introduced the bug. The next examples support this urgent necessity by explaining what caused the bug when the bug manifests itself and how other approaches fail to identify the *BIC*. It is

important to keep in mind that the heuristics of the most famous approaches to identify *BIC* is through looking at the previous changes that touched the fixed lines in the *BFC*. In case there are more than one change, these approaches use heuristics to determine which lines contain the fix that caused the bug. It may be one, or a combination of many. It could also be none of the previous changes or none of the ancestral changes. In fact, the bug cannot be caused by any previous commit or any ancestor commit because the bug was caused by a change in an external artifact that the project uses.

The first example is the bug #3820[1] from ElasticSearch. Figure 3.1 (a) is the bug report description. Figure 3.1 (b) is the description of its *BFC 565c21273*[2]. The bug report describes a bug when setting permissions for subdirectories in Debian. According to the description, the bug is due to a wrong configuration in a new scenario where subdirectories exist. For a period of time in ElasticSearch, there was no possibility of creating subdirectories in */etc/elastcisearch*. As a result the files under */etc/elastcisearch* can be set with permission 0644, but at some point in the history of the project, this changed and it was possible to have subdirectories under */etc/elastcisearch*. For this new configuration, it is not reasonable to have the setting with permission 0644, in which case the bug manifested itself in the system. When looking at Figure 3.2, it is clear that to fix the bug, the developer modified line *37* and also added a new line. However, the modification of line *37* does not mean that the line was buggy at the moment of its insertion, but rather due to other factors (i.e., a new configuration); this line merely manifested the failure. When the current approaches are applied to the bug in order to find the *BIC*, all of them will fail because they will blame the previous commit *bccf0b1* as the *BIC* when in fact it did not introduce a bug. Hence, in this case it is impossible to pinpoint a concrete change as the *BIC*, because the bug depends on the moment when the developers decided to allow subdirectories under */etc/elastcisearch*. For this reason it is only possible to identify the *FFM*.

The second example is bug #3551[3] which is also from ElasticSearch. Figure 3.3 (a) is the bug report description and (b) is the message description of its *BFC*. Below, Figure 3.4 shows the code in the *BFC 8668479b9*[4]. The bug report describes a bug when downloading

[1] https://github.com/elastic/elasticsearch/issues/3820

[2] https://github.com/elastic/elasticsearch/commit/565c21273

[3] https://github.com/elastic/elasticsearch/issues/3551

[4] https://github.com/elastic/elasticsearch/commit/8668479b9

Figure 3.1: Bug caused after changing the version of the software.

Figure 3.2: diff of the Bug-Fixing Commit

a site plugin from GitHub. In this case, the dependency of the source code of ElasticSearch on a third-party as GitHub is what caused the bug. At some point the API of GitHub changed and, as a consequence, the plugin to download *URL* from the master zip file does not work, as a result the *BFC* will have to pass the path of GitHub in order to fix the bug. In Figure 3.4, it can be observed that to fix the bug, the developer modified two lines *182 and 196*. These modifications however do not mean that the lines were inserting a bug at the moment of the change, but these lines manifested the failure due to other factors (i.e., a change in a third-party). Thus, as in the above example, none of the previous commits were inserting the bug, so it can be only identified after the change when the bug starts to manifest, meaning it is only possible to identify the *FFM*.

Finally, the last example is the bug #1305897[5] from Nova. Figure 3.5 shows the descrip-

[5]https://bugs.launchpad.net/nova/+bug/1305897

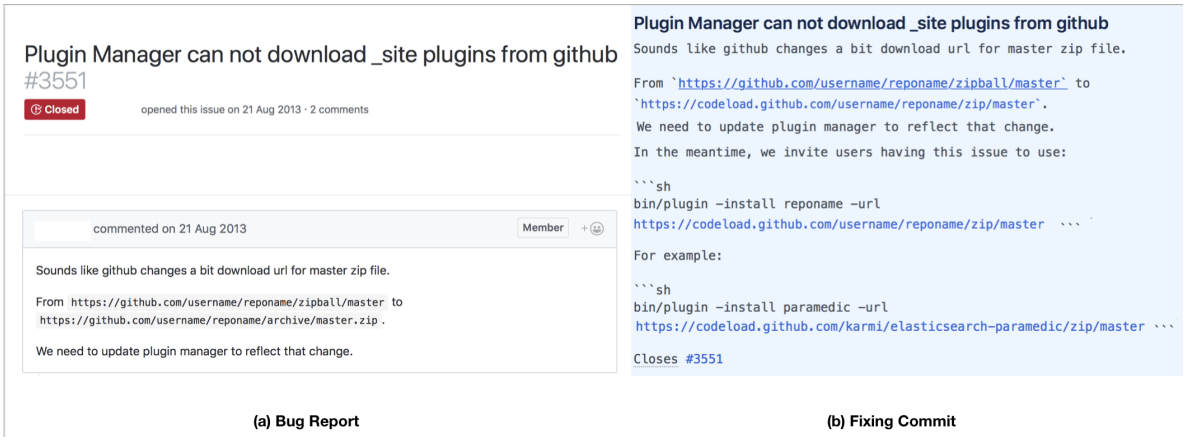
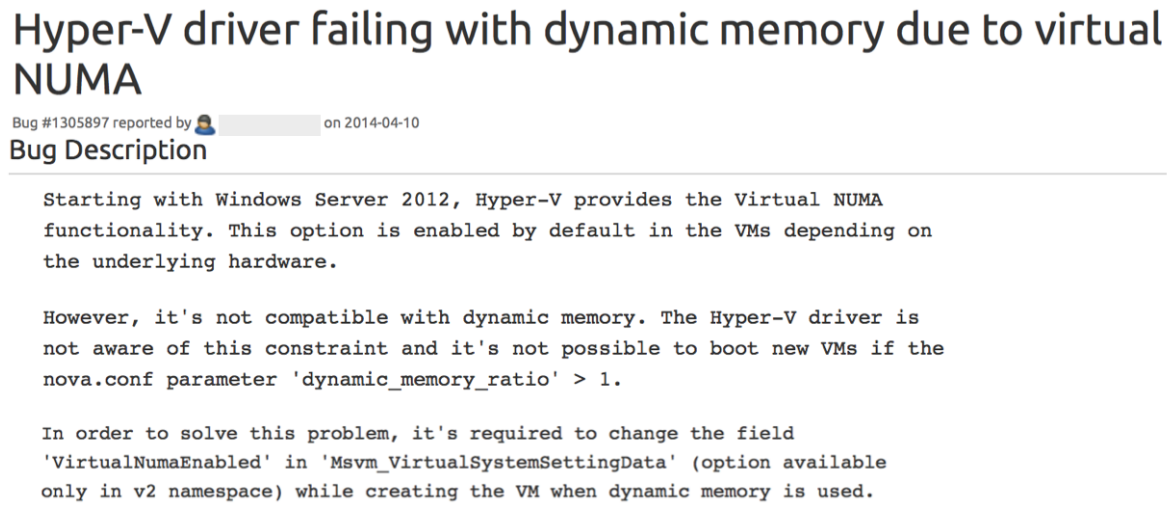


Figure 3.3: Bug caused by an external artifact.

Figure 3.4: diff of the Bug-Fixing Commit.

tion of the bug report in Launchpad. According to this report, the bug was caused by an incompatibility between the software and the hardware used. The bug appeared when an option was enabled by default in the VMs; this option depends on the underlying hardware. Thus, when a user has Windows Server 2012, this option is enabled and it causes the Hyper-V driver to not be aware of the constraint, therefore it becomes impossible to boot new VMs. Figure 3.6 shows the *BFC* that modified line *92* and introduced two new lines into the source code. The modification added a new argument into a function. This argument is used in the added lines to check the constrain that caused the bug. Thus, the change that introduced the modified line in the *BFC* cannot be labeled as the *BIC*. This is because the bug manifests depending on the environment. In this example, the current approaches also identify a false positive *BIC*, because it was correct in the moment of their insertion, while assuming that the

developers were not aware of this issue and their intentions were not to make the software compatible in all the possible scenarios.



Hyper-V driver failing with dynamic memory due to virtual NUMA

Bug #1305897 reported by on 2014-04-10

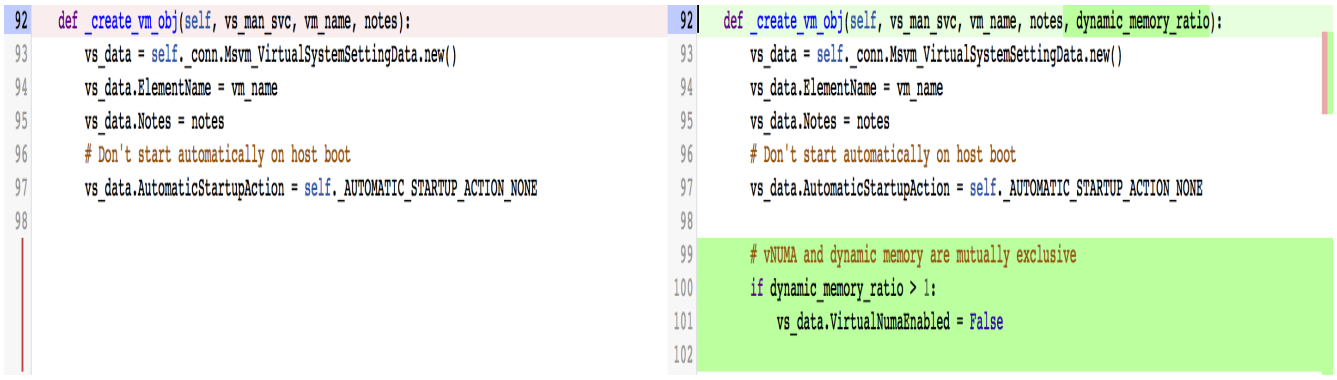
Bug Description

Starting with Windows Server 2012, Hyper-V provides the Virtual NUMA functionality. This option is enabled by default in the VMs depending on the underlying hardware.

However, it's not compatible with dynamic memory. The Hyper-V driver is not aware of this constraint and it's not possible to boot new VMs if the nova.conf parameter 'dynamic_memory_ratio' > 1.

In order to solve this problem, it's required to change the field 'VirtualNumaEnabled' in 'Msvm_VirtualSystemSettingData' (option available only in v2 namespace) while creating the VM when dynamic memory is used.

Figure 3.5: Bug caused by the operating system where the code is being used.



```
92  def _create_vm_obj(self, vs_man_svc, vm_name, notes):
93      vs_data = self._conn.Msvm_VirtualSystemSettingData.new()
94      vs_data.ElementName = vm_name
95      vs_data.Notes = notes
96      # Don't start automatically on host boot
97      vs_data.AutomaticStartupAction = self._AUTOMATIC_STARTUP_ACTION_NONE
98
```

```
92  def _create_vm_obj(self, vs_man_svc, vm_name, notes, dynamic_memory_ratio):
93      vs_data = self._conn.Msvm_VirtualSystemSettingData.new()
94      vs_data.ElementName = vm_name
95      vs_data.Notes = notes
96      # Don't start automatically on host boot
97      vs_data.AutomaticStartupAction = self._AUTOMATIC_STARTUP_ACTION_NONE
98
99      # vNUMA and dynamic memory are mutually exclusive
100     if dynamic_memory_ratio > 1:
101         vs_data.VirtualNumaEnabled = False
102
```

Figure 3.6: Bug caused by an operating system where the code is being used.

From the demonstration of the concrete examples from ElasticSearch and Nova, it is clear that there are cases where the effect of the external changes, the new requirements and incompatibilities are what caused the failure. These bugs self-manifest without the necessity of having any previous or ancestral changes to introduce the bug into the source code. Hence, in order to find the origin of a bug, not only the fixed lines in the Bug-Fixing Commit have to be taken into account, the dependencies or ecosystem of such lines must also be considered. It is important to also note that there is no sense to lay the blame on a change as the cause of the bug, because they did not introduced them. However, only the fist change that manifest

the bug may be defined within the context of a concrete test system that contains all the dependencies.

3.3 Introduction of Version Control Systems

This thesis has considerable interest in version control systems (*VCS*), since it provides researchers and practitioners with all the necessary data to analyze and understand whether the moment when a bug is introduced the first time is also the moment when the bug manifests itself in the project for the first time. Thus, it is important to understand how the *VCS* work and what are the limitations these systems present in order to find the *BIC*. In this subsection we explain how developers and practitioners have been using them over the last decades and how some of their characteristics affect in the identification of *BICs*. Furthermore, there is a special interest for these systems because the selected projects to be analyzed in this thesis use a *VCS*.

VCS were developed to coordinate the shared access between many developers to the documents and files. These systems allow for simultaneous development of many branches and can detect any change committed in the source code. Then, these changes are saved along with the information of the timestamp and the identifier of the developer that make them. The VCS of a project keeps track of every modification to the source code and allows developers to turn back to any previous moment and compare earlier versions of the code. They can also revert to the last modification or return to a specific version of the project.

The most common use of these systems is to develop software, but they are also used in content management systems. The best-known VCS are Apache *Subversion (SVN)* and *Git*. The web-based hosting service for Git is GitHub[6], whereas RiouxSVN[7] hosts SVN. Although, Mercurial[8], Bazaar[9] and CVS[10] are also well known. These *VCS* can be classified into two classes; *Centralized VCS* that keep the history of changes on a central server where everyone requests the latest version and pushes the latest changes to. *Distributed VCS*, which

[6]https://github.com/

[7]https://riouxsvn.com/

[8]https://www.mercurial-scm.org

[9]http://bazaar.canonical.com/en/

[10]https://www.nongnu.org/cvs/

is when everyone has a local copy of the entire repository. Thus, it is not necessary to push changes of your work all time, and it allows for anyone to sync with any other team member.

SVN is a centralized *VCS* with a unique central repository which hosts all the data for the users. This prevents two users to edit a file at the same time, and also to push every single change immediately. Figure 3.7 shows how an SVN project works. In the picture there are three developers with access to modify the files; when developer *A* modifies a file she has to push the changes to the central repository. Developer *B* and *C* only have the new state of the project after pulling the central repository, furthermore they cannot make any changes to the same file as developer *A*. While developer *A* is making changes to a file, other developers are unable to make any modification until developer *A* finishes and pushes the changes.

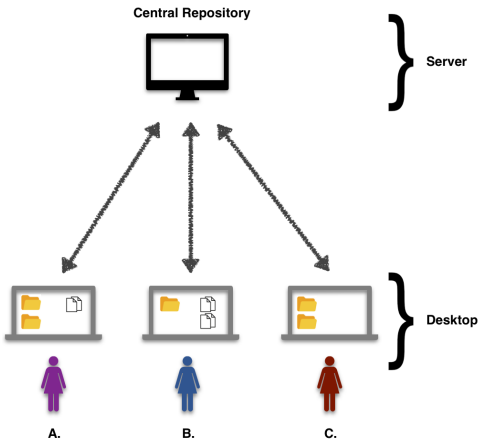


Figure 3.7: subversion

On the contrary, Git is a decentralized VCS, it has a unique central repository but each developer has their own local repository. Thus, developers can push their code to their private repository, while getting all the benefit of VCS. They can make their code better after some changes, before pushing the changes from their repository to the central repository and letting other people use their new code. Figure 3.8 presents how a Git project works, in the picture there are three developers with access to modify the files, each developer has a local copy of the central repository in their computer. Thus, developer *A*, *B* and *C* can modify the files as many times as they want before pushing the changes to the central repository. This may cause the developers to be in different states of the same project in their local repository because they have not pushed or pulled frequently. In Git, the same files can be modified at the same time, however if the developers have changed the same line, the system enters into a conflict

that has to be solved before developers can continue committing changes.

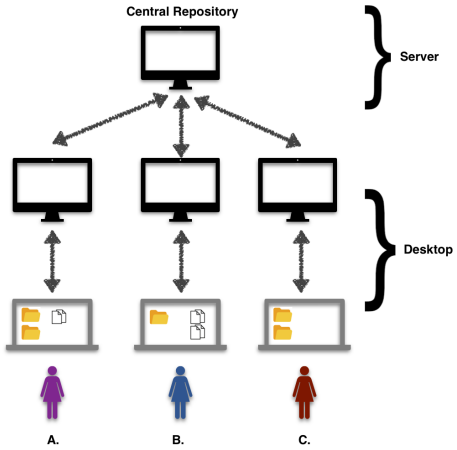


Figure 3.8: git-mercurial

Unfortunately, the current approaches used to identify the *BICs* are not understood for the *real* moment of inserting a buggy line in the source code, the cause of the bug, and it force practitioners to apply these approaches to all the projects without regard for the nature of the bug, the dependencies of the bug or even whether they use SVN or Git. It is important to understand why the VCS are relevant when using the approaches to find the *BIC*, specifically, because some of the approaches were developed to be used in SVN and they understand the branches in a different way than Git. For example, a branch in Git is only a pointer to some commit which can move and it causes the complete loss of the previous states whereas this is impossible in SVN. In other words, git does not keep the relationship of the changes in time, thus the approaches that use temporally windows to remove false positives or have faith that the precedence between commits is set by dates are not suitable for Git. Furthermore, another characteristic of Git is the wide variety of commands at our disposal such as *git diff, git remote, git bisect or git blame*. However, some other options such as *git merge, git rebase or git squash* can alter the natural order of the commits, in the sense that when the history of the commits are viewed using *git log*, it may occur that the commits are not sorted by dates as presumed in the beginning. Hence, the use of Git also can affect the correct identification of *BICs*, and since the project analyzed in this thesis uses Git as *VCS*. How these command affect the identification of the origin of a bug is described in detail.

To better understand how Git might affect the identification of *BIC*, this thesis explains three common scenarios with different commands available in Git. All of the scenarios use

the same simple example. In the example, the repository only has two diverging branches, the master branch and the feature branch. The commit hashes[11] are represented with integers, and in addition, these integers also represent timestamps, where a smaller number means an earlier commit.

Git Merge: The first scenario is when a developer uses git merge. This command is used to create a new commit. This new commit has two different parents and it is the only commit with this characteristic. The only time that git merge does not create a new commit is when the developer uses the *fast-forward merge* command. This situation occurs when there are no commits in another branch. Figure 3.9 shows the repository before and after the merge. In this case, to see the precedence between commits, *git log* can be typed after merging the branch and the result is a linear log sorted by date: 7,6,5,4,3,2,1.

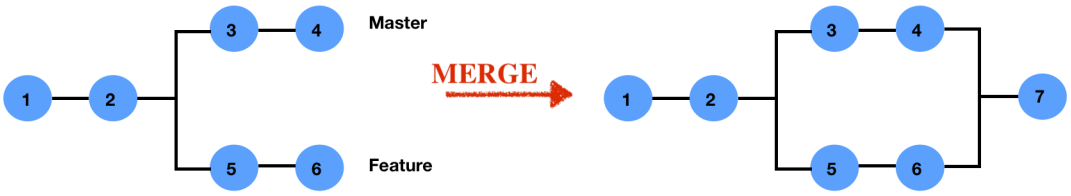


Figure 3.9: git-merge

Git Rebase: The second scenario is when a developer uses git rebase. This command recreates the work made from one branch onto another. For example, if a developer wants to rebase the master branch in the feature branch, for every commit that the master branch has that is not in the feature branch, a new commit will be created on top of the feature branch. The Figure 3.10 shows the output before and after the rebase from master branch onto the feature branch. In this case, git rebase has moved changes 3 and 4 to the master branch, and it has changed the hash in order to add a new one. In this case, to see the precedence between commits, *git log* can be typed after rebasing, the result is a linear log that is not sorted by date: 8, 7, 6, 5, 2, 1.

[11] a unique 40 character string generated by the SHA-1 algorithm which takes some data as input and generates the hash

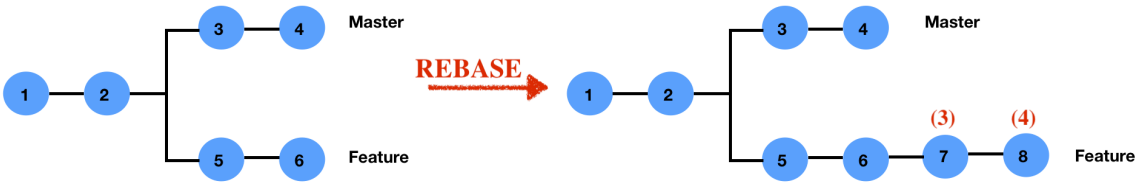


Figure 3.10: git-rebase

Git Squash The third scenario is when a developer uses git squash. This command takes a series of commits and squashes them down into a single commit. The main problem of this option is that the authorship of each squashed commit is lost, because these commits disappear from the history. Figure 3.11 shows the output before and after squashing commits 5 and 6 from feature branch onto master. In this case, Git Squash has combined changes 5 and 6 to create a new commit 7, this commit is then merged with the master branch. In this case, to see the precedence between commits we can type *git log* after squashing, the result is a linear log that is sorted by date: 7, 4, 3, 2, 1.

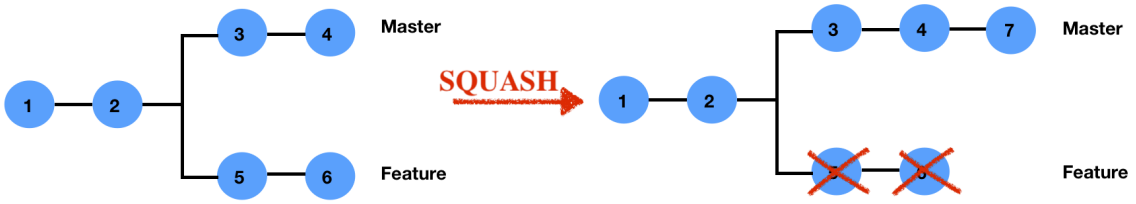


Figure 3.11: git-squash

At the end, the *VCS* facilitates the interaction between developers when a large number of them work in the same project. They are even more useful when the developers are distributed around the world working in different time zones. However, it must be kept in mind that in order to find the *BIC*, some available options in Git might hint the *real* moment of bug introduction because some commands such as *git merge, git rebase or git squash* might alter the natural order of the commits and their origin.

Chapter 4

Reproducibility and Credibility of the SZZ

Reproducibility of Empirical Software Engineering (ESE) studies is an essential part for improving their credibility, as it offers the opportunity for the research community to verify, evaluate and improve their research outcomes where concerns related to the reliability of the results may arise. Furthermore, it is one of the fundamental characteristics of the scientific method [González-Barahona and Robles, 2012]. Juristo and Vegas state that reproducibility "is important to increase and consolidate the body of empirical knowledge" [Juristo and Vegas, 2009], and Robles shows that reproducibility may be hindered by many factors [Robles, 2010].

Through this thesis, we adopt the definition of reproducibility by Madeyski and Kitchenham [Madeyski and Kitchenham, 2017] which claims that "reproducible research is the extent to which the report of a specific scientific study can be reproduced (in effect, compiled) from a reported text, data and analysis procedures, and thus can be validated by other researchers". Although there are differences between reproducibility and replication, we assume that a research work is more likely to be replicated when it incorporates means of reproducibility. However, reproducibility (and by extension credibility, since multiple replications of an experiment increase it [Juristo and Vegas, 2009]) may be a challenging work, as access to data sources, use of specific tools, availability of detailed documentation has to be handled. Thus, detecting elements that my hinder reproducibility should help strengthen the credibility of the empirical studies [Perry et al., 2000].

This chapter addresses how the scientific practice of the ESE research community affects

the reproducibility and credibility of the results of studies that use the SZZ algorithm, published in 2005 by Śliwerski, Zimmermann and Zeller [Śliwerski et al., 2005b] to detect the origin of a bug. The goal is to give a detailed description of the algorithm and explain its limitations and enhancements; then we detail the Systematic Literature Review *(SLR)* in the credibility and reproducibility of the SZZ. Notice that this section is based on the manuscript "Reproducibility and credibility in empirical software engineering: A case study based on a Systematic Literature Review of the use of the SZZ algorithm" published in the *Information and Software Technology* Journal. Regarding further details about this section, please refer to the paper.

4.1 Description of the SZZ algorithm.

In software engineering research, the SZZ algorithm is a popular algorithm for identifying Bug-Introducing Commits [da Costa et al., 2016]. SZZ relies on historical data from version control systems and bug tracking systems to identify change sets in the source code that introduce bugs. The algorithm addresses two different problems. The first problem is related to the linkage between the VCS and the issue tracking system in order to identify the Bug-Fixing Commit; the second problem is related to the identification of the Bug-Introducing Commit.

In the first part, the algorithm identifies, by means of a set of heuristics, Bug-Fixing Commits through using a technique that matches commits with bug reports labeled as *fixed* in the bug tracking system. Therefore it uses regular expressions to identify bug numbers and keywords in the commit messages that are likely to point out a real bug fixing change. This is possible since many projects have adopted the policy of recording the bug report number in the message of the commit that fixed the bug. Thus, the algorithm splits every log message into a stream of tokens to find the potential *bug number* with the use of regular expressions. For instance, the algorithm looks for keywords such as *fix(e[ed]), bugs, defects, patch* followed by a number.

The second part of the algorithm is concerned with the identification of the Bug-Introducing Commit(s). The algorithm employs the *diff* functionality implemented in source code management systems to determine the lines that have been changed (to fix the bug) between the

fix commit version and its previous version. Then, using the *annotate/blame* functionality, SZZ is able to locate who modified or deleted those lines for the last time in previous commit(s), and, whether they were committed before the bug was reported; those change(s) are flagged as suspicious of being the Bug-Introducing Commit(s).

As an example, Figure 4.1 shows three different snapshots o at different times and Figure 4.2 shows when and who changed the code in each commit. The first change is made by Alice on the 1st of June where *12cf3s*, is the Bug-Introducing Commit; the bug is introduced in line 21 because the condition for the *if* statement is used incorrectly. The bug was then reported on the 2nd of June. After that, Becky made the second change on the 3rd of June, *4asd23f*, which added code to the *foo()* function in lines 24, 25 and 26. Finally, the third change, *21esd33* is made by Cloe on the 5th of June. The change fixed the bug by modifying two lines: the buggy line 21 and the clean line 25. In the Bug-Fixing Commit, the modification of line 25 was purely semantic because the line retains with the same behavior from before its modification, in both cases the variable *bar* is incremented by one.

Figure 4.3 shows how the algorithm works. First, after the report of the bug notification *#159* in the issue tracking system, the algorithm identifies the Bug-Fixing Commit *21esd33* by looking in the logs for a commit with the commit message containing bug number *#159*. After that, the second part of the algorithm searches for the Bug-Introducing Commit by using the *diff* tool and *annotate/blame* tool in each of the lines modified in the Bug-Fixing Commit. In this example, lines 21 and 25 have been changed in the Bug-Fixing Commit in order to fix the bug, thus both lines are marked for suspicion of being the ones where the bug was introduced. These lines have been introduced in two different commits, however, as line 25 was introduced in a commit after the bug was reported, the algorithm removed it from the list of suspicious Bug-Introducing Commits. Consequently, only the commit that last modified line 21 can be blamed as the Bug-Introducing Commit. Thus, in this case, the SZZ algorithm correctly points out that *12cf3s* is the bug introducing change.

Unfortunately, the algorithm does not always behave as demonstrated in the above example. In some scenarios, the practitioners can notify some of the shortcomings which may cause the malfunction. These shortcomings, as well as some examples of the malfunction of the SZZ are detailed in the next section.

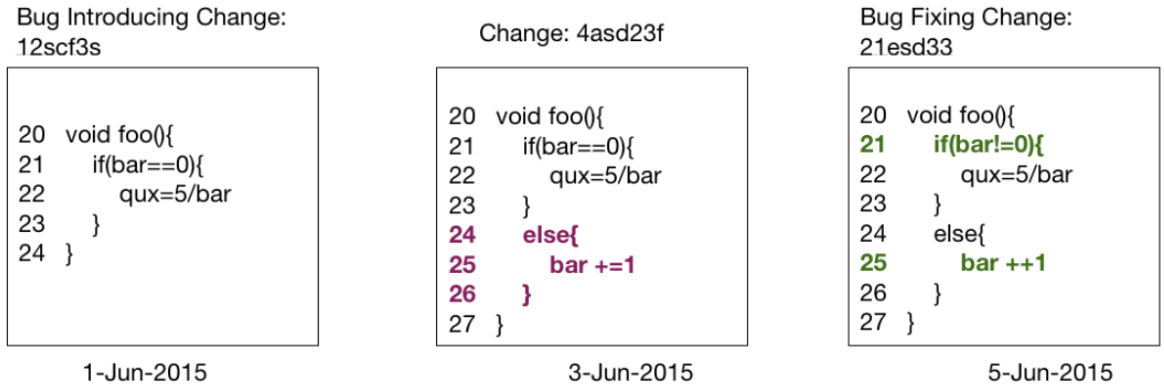


Figure 4.1: Example of changes committed in a file, the first change is the bug introducing change and the third change is the bug fixing change.

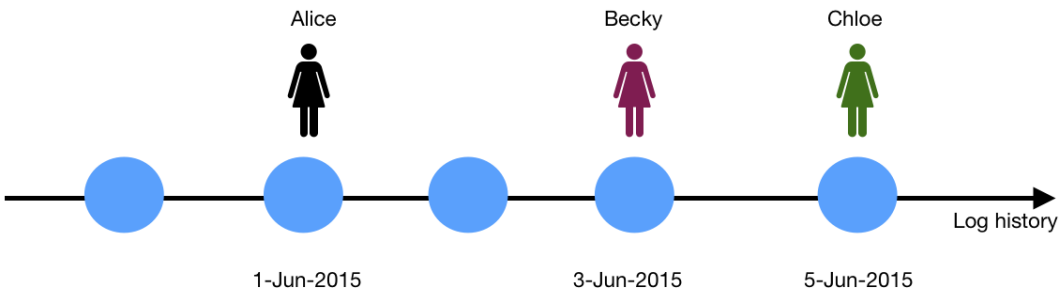


Figure 4.2: The changes were committed by Alice, Becky and Chloe in different days.

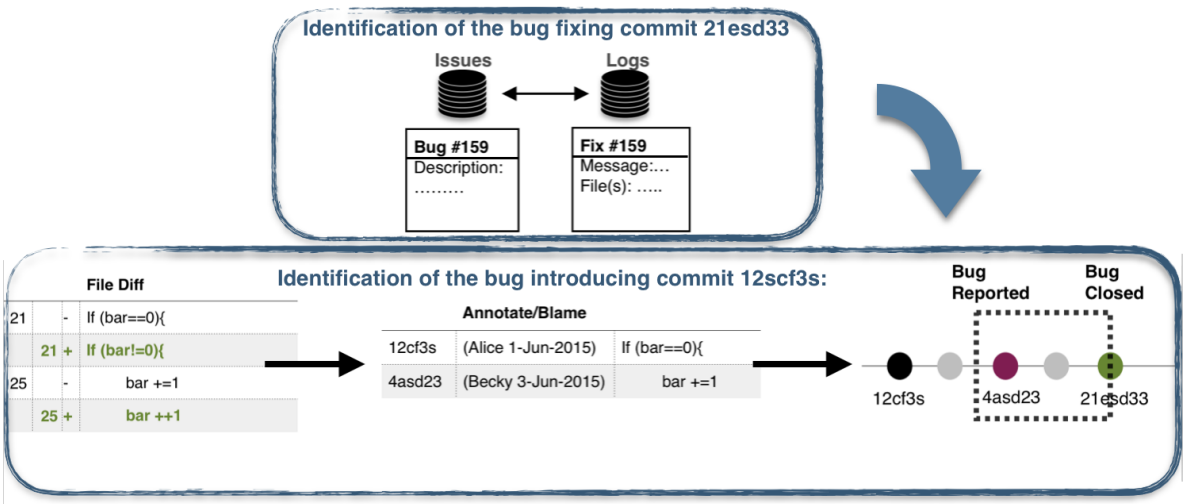


Figure 4.3: First and Second part of the SZZ algorithm

4.2 Shortcomings of the SZZ Algorithm

Despite SZZ being largely used in ESE to locate bug origins, it presents some shortcomings which makes it error prone. Table 4.1 offers a detailed overview of the shortcomings in SZZ as reported in the literature. Some of them have been previously explained in the Chapter 3, however, in this section they are described with examples based on the illustrative Figure 4.1.

In the first part of the algorithm the limitation lies in how bug reports are linked to commits. If the fixing commit (in the versioning system) does not contain a reference to the bug (usually the reference is the unique id assigned by the bug tracking system, but it could be certain keywords as well), it is very difficult to link both data sources. Sometimes this linking is incorrect as the Bug-Fixing Commit do not correspond to the bug report. If the fixing commit is not identified, the Bug-Introducing Commit cannot be determined and this causes a *false negative*[1]. Studies have demonstrated that 33.8% [Herzig et al., 2013] to 40% [Rodríguez-Pérez et al., 2016] of the bugs in the issue tracker are misclassified, i.e., issues categorized as bugs are actually functionality requests or refactoring suggestions. A false positive[2] occurs when a bug report does not describe a real bug, but a fixing commit is still linked to it. Herzig et al. pointed out that 39% of files marked as defective have never had a bug [Herzig et al., 2013].

In the second part of the algorithm, lines might be incorrectly identified by SZZ as the place for where the bug was introduced, causing a *false positive*. It may also be that the buggy line was not analyzed by SZZ, producing a *false negative*. In some cases, the bug had been introduced before the last change to the line; then, the history of the line has to be traced back until the *true* source of the bug is found [Williams and Spacco, 2008]. An example of this can be found when SZZ flags changes to *style* (i.e., non-semantic/syntactic changes such as changes to white spaces, indentation, comments, and some changes that split or merge lines of code) as Bug-Introducing Commits [da Costa et al., 2016], or when a project allows *commit squashing*, since this option removes authorship information resulting in more *false positives*. It may also happen that the bug may have been caused by a change in another part of the system [German et al., 2009]. A final possibility is that the bug fix modified the surrounding context rather than the problematic lines, thereby misleading the

[1]The definition of false negative has been addressed in the previous Chapter 3

[2]The definition of false positive has been addressed in the previous Chapter 3

Table 4.1: Shortcomings that can lead to false negatives when using SZZ.

Part	Type	Description
First part	Incomplete mapping [Bird et al., 2009a].	The fixing commit cannot be linked to the bug report.
	Inaccurate mapping [Bissyande et al., 2013].	The fixing commit has been linked to a wrong bug report, they don't correspond each other.
	Systematic bias [Bird et al., 2009a].	Linking fixing commits with no *real* bug reports.
Second part	Cosmetic changes, comments, etc [Kim et al., 2006c].	Variable renaming, indentation, split lines, etc.
	Added lines in fixing commits [da Costa et al., 2016].	The new lines cannot be tracked back.
	Long fixing commits [da Costa et al., 2016].	The larger the fix the more false positives.
	Semantic level is weak [Williams and Spacco, 2008]	Changes with the same behavior are being blamed.
	Clean changes [da Costa et al., 2016].	This changes start leading to bugs due to external changes or artifacts are being blamed.
	Commit Squashing [Gousios, 2013].	This practice might hide the real Bug-Introducing Commit, it combine/merge multiple commits into a single commit.

algorithm [Davies et al., 2014].

4.2.1 Some Examples

To provide more insights on the reasons why SZZ identifies false positive commits, we explain three possible scenarios based on the example shown in Figure 4.1 in where the algorithm identifies false Bug-Introducing Commits. However, in these scenarios the day of the bug report has changed from the 2nd of June to the 4th of June. Thus, the bug was not reported after the second change was committed, and after applying the heuristics, the SZZ is unable to remove the commit made by Becky from the list of suspicious Bug-Introducing Commits.

The first scenario is represented in the code of Figure 4.1 and is related to the problem of identifying more than one Bug-Introducing Commit, and how practitioners should behave. This example shows how after applying the algorithm, it identifies two commits as being the possible Bug-Introducing commits. While the commit made by Alice is correctly identified as a Bug-Introducing commit, the change made by Becky is a false positive because of two main reasons; i) she did not introduce any bug when the lines were committed and, ii) she modified the code but it maintained the same logic of its previous state where line 25 still increments one unit on the variable *"bar"*.

The second scenario is represented in the code of Figure 4.4. In this scenario, the semantic changes made by Becky are hiding the real Bug-Introduction Commit because she decided to replace the name of the variable *"bar"* for *"people"*. Even though this change may appear

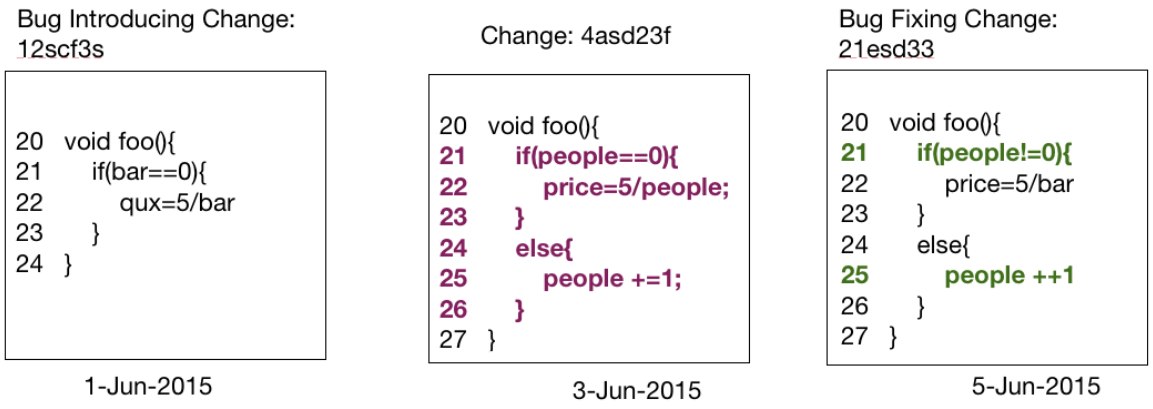


Figure 4.4: Example where semantic changes in the buggy line hide the bug introducing change and the SZZ cannot identify it.

to be inoffensive, it modified the variable name in the buggy line 21 hinting the true Bug-Introducing Commit. Thus, when applying the SZZ algorithm, the outcome is that the last commit which touched buggy line 21 is the commit made by Becky, when in fact, the bug was already in the line when Becky decided to change the name of the variable. In this case, the SZZ identifies a commit but it is not the Bug-Introducing Commit.

Finally, the third scenario is represented in the Figure 4.5. It shows an example where unchanged lines introduced the bug. In this example, Alice wrote the function *foo()*, but she forgot to add the *if* condition which checks whether the variable *"bar"* is not equal to 0, otherwise the logic in line 22 will fail because it is not allow to divide a variable between 0. Thus, in order to fix the bug Chloe added the *if* condition in the Bug-Fixing Commit. Therefore when the SZZ algorithm is applied, the outcome blames a wrong change as the Bug-Introducing Commit because the bug was fixed by adding a new line, and the SZZ algorithm cannot track back the line.

4.2.2 Enhancements of SZZ

After describing the shortcomings of the SZZ found in the literature, the focus is now on how researchers have addressed them over time and in how far the enhancements have mitigate the problems.

The misclassification problem has been further investigated by researchers, aiming at mitigating the limitations found in the first part of SZZ [Herzig et al., 2013], [Tan et al., 2015].

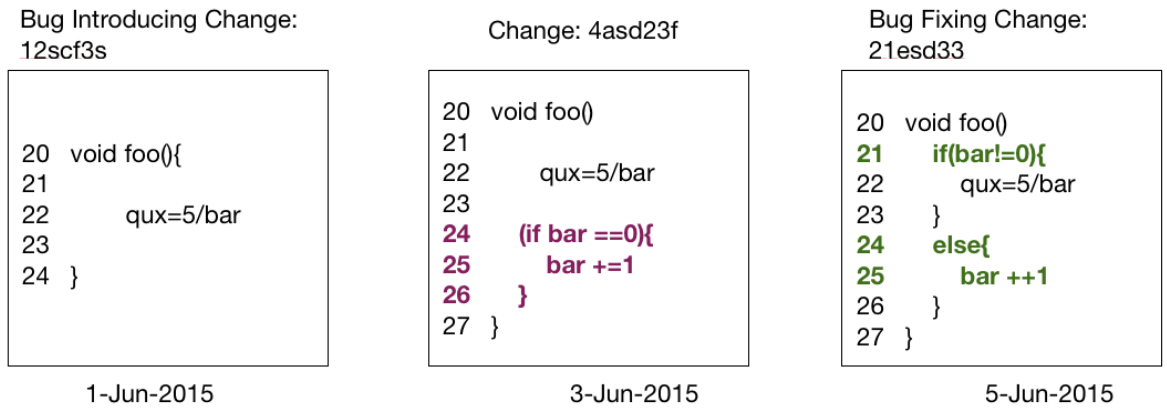


Figure 4.5: Example where unchanged lines introduced the bug, and SZZ cannot identify the Bug-Introducing Commit.

Tools and algorithms have been created based on the information from version control systems and issue tracking systems to map bug reports to fixing commits [Wu et al., 2011], [Nguyen et al., 2012], [Le et al., 2015], [Sun et al., 2017]. As a result of these efforts, the first part of SZZ has seen how its accuracy has significantly increased.

Related to the second part of SZZ, two main improvements have been proposed in the literature; they are referred to as SZZ-1 and SZZ-2:

- *SZZ-1*) Kim *et al.* suggest an SZZ implementation that excludes cosmetic changes, and propose the use of an *annotation graph* instead of using *annotate*[3] [Kim et al., 2006c].

- *SZZ-2*) Williams and Spacco propose to use a mapping algorithm instead of annotation graphs; this approach uses weights to map the evolution of a source code line and ignores comments and formatting changes in the source code with the help of `DiffJ`, a Java-specific tool [Williams and Spacco, 2008].

The second part of the algorithm however still has room for further improvements such as the work from Da Costa *et al.* who have created a framework to eliminate unlikely Bug-Introducing Commit from the outcome of SZZ. Their framework is based on a set of requirements that consider the dates of the suspicious commit and of the bug report [da Costa et al., 2016]. By removing the commits that do not fulfill these requirements, the number of false positives provided by SZZ is lowered significantly. . As can be seen, addressing the limitations of the

[3]Notice that annotate is used in SVN and blame is used in git.

SZZ often requires a manual, tedious validation process which can be impractical at times.

4.3 Systematic Literature Review on the use of SZZ algorithm

Through this subsection we present the Systematic Literature Review (SLR) on the use of the SZZ algorithm in 187 academic publications in order to address how the scientific practice of the ESE research community affects the reproducibility and credibility of the results. In particular, we want to address studies that use the SZZ algorithm, published in 2005 in "When do changes induce fixes?" by Śliwerski, Zimmermann and Zeller [Śliwerski et al., 2005b] at the MSR workshop[4]. SZZ has been largely used in academia, counting, as of May 2018, with more than 610 citations in Google Scholar[5].

The purpose of a SLR is to identify, evaluate and interpret all available studies relevant to a particular topic, research question, or effect of interest [Kitchenham and Charters, 2007]. A SLR provides major information about the effects of a particular topic across a wide range of previous studies and empirical methods. As a result, a SLR should offer evidence with consistent results and suggest areas for further investigation. To address the SLR, we followed the approach proposed by Kitchenham and Charters [Kitchenham and Charters, 2007] in order to analyze the credibility and reproducibility of the SZZ algorithm. Therefore, we address the following questions:

1. **What is the impact of the SZZ algorithm in academia?** The SZZ algorithm has been shown to be a key factor in locating when a change introduced fixing commits. However, many papers use only the first part of the algorithm to link bug fix reports to commits. As the second part of SZZ is shown to encompass significant threats, we identify only those publications that use both parts, or at least the second part, of the SZZ algorithm. In addition to this, we offer other metrics on the publications, such as the number of authors and the geographic diversity of the institutions they work for, in order to provide insight of how widespread the use of SZZ is. Furthermore, one of

[4]MSR is today a working conference, but at that time it was a co-located workshop with ICSE in its second edition.

[5]https://scholar.google.es/scholar?cites=3875838236578562833

our goals addresses the maturity and diversity of the publications where SZZ has been used in order to understand its audience. We address the *maturity* of a publication by analyzing whether it has been accepted in a workshop, a conference, a journal, or a top journal. Diversity is given by the number of distinct venues where publications using SZZ can be found.

2. **Are studies that use SZZ reproducible?** Reproducibility is a crucial aspect of a credible study in ESE [González-Barahona and Robles, 2012]. Piwowar *et al.* state that reproducibility improves the impact of research [Piwowar et al., 2007]. In addition, when a research work incorporates reproducibility, it is more likely to be replicated. However, there is evidence in the ESE literature that replicable studies are not common [Robles, 2010]. By providing a replication package (or a detailed description of the analysis and the environment and data used), the authors facilitate others to replicate or to reproduce their experiment, which increases the credibility of their results [Juristo and Vegas, 2009]. In addition, replication packages help in the training of novice researchers [Madeyski and Kitchenham, 2017]. To provide trustworthy results in ESE research, authors should offer a replication package and/or a detailed description of the research steps, and the environment and data used. This would allow others to reproduce or replicate their studies [González-Barahona and Robles, 2012].

3. **Do the publications mention the limitations of SZZ?** It has already been shown that limitations of SZZ are well-known in the research literature but there is still the question of how many papers report any of these. Therefore, this chapter also studies whether authors mention the limitations of SZZ that may affect their findings, be it in the *description of the method*, in the *threats to validity* or in the *discussion*.

4. **Are the improvements to SZZ (SZZ-1 and SZZ-2) used?** The improved versions of the original SZZ algorithm address some of its limitations. We analyze whether any of the improvements to the SZZ algorithm can be found in the primary studies included in the SLR. Thus, we search for any mention of their use, be it in the *description of the method* or in the *threats to validity*. Answering this research question enables further understanding on how authors who use SZZ behave given the limitations of SZZ.

4.3.1 Inclusion Criteria

After enumerating the questions, we present the inclusion and exclusion criteria for the SLR. In addition, we describe the search strategy used for primary studies, the search sources and the reasons for removing papers from the list. The inclusion criteria address all published studies written in English that cite either:

1. The publication where SZZ was originally described,"When do changes induce fixes?" [Śliwerski et al., 2005b], or

2. (at least) one of the two publications with improved versions of the algorithm, "Automatic Identification of Bug-Introducing Changes" [Kim et al., 2006c] and "SZZ Revisited: Verifying When Changes Induce Fixes" [Williams and Spacco, 2008].

There was no need to further investigate the references to the resulting set of publications (a process known as *snowballing*): if one of these papers contained as well a reference to the papers that fit the inclusion criteria, it is assumed to be already in our sample.

Before accepting a paper into the SLR, we excluded publications that are duplicates, i.e., a *matured* version (usually a journal publication) of a *less matured* version (conference, workshop, PhD thesis...). In those cases, we only considered the *matured* version. When we found a short and a long version of the same publication, we have chosen the longer version. However, in those cases where the publication is a PhD thesis and a related (peer-reviewed) publication exists in a workshop, conference or journal, the thesis is discarded in favor of the latter, because conference and journal publications are peer-reviewed whereas a PhD theses are not. Documents that are a *false alarm* (i.e., not a *real*, scientific publication) have also been excluded.

4.3.2 Search Strategy used for Primary Studies

The studies were identified using Google Scholar and Semantic Scholar as of November 8th 2016. We have searched exclusively in Google Scholar and Semantic Scholar because of i) their high accuracy in locating citations, providing more results than other databases (from Table 4.2 it can be seen that they contain three times more citations than other platforms, such as the ACM Digital Library), and ii) because it was observed that they offer a superset of the

Table 4.2: Number of citations of the SZZ, SZZ-1 and SZZ-2 publications by research databases.

	Google Scholar	Semantic Scholar	ACM Digital Library	CiteSeerX
# SZZ	493	295	166	26
# SZZ-1	141	100	60	18
# SZZ-2	26	15	8	0

other databases, i.e., it was checked checked that no publication in the other sources is missing from the list provided by Google Scholar and Semantic Scholar. However, Google Scholar gives many *false alarms*, in the sense that they are not publications but slide sets, notes, etc. Examples of those *false alarms* are "Strathprints Institutional Repository"[6] or "Home Research"[7]), which was removed manually from our set. Some academic databases which are commonly used for SLRs, such as Scopus, could not be employed to gather citations, because SZZ was published at a time when MSR was a workshop, and thus the original publication [Śliwerski et al., 2005b] is not included in those databases.

4.3.3 Study Selection Criteria and Procedures for Including and Excluding Primary Studies

Table 4.3 shows that our searches elicited 1,070 citation entries. After applying the inclusion criteria described above, a list of 458 papers was obtained. This process was performed by the first author. The process is objective, as it involves discarding false alarms, duplicates, and papers not written in English.

Then, the first author analyzed the remaining 458 papers looking for the use of SZZ, SZZ-1 and SZZ-2 in the studies. This resulted in 193 papers being removed because of three main reasons: i) they only cited the algorithm as part of the introduction or related work but never used it, ii) they only cited the algorithm to support a claim during their results or the discussion, and iii) the papers were essays, systematic literature reviews or surveys. This process was discussed in advance by all the authors. The second author partially validated the process by analyzing a random subset comprising 10% of the papers. The agreement

[6] `https://core.ac.uk/download/pdf/9032200.pdf`

[7] `http://ieeexplore.ieee.org/document/1382266/\#full-text-section`

Table 4.3: Number of papers that have cited the SZZ, SZZ-1 and SZZ-2 publications by joining the research databases Google Scholar and Semantic Scholar during each stage of the selection process.

Selection Process	#SZZ	#SZZ-1	#SZZ-2
Papers extracted from the databases	788	241	41
Sift based on false alarms	29 removed	10 removed	2 removed
Sift based on not available/English writing	40 removed	4 removed	0 removed
Sift based on duplicates	308 removed	187 removed	32 removed
Full papers considered for review	411	40	7
Removed after reading	149 removed	32 removed	4 removed
Papers accepted to the review	262	8	3

between both authors was measured using Cohen's Kappa coefficient, resulting in a value of 1 (perfect agreement). These papers were removed on the basis that they do not answer our research questions. After this process 273 papers were included in this SLR.

4.3.4 Quality Assessment Criteria

The approach employed to study the quality assessment is based on Kitchenham and Charter's [Kitchenham and Charters, 2007] concept of quality. Thus, the assessment is focused on identifying only papers that report factors related to the credibility and reproducibility of the studies using SZZ. The specific criteria are described in the next phase.

Phase 1: Establishing that the study uses the complete SZZ algorithm

In this SLR we only consider studies that use the complete algorithm, or at least its second part. Even though shortcomings have been reported in both parts of the SZZ algorithm (see Section 4.1), most of the shortcomings present in the first part have been successfully addressed in the last years.

To analyze the ease of reproducibility of each study, we looked for (1) a replication package provided by the authors or (2) a detailed description. A detailed description must have: (a) precise dates when the data were retrieved from the projects under analysis, (b) the versions of the software and systems used in the analysis, (c) a detailed description of the methods used in each phase of the study, and (d) enumerate the research tools used. It should be

noted whether the that we did not inspect whether the replication package is still available, or whether elements in the package make the study reproducible.

It is also important to point out that during this SLR an assumption was made on the availability of the replication package and that it was available at the time when the articles were submitted. And we do not claim the availability of these packages in the long term, because it is possible that some factors such as a change in the author's affiliation, an inaccessible URL or other reasons might cause the package to not be available anymore. For instance, the reproduction package from the original SZZ paper [Śliwerski et al., 2005b] is no longer available.

Applying our criteria to the set of 273 papers, we obtain 187 papers that fulfill this criterion.

4.3.5 Extracting Data from Papers

We have read and analyzed the 187 papers, and extracted the following data information to answer the questions:

1. Title,
2. Authors,
3. Countries of the authors' institutions,
4. Purpose of the study,
5. Outcome of the study, and
6. Venue and class of publication (journal, conference, workshop or university thesis).

Then, in a second phase, we have carefully analyzed each publication looking:

1. For a replication package (as in [Robles, 2010]).
2. For a detailed description of the methods and data used (as in [Robles, 2010]).
3. Whether shortcomings are mentioned.
4. Whether a manual inspection to verify the results has been done, to answer.

5. Whether authors use an improved version of SZZ (differentiating between a version found in the research literature and *ad-hoc* improvements implemented by the authors).

4.3.6 Overview across Studies

Cruzes and Dybå reported that synthesizing findings across studies is specially difficult, and that some SLRs in software engineering do not offer this synthesis [Cruzes and Dybå, 2011]. For this SLR we have extracted and analyzed both quantitative and qualitative data from the studies, but we have not synthesized the studies, as they are too diverse. Doing a meta-analysis would offer limited and unstructured insight [Clarke and Oxman, 2000] and results would suffer from some of the limitations in SLRs published in other disciplines [Rosenthal and DiMatteo, 2001]. Thus, we combined both our quantitative and qualitative data to generate an overview of how authors have addressed the reproducibility and credibility of the studies. The results are presented in Subsection 4.3.7. In addition, we have constructed a quality measure[8] that assesses the ease of reproducibility of a study. This measure is based on the score of five characteristics of the papers that was looked for in the second reviewing phase. If the questions were answered positively, the paper was marked with a positive score, otherwise with a 0:

1. Does the study report limitations of using SZZ? (score = 1 point)
2. Do the authors carry out a manual inspection of their results? (score = 1 point)
3. Does the study point to a reproducibility package? (score = 2 point)
4. Does the study provide detailed description of the methods and data used? (score = 1 point)
5. Does the study use an improved version of SZZ? (score = 2 point)

We believe that elements with a higher impact on ease reproducibility of the studies should be scored with 2 points. Partial scores are summed up to obtain an overall score. Table 4.4 offers a mapping of this overall measure with the ease of reproducibility of a study.

[8]The main goal of this quality measure is to determine the reproducibility and credibility of the studies in the moment in which the study was submitted.

Table 4.4: Mapping of overall score and the quality measure on ease of reproducibility of a study.

Score	Quality Measure
0 – 1	*Poor* to be reproducible and to have credible results
2 – 4	*Fair* to be reproducible and to have credible results
5 – 6	*Good* to be reproducible and to have credible results
7	*Excellent* to be reproducible and to have credible results

Table 4.5: Quantitative results from the 187 studies.

Purpose:	Outcome:	Common Metrics:	Versioning:
Bug Prediction (BP) (32%) Bug Proneness (BProne) (27%) Bug Detection (BD) (22%) Bug Localization(BL) (19%)	New Method/Approach (NM) (41%) Empirical Study (ES) (37%) New Tool (NT) (8%) Human Factors (HF) (8%) Replication (R) (3%) Create a Dataset (D) (2%)	Commits (26%) Bug Reports (16%) LOC (15%) Changes (12%) Files (8%) Faults (8%) Revisions (6%) Modules (4%)	CVS (50%) Git (28%) SVN (25%) Mercurial (6%)

Table 4.6: Results of the calculating the quality measure of reproducibility and credibility.

Quality Measure	#papers
Poor to be reproducible and to have credible results	34 (18%)
Fair to be reproducible and to have credible results	126 (67%)
Good to be reproducible and to have credible results	24 (13%)
Excellent to be reproducible and to have credible results	3 (2%)

4.3.7 Results of Questions

Data analysis

Here, we present our quantitative and qualitative data analysis extracted from the 187 studies analyzed during the SLR. Table 4.5 summarizes the frequency of the different outcomes, purposes, versioning systems and metrics in the studies. The most common purpose is *bug prediction* followed by *bug proneness*. The most common outcomes are the *development of a new method/approach* and the *evidence of empirical results*.

The qualitative data, summarized in Table 4.6, consists in the number of papers that belong to each quality measure levels. It can be observed that 67% of the papers present a fair reproducibility measure, and only 2% of them provide excellent means to be reproducible.

The combination of quantitative and qualitative data is addressed in Table 4.7, where the purpose and outcome of each individual study is grouped according to our quality measure. It can be observed that the distribution by purpose or outcome does not differ much from the global distribution, although studies offering new tools (NT) and replications (R) offer slightly better results than the rest.

Table 4.7: Distribution of the ease of reproducibility quality measure of studies depending on purpose and outcome. Acronyms are defined in Table 4.5.

Quality	**Purpose**				**Outcome**					
	BP	**BProne**	**BD**	**BL**	**ES**	**HF**	**NM**	**NT**	**R**	**D**
Poor	15 (25%)	10 (19%)	6 (15%)	3 (9%)	14 (20%)	2 (13%)	17 (22%)	1 (7%)	0 (0%)	0 (0%)
Fair	38 (64%)	32 (62%)	29 (72%)	27 (77%)	49 (70%)	13 (81%)	50 (65%)	8 (53%)	3 (60%)	3 (75%)
Good	6 (10%)	9 (17%)	5 (13%)	4 (11%)	5 (7%)	1 (10%)	10 (13%)	5 (33%)	2 (40%)	1 (25%)
Excellent	1 (1%)	1 (2%)	0 (0%)	1 (3%)	2 (3%)	0 (0%)	0 (0%)	1 (7%)	0 (0%)	0 (0%)

The type of paper (journal, conference, workshop and university thesis) as well as the size (short, medium, long) of the paper might be a restriction to provide means of reproducibility. We have labeled paper size to be less than 8 pages for short, from 9 to 50[9] pages for medium, and more than 50 pages for long publications. Table 4.8 reports the type and the size of each individual study grouped according to our quality measure of reproducibility and credibility. Again, the results do not differ much from the global distribution. However, it can be seen that (a) workshop papers perform worse than the rest, and (b) reproducibility increases slightly with the size of the publication.

What is the impact of the SZZ algorithm in academia?

Figure 4.6 shows the evolution of the number of publications that have cited and used SZZ, SZZ-1 or SZZ-2 up to November 2016. The SZZ algorithm was published in 2005 and afterwards 178 studies have cited it. SZZ-1 was published in 2006 and its number of citations is 53. Finally, SZZ-2 was published in 2008 and counts with 16 publications[10].

[9]We argue that master and PhD theses should be categorized as long publications. We have chosen 50 as limit between medium and long papers because in our data we have observed that all master theses have more than 50 pages whereas none of the journals articles have more than 50 pages.

[10]Note that a paper can cite more than one version of SZZ.

Table 4.8: Results to measure the ease of reproducibility and credibility of the studies depending on the type of paper and their size.

Quality	**Venue**				**Size**		
	Journal	**Conference**	**Workshop**	**University**	**Short**	**Medium**	**Long**
Poor	5 (12%)	20 (20%)	5 (38%)	4 (13%)	10 (26%)	21 (17%)	3 (13%)
Fair	32 (76%)	65 (63%)	7 (54%)	22 (74%)	25 (64%)	85 (68%)	16 (70%)
Good	4 (10%)	15 (15%)	1 (8%)	4 (13%)	4 (10%)	16 (13%)	4 (17%)
Excellent	1 (2%)	2 (2%)	0 (0%)	0 (0%)	0 (0%)	3 (2%)	0 (0%)

Table 4.9: Most frequent types of publications using (the complete) SZZ (N=187). *# different* counts the different venues, *# publications* counts the total number of publications in that type of venues.

Type	# different	# publications
Journals	21	42
Conferences & Symposiums	40	102
Workshops	13	13
University theses	20	30

The number of studies per year peaked in 2013, with 30 papers using an SZZ version. In general since 2012, the number of studies using this algorithm seems to have stabilized with over 15 citations/year for the use of the complete algorithm.

Table 4.9 shows the different types of venues with publications where SZZ has been used. We have classified the venues in four different categories: university theses, workshop papers, conference and symposium publications, and journal articles. Master theses, student research competitions and technical reports have been grouped under *university theses*. Diversity and maturity can be found in the sample, as it can be seen from the number of different venues (second column in Table 4.9) and the considerable number of journal publications (third column in Table 4.9).

Table 4.10 offers further insight into venues that have published more studies that use SZZ. The most frequent is the conference where SZZ itself was presented, the Working Conference on Mining Software Repositories (MSR). Two top conferences, such as the International Conference on Software Maintenance and Evolution (ICSME) and the International Conference of Software Engineering (ICSE), are second and third. SZZ can also been fre-

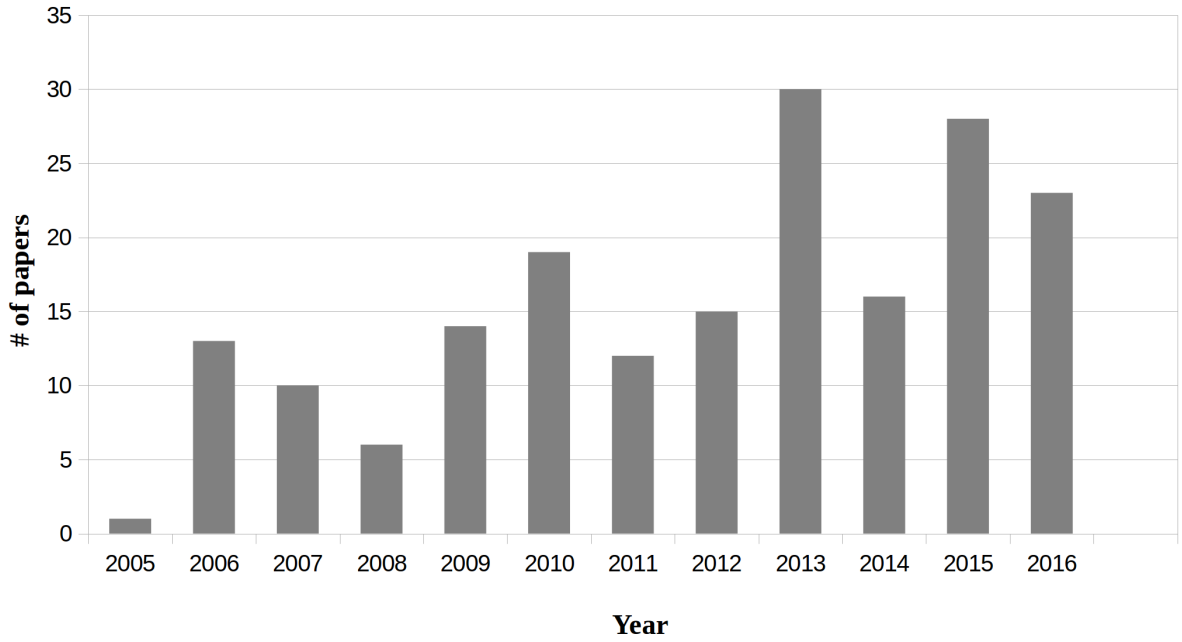


Figure 4.6: Sum of the number of publications using the (complete) SZZ, SZZ-1 or SZZ-2 by year of publication (N=187).

quently found in high quality journals, such as Empirical Software Engineering (EmSE) and Transactions on Software Engineering (TSE). The quality rating of conferences given in Table 4.10 has been obtained from the GII-GRIN-SCIE (GGS) Conference Rating[11]; Class 1 (CORE A*) conferences are considered *excellent, top notch events* (top 2% of all events), while Class 2 (CORE A) are *very good events* (given by the next top 5%). For journals, we offer the quartile as given by the well-known Journal Citation Reports (JCR) by Clarivate Analytics (previously Thomson Reuters).

The impact of the SZZ algorithm is significant: 458 publications cite SZZ, SZZ-1 or SZZ-2; 187 of these use the complete algorithm. The popularity and use of SZZ has risen quickly from its publication in 2005 and it can be found in all types of venues (high *diversity*), ranging from top journals to workshops and PhD theses; SZZ related publications have often been published in high quality conferences and top journals (high *maturity*).

[11] http://gii-grin-scie-rating.scie.es/

Table 4.10: Most popular media with publications using SZZ, SZZ-1 and SZZ-2 (N=187). "J" stands for journal and "C" for conference/symposium.

Type	Name	Rating	# papers)
C	Conf Mining Softw Repositories (MSR)	Class 2 - CORE A	15 (8%)
C	Intl Conf Software Eng (ICSE)	Class 1 - CORE A*	12 (6%)
C	Intl Conf Soft Maintenance (ICSME)	Class 2 - CORE A	10 (5%)
J	Empirical Software Eng (EmSE)	JCR Q1	9 (5%)
J	Transactions on Software Eng (TSE)	JCR Q1	9 (5%)
C	Intl Symp Emp Soft Eng & Measurement (ESEM)	Class 2 - CORE A	8 (4%)
C	Intl Conf Automated Softw Eng (ASE)	Class 2 - CORE A	7 (4%)
C	Symp Foundations of Software Eng (FSE)	Class 1 - CORE A*	6 (3%)

Table 4.11: Publications by their reproducibility: Rows: *Yes* means the number of papers that fulfill each column, whereas the complement is *No*. Columns: *Package* is when they offer a replication package, *Environment* when they provide a detailed methodology and dataset. Note that *Both* is the intersection of *Package* and *Environment*. (N=187)

	Package Only	Environment Only	Both	None
Yes	19	72	24	72
No	168	96	163	115

Are studies that use SZZ reproducible?:

Table 4.11 shows the number of analyzed studies that a) offer a replication package or b) have carefully detailed the methodology and the data used to allow the reproducibility of their studies. We have classified the publications in four groups: i) publications that offer a replication package (*Package*), ii) publications that detail the methodology and data used (*Environment*), iii) publications that have both (*Both*), and iv) none (*None*).

From the 187 analyzed publications, 43 offer a replication package, and 96 carefully detail the steps followed and the data used. Furthermore, only 24 provide both the replication package and the detailed methodology and data. 72 of the papers do not offer a replication package or a detailed description of the methodology and data.

Only 13% of the publications using any of the variants of SZZ provide a replication package and carefully describe each step to make reproduction feasible. 39% of the papers do not

Table 4.12: Number of publications that mention limitations of SZZ in their Threats To Validity (TTV). Mentions can be to the first (TTV-1st), second (TTV-2nd) or both parts (Complete-TTV). The absence of mentions is classified as No-TTV. Note that *Complete-TTV* is the intersection of *TTV-1* and *TTV-2*.

	No-TTV	TTV-1st only	TTV-2nd only	Complete-TTV
Yes	94	44	10	39
No	93	143	177	148

provide replication package or a detailed description of each step, making their reproduction very unlikely.

Do the publications mention the limitations of SZZ?:

We have classified publications into four groups, depending on how they address limitations in SZZ as a threat to validity (*TTV*). Thus, we have publications that i) mention limitations of the complete algorithm (*Complete-TTV*), ii) mention only limitations in the first part (*TTV-1st*), ii) mention only limitations in the second part (*TTV-2nd*), and iv) do not mention limitations at all (*No-TTV*).

Table 4.12 offers the results of the analysis. From the 187 publications, only 39 mention limitations of the complete SZZ as a threat to validity, whereas 83 refer to limitations in the first part, and 49 only mention it for the second part. The rest, 94 studies, do not mention any limitation.

In a more profound review, we found 82 publications where a manual inspection had been done to assess these limitations: 33 of them referred to issues related to the first part of the SZZ algorithm, while 30 analyzed aspects from the second part (i.e., the Bug-Introducing Commit). In the remaining 19 papers, the manual validation of results did not focus on outputs of the SZZ algorithm.

Almost half (49.7%) of the analyzed publications mention limitations in the first or second part of SZZ as a threat to validity. Limitations to the first part are reported more often than to the second part.

Table 4.13: Number of papers that have used the original SZZ, the improved versions of SZZ or some adaptations to mitigate the threat.

	Original SZZ only	SZZ-improved only	SZZ-mod only	Mixed
# publications	71 (38%)	26 (14%) [a]	75 (40%)	15 (8%)

[a] 22 (12%) of the papers use SZZ-1 and only 4 (2%) of the papers use SZZ-2.

Do the publications mention the limitations of SZZ?:

It is difficult to determine which improvement has been used when the authors do not mention it in the publication. Thus, if the authors do not explicitly specify of having used an improvement, we assume that they use the *original* version of SZZ. The publications are classified into one of the following groups, depending on the kind of improvement they used:

- *original SZZ*: Those only citing the original version and not mentioning improvements.
- *SZZ-1*: Those citing the improved version of Kim *et al.* [Kim et al., 2006c].
- *SZZ-2*: Those citing the improved version of Williams and Spacco [Williams and Spacco, 2008].
- *SZZ-mod*: Those citing the original SZZ with some (own) modification (by the authors). Publications in this group contain statements like "we adapt SZZ", "the approach is similar to SZZ' or "the approach is based on SZZ", but do not refer explicitly to SZZ-1 or SZZ-2.

Table 4.13 shows how many publications have used improvements to SZZ to mitigate the limitations of the original SZZ. The largest groups correspond to publications where authors use their own enhancements/adaptations (40%) and the original SZZ algorithm (38%). This suggests that researchers prefer to address the limitations of SZZ themselves instead of using enhancements proposed by others. Notice that the "Mixed" column in Table 4.13 refers to papers that have used either the original version, the improved versions or some adaptations of SZZ in the same study (e.g., to compare their performance in the same case study).

Chapter 5

The Theory of Bug Introduction

The proper understanding of the bug introduction process is an essential part of any research work related to the identification of the origin of a bug. The study of the changes in a Bug-Fixing Commit (*BFC*) to locate the origin of a bug is the foundation for researchers to carry out studies in other disciplines of Software Engineering. For example, researchers need to identify where previous bugs were introduced and obtain their characteristics in order to build models that can predict future bugs. They should define and understand how a bug was introduced into the project and what were its causes before building any classification model. To detect bugs, researchers can develop algorithms based on the learning from previous bugs patterns.

To identify the Bug-Introducing Commit (*BIC*) in Empirical Software Engineering, researchers rely on a common practice that analyzes static metadata retrieved from previous changes to the modified lines of a *BFC* (i.e, developer information, number of lines of code introduced, type of changes, etc.). Although these metadata can help to understand were the bug was introduced, it is necessary to keep in mind that the software evolves. Lehman formulated some laws that state how a E-type software system evolves [Lehman, 1979], [Lehman et al., 1998]. Some of the Lehman's laws to take into account are:

- **First law - continuing change:** "a system must be continually adapted or it becomes progressively less satisfactory".

- **Second law - increasing complexity:** "as a system evolves, its complexity increases unless work is done to maintain or reduce it".

- **Sixth law - continuing growth:** " the functional content of an system must be continually increased to maintain user satisfaction over its lifetime".

These laws explain that E-type[1] software systems are continuing being adapted, with more complexity and functionality over the time. These factors help to understand why it is possible that a piece of code that did not insert any error may manifest the bug later, due to the evolution of the software. For example, there could be instances where other external software changes to the project are affecting its source code and it caused the malfunction. When researchers are looking for the origin of the bug it is important that they have this factors into account, in the sense that it may be possible that a line did not inserted the error in the at the moment of their written, but the evolution of the system caused that this line manifested itself the bug. Thus understanding the introduction of bugs from a static point of view may lead to problems when locating the line(s) that inserted the bug. It is therefore logical to think that a more in-depth knowledge of when and how a bug is introduced will make the state-of-the-art techniques vary in accuracy accurate

There are two primary moments that should be understood and distinguished when analyzing the origin of a bug, the bug manifestation moment and the bug introduction moment. The second moment refers to when a failure is directly caused by the introduction of a change which is visible in the version control system. The bug manifestation moment refers to when a failure manifests itself for the first time, and it is not directly caused by the introduction of a change visible in the version control system, but rather, the failure is due to changes in the context or the environment. It is important to distinguish both moments because not in all cases the bug introduction moment coincides with the bug manifestation. For example, imagine that Alice inserted a line to the project that opens the html code of a website, and one week later Bob reported that the website was different from what users expected and he fixed the bug by modifying that line. Thus, there are two possible scenarios:

1. The bug introduction occurs whether the *URL* that Alice wrote is incorrect. Thus Alice introduced the error at this moment and it manifested itself in the project, although it was not notified until one week later.

[1] An E-program is written to perform some real-world activity; how it should behave is strongly linked to the environment in which it runs, and such a program needs to adapt to varying requirements and circumstances in that environment

2. The bug introduction does not exist whether the *URL* that Alice wrote is correct. Thus other reasons such as the website has been removed or suspended by the server administrator caused the bug. In this scenario there is no bug introduction moment when Alice wrote the line, but there is a first failing moment in which the bug manifested itself in the project.

However, currently there is no clear distinction between them in the software research literature on bugs. In fact, there is not a clear definition of what to introduce a bug means as it depends on the definition of a bug, which is no clearly defined either [Kim et al., 2006c]. The limited knowledge of when a error is introduced into the source code makes it difficult to distinguish between these moments, and as a consequence, researchers cannot be sure whether the line(s) identified after applying some approaches/techniques introduced the bug at the same moment of inserting the lines, or if the line manifested the bug because of other reasons. Furthermore, the lack of a meaningful model to validate these algorithms as well as the lack of definition of what needs to validated prevents researchers from calculating what a false positive or true positive is as they unsure if the line was buggy or clean at the moment of their insertion.

With the intention of better understanding the complex phenomenon of bug introduction and bug fixing, this chapter carefully introduced the proposed theory of bug introduction which explains the necessity of distinguishing how bugs are introduced and how they are manifested in software products using the concept of a test. The moment of bug introduction can be identified by using a hypothetical test that checks whether the code in the moment of its writing presents the *symptoms* described in the bug report. When the test fails, it means that the lines were buggy at this time, and we can be sure that the bug was introduced in this moment. When the test passes, it means that the lines were clean at this time and that there was not a bug introduction moment. Furthermore, we include the definitions and a taxonomy which help to analyze formally the process. The taxonomy helps to understand the many different ways in which a bug is introduced, and why some of the methods proposed in the literature might fail to find many of them. Finally, we define a model for what are *BICs*, and how they are related to *BFCs* in order to show how state-of-the-art algorithms can be evaluated in a comprehensive way, something that is missing in the current literature.

5.1 Towards a Theoretical Model

This section introduces the definition of a model to identify the changes that introduce errors. This model identifies a set of *BICs* that corresponds to a set of *BFCs*. It includes precise definitions of *BFC* and *BIC* based on the assumption that there is a hypothetical test with the perfect test coverage that could be run indefinitely across the history of the source code. This model returns as true or false depending on whether or not the bug was present at a given specific moment.

The main aim of describing this model is to extend the current state-of-the-art approaches in order to ensure that the commits identified as *BICs* introduced the bug into the source code at some point in the project history. Also, this model can be used as a framework to evaluate in a comprehensive way the performance of other approaches as well as to compare the effectiveness between different algorithms since it defines the *"gold standard"* of which commits in a project are *BICs*. Before explaining in detail the theory of the model, we should describe some important concepts used in the model that help to better understand it.

5.1.1 Definitions

Currently, there are no formal descriptions of Version Control Systems based software development process, even though some authors have attempted this before [Rosso et al., 2016, Brun et al., 2013]. These articles do not cover all the elements or set of elements that are required to describe a VCS-based software development process. The projects analyzed in this thesis use git as their VCS which records observable changes to a file or set of files. Observable changes are alterations of the file(s) caused by additions, deletions or modifications. This thesis is only focused on observable changes in the lines of source code of the project; thus, changes are preceded by other changes making up a linear vision of precedence. This precedence is not set by dates, but by previous versions (changes) in the VCS.

Looking for the origin of bug is a complex task. In this way, we found the necessity of formulating a terminology which is one of the valuables parts in this paper, this terminology can be applied to the version control systems. The terminology defines every element and every set of elements that take place during the analysis from the fixed code to the identification of the *BIC* or the *FFM*. To avoid confusion, next we define the concepts we are going to

work with:

Atomic Change (at): An operation that applies a set of changes (modification, deletion, addition) as a single operation. In this thesis, *atomic change* is assumed to be one line minimum change.

Previous Atomic Change (at'): Given an atomic change at, we refer to at' as the last modification which changed the line l of a file f. Thus, the precedence relation between an atomic change and its previous atomic change is as follow:

$$at' \rightarrow at$$

Commit (c): An observable change that records one or more atomic changes to the source code of a software system. These changes are generally summarized in a patch which is a set of lines that a developer adds, modifies or deletes in the source code. Commits update the current version of the tree directory.

Lines Changed (LC): By definition, a commit may change zero[2] or more lines of code; we call these *lines changed* of a commit and denote them as $LC(c)$.

Precedence between commits: Relation between the *atomic changes* of a commit with their *previous atomic changes* in the file f. Given a commit c and a line l, a commit c' is called the previous commit of c if it last touched the line and l of c. We will refer to this precedence between commits as the *previous commit (*pc*)* of a commit in f and denote it as:

$$pc'(c) \rightarrow pc(c)$$

Previous Commit Set (PCS): Set that includes the different previous commits of a commit, we refer to it as $PCS(c)$. Formally:

$$PCS(c) = \bigcup pc'(c)$$

[2]When only new lines are added in a commit, zero lines are changed.

Descendant Commit (dc): Given a commit c and a file f and the lines changed LC, a descendant commit of c is any of the commits that belongs to the precedence commit chain of the $LC(c)$ in f, we will refer it as dc.

Descendant Commit Set (DCS): Set of descendant commits for a given commit; we refer to it as $DCS(c)$. Note that the *previous commit set* contains only the previous commit to a commit, whereas the *descendant commit set* contains all the commits that have modified, in somehow, the lines changed in c during all the history of f.

Ancestor Commit (ac): Given a commit c a commit ac is called the ancestor commit of c if its precedence is previous to c.

Ancestor Commit Set (ACS): Set of the ancestor commits of a given commit; we refer to it as $ACS(c)$. Note that from a specific commit of the repository, the *ancestor commit set* contains all the commits of that repository.

Immediately Ancestor Commit (iac): Is the commit immediately before to a given commit in the ancestor commit set, we will refer it as iac.

Snapshot: It represents the entire state of the project at some point in the history. Using git as example, given a commit c, the corresponding snapshot would be the state of the repository after typing "git checkout c". The evolution of the software can be understood as a sequence of snapshots, each corresponding to a commit, in the order shown by "git log" (order of commits in the considered branch).

Bug: It refers to a software component malfunctioning. In the literature, these causes are also referred to as "defects", or "errors".

Bug-Fixing Commit (*BFC*): Commit where a bug is fixed. As a fixed bug b might require one or more commits to be fixed, we define the set of Fixing Commits (*BFC*) of a bug b as following set: $BFC(b)$. In general, we expect this set to be a singleton, i.e., a bug is fixed in a single commit, although several commits may be needed to fix a bug. Furthermore, a commit

fixing some bug only exists whether it is really a bug at the moment of fixing, because to find out which commit introduced the bug, and it is necessary that it really being a bug.

Bug-Fixing Snapshot (*BFS*): snapshot of the code corresponding to the *BFC*.

Test Signaling a Bug (*TSB*): A test used to signal that a bug is present. It is defined as an hypothetical test, that could be run on any snapshot of the code, returning *True* whether the test is passed, meaning that the snapshot does not contain the bug. And *False* whether the test is not passed, meaning that the snapshot contains the bug. The test is known to pass in the

Test failing snapshot *(T-S)*: snapshot for which *TSB* fails.

Test passing snapshot *(T+S)*: snapshot for which *TSB* passes.

Bug-introducing snapshot (*BIS*): First snapshot in the longest continuous sequence of T-S, which finishes right before the *BFS*. That is, there is a continuous sequence of snapshots for which the test fails, starting in the *BIS*, and finishing right before the *BFS*. Since the test is failing all the way from this snapshot up to the fix, we can know that the test was failing all the way in that sequence, and since this is the first snapshot with the test failing, we can say that this is the first snapshot "with the bug present".

Bug-introducing commit (*BIC*): A specific commit corresponding to the *BIS* that introduced the buggy line(s) at the moment of their insertion, and the bug propagated through each following commit until the *BFC* fixed the line(s).

First Failing Moment (*FFM*): The first commit corresponding to the *BIS* that manifest the bug but it did not introduce buggy line(s) at the moment of their insertion.

Discovery Moment (*DM*: The moment when a developer or user finds a bug and reports it in the issue tracking system.

Committer: The person who has the rights to commit to the source code of a particular piece of open source software. The committer might not be the original author of the source code. The author is someone who writes the original code, whereas the committer commits the code on behalf of the original author. This is important in Git[3] it allows the possibility of rewriting history or apply patches on behalf of another person.

It is to be noted that some of these terms have been used for different concepts in the literature; and from it, it can be understood why we argue that a common terminology when investigating bug fixing activity is needed. Table 5.1 offers a comparison of the terminology proposed in this paper and how these concepts have been referred in previous works through a diverse terminology. To our knowledge, no previous studies has presented a comprehensive list of all concepts needed to have a clear and complete vision of the bug introduction problem.

5.1.2 Explanation of the Model

All too often when analyzing projects, researchers use `git` as their *Version Control System (VCS)*. The VCS records *observable changes* to a file or set of files. Observable changes are alterations of the file(s) caused by additions, deletions or modifications of one or more lines in the source code. Thanks to the VCS, researchers are able to manually or automatically track back deletion and modification of lines from a specific moment up until its origin, they are also able to identify which lines are new additions in each commit. Navigating back into the relationship between the altered lines of each observable change and its previous one, a *"precedence observable change tree"* or *"genealogy tree"* can be built. Figure 5.1 shows an example of the commit i made in a file f that fixed the bug b. Backtracking each modified or removed line from f, we can draw the *genealogy tree* of the changed lines in i. It is important to notice that the new addition in i cannot be tracked, but they still remain in the model. The black boxes represent the different commits, the dots represent hunks of only new lines, the arrows show the precedence between commits, and the color of the lines depend on whether they are removed (red), added (green) or modified (black).

Using the defined terminology, we will refer to the commit i as the *BFC*. From the lines changed in the *BFC*, $LC(\textit{BFC})$, we might draw its genealogy tree in which, thanks to the

[3]To further information see https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

Table 5.1: Comparison of our proposed terminology with previous terms found in the bug introducing literature.

Terminology	**Found as ...**	**References**
Commit	Change	[da Costa et al., 2016][Kim et al., 2006c]
	Commit	[Izquierdo et al., 2011]
	Revision	[Kim et al., 2008]
	Transaction	[Śliwerski et al., 2005b][Bettenburg and Hassan, 2013]
Prev. Commit	Earlier change	[Śliwerski et al., 2005b]
	Change prior	[Williams and Spacco, 2008]
	Ancestor commit	[Blondeau et al., 2017]
	Prev. commit	[Izquierdo et al., 2011]
	Recent version	[Kim et al., 2008]
	Preceding version	[Hata et al., 2010]
Ancestor Commit	Revision	[Śliwerski et al., 2005b][da Costa et al., 2016], [Kim et al., 2006c]
	Changes	[Kim et al., 2008]
	Ancestor	[Bird et al., 2009b]
BFC	Fix for a bug	[Śliwerski et al., 2005b]
	Bug-fixing change	[Izquierdo et al., 2011][Williams and Spacco, 2008], [Kim et al., 2008][da Costa et al., 2016][Kim et al., 2006c]
	Fixed revision	[Hata et al., 2010]
BIC	Fix-inducing change	[Śliwerski et al., 2005b][Williams and Spacco, 2008]
	Bug-introducing change	[da Costa et al., 2016][Kim et al., 2006c][Kim et al., 2008]

precedence between commits, there is a genealogical relationship. Visually from this relationship, we distinguish between commits of the first generation *(i-1a,i-1b,i-1c)*, second generation *(i-2a, i-2b, i-2c, i-2d)*, and third generation *(i-3a)* of the *BFC*. By extension, the Previous Commit Set of the *BFC*, are the first generation commits and the Descendant Commit Set are the first, second and third generation of commits.

$$PCS(\textbf{\textit{BFC}}) = (i-1a, i-1b, i-1c)$$

$$DCS(\textbf{\textit{BFC}}) = (i-1a, i-1b, i-1c), (i-2a, i-2b, i-2c, i-2d), (i-3a)$$

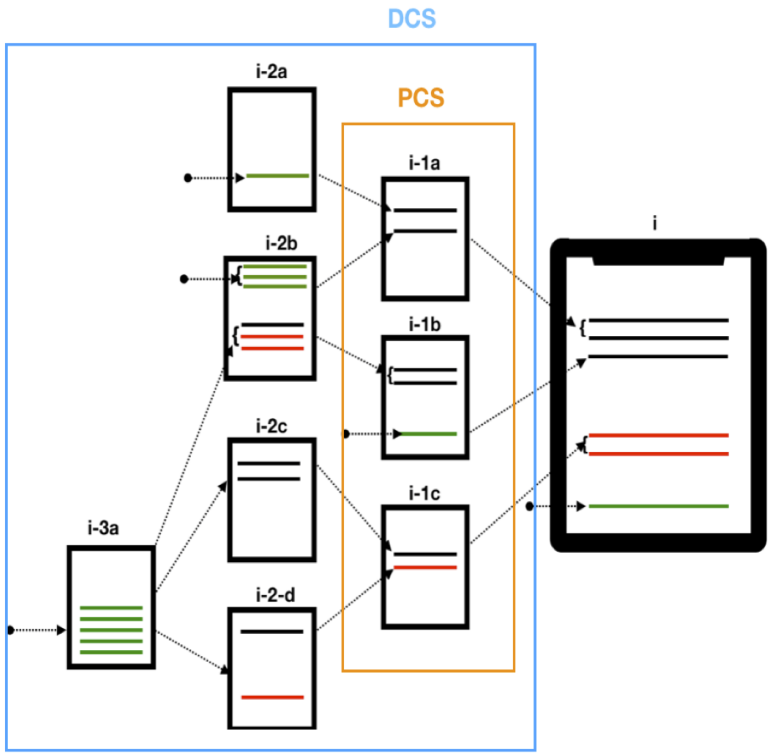


Figure 5.1: Genealogy tree of the commit i, each commit shows a precedence relationship with its descendant commits.

When focusing on commits in a branch of a project repository we do not have a clear visual access to the genealogy tree of a given commit, but we have a flatten version of all the ancestor commits. In this flattened version, the commits are preceded by other changes making up a linear vision of precedence, where the commits of the genealogy can be found. An important concept is that this precedence is not set by dates, but by previous versions in the VCS. This occurs due to the way in which a decentralized VCS system works. Bird

et al. explained how the local repositories of two collaborating developers working with git might diverge, which causes that each repository to contain new commits that are not present in the other. Thus, in the moment of combining both local repositories, the user can select between many options regarding the sequence of commits such as to rebase, merge, remove, squash, etc. These actions may alter the natural order of commits, which inhibits them from being sorted in time [Bird et al., 2009b]. Continuing with the above example, Figure 5.2 demonstrates the linear vision of precedence of the change i in the *master* branch of a project repository. The commits are represented by circles, and the changes belonging to the genealogy tree of i can be found in orange, based on whether they are a pc in blue, or whether they are a dc; the remaining commits are the ac where the commit before a is is the immediately ancestor commit iac. The $ACS(i)$ were committed to the project; however, they do not have a precedence relationship with the lines modified in i.

For those who are familiarized with `git`, we can compare the Figure 5.1 with the `git blame` of the modified lines in a *BFC* and Figure 5.2 with the `git log` of a given commit in the master branch of a project repository.

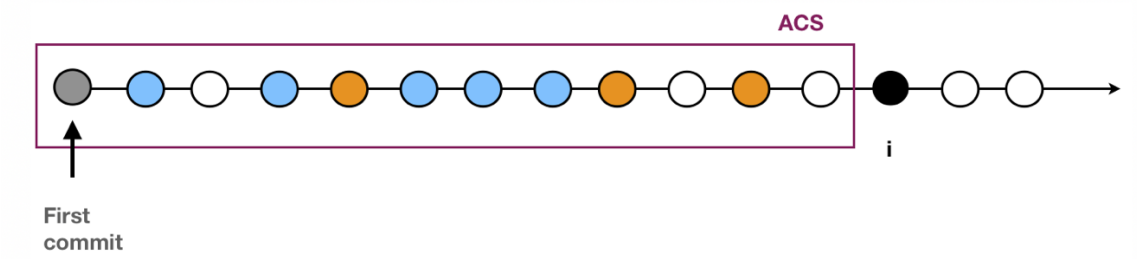


Figure 5.2: Linear vision precedence in the master branch of the bug-fixing change i. The colored commits belongs to the $PCS(i)$ (orange) and $DCS(i)$ (blue), the black commit is the *BFC* and the gray commit is the initial commit of the project. Notice that the commits are not sort in time because we are not assuming a precedence set by dates.

From an objective point of view, it is not important *WHEN* the bug was introduced in time but, *WHAT* commit introduced it. This is because after inserting the erroneous lines in an ancestor commit of a *BFC*, the bug is in the system and furthermore, it may propagate through each new change in the file. Hence, determining the *FFM* that manifests the bug implies to navigate back into the *ACS*. Thus, from a theoretical point of view, there will be one commit in the linear vision precedence that manifest the malfunction the first time.

This commit will be the *FFM*. Furthermore, the *FFM* may be the *BIC* based on whether it introduced a change visible in the VCS of the project that caused the bug. When there is no change that introduced the error the *BIC* cannot be found in the $ACS(BFC)$. As a result, the *FFM* is used to explain that in this precise commit, other (external/internal) changes that do not belong to the $ACS(BFC)$, affected in somehow the project and caused the failure of the system.

5.2 How to Find the Bug-Introducing Commit and the First-Failing Moment:

In order to discover the *BIC* with the maximum accuracy, it is recommended to do it manually by backtracking each line of the source code of all changes of a *BFC* until find the *moment* where the bug was introduced the first time. To ensure that in this moment the bug was introduced, it may be necessary to use information from the code review system and version control system. If according to this information, the commit introduced the error, the change is regarded as *BIC*. On the contrary, if the information gathered from these systems explains that there was a change in the *environment or context* that caused the failure, it is not a *BIC*, and in this case the change is the *FFM*.

Theoretically and under optimal conditions, this process can be fully automatic relying on the Test Signaling a Bug (*TSB*) which flags as *True* when the test is passed and *False* when the test fails in the analyzed snapshots of the project. Despite the high complexity of automating, there is an easy way to find the *BIC* or *FFM* in this model, which can be achieved through looking manually for the first snapshot in the linear vision precedence when the *TSB* fails.

This model focuses on the cases when a *BIC* for a *BFC* can be found or the cases when it is sure that a *BIC* for a *BFC* does not exist. To simplify, we assume that there is only the master branch in the repository of a project. To show how the model works, the definitions of *BIC* and *BFC* based on the existence of a *TSB* are applied. The *TSB* is applied in all the snapshots of the linear vision precedence of a *BFC* to identify whether there is a *BIC*. Considering that the *TSB* has a coverage of 100% and that it can run indefinitely across the history of the source code, the proposed model is able to find out the *BIC* or the *FFM* of a bug report by analyzing the changes that fixed the bug. This *TSB* will be passed into all the

ancestor commit set of the —BFC. Thus, when the *TSB* test is passed to all the snapshots, it looks for the snapshot that fails; if found, the model will consider it as a candidate for the *BIC* and *FFM*.

This technique differs from previous techniques used in software testing to locate faults as it looks for the origin of the bug by understanding the different origins of bugs. The software testing techniques do not look at the origins of the bugs by understanding their reasons and their dependencies that may cause the problem that a clean line can become to be buggy in the future, but they usually are focused on minimize the cost that a fault may cause in the system by guiding developers to the faulty location. Our proposed technique attempts to rebuild the complete history of the system in each snapshot in order to know what dependencies are being used and whether the bug was inserted at at that moment. Thus, to our knowledge, this is the first time that this idea is presented, where the software testing can be used to find the *BIC* and the *FFM* using the *TSB*.

5.2.1 Outcome of the Test Signaling a Bug

As mention earlier, the *TSB* checks for the functionalities and features of a project with a test coverage of the 100%. Currently, a common practice in the bug fixing process is to add a test case checking the behavior of the fixed bug when submitting the *BFC*. Thus, the proposed model may use this information from the test case to build the *TSB*.

The outcome of the *TSB* varies depending on each snapshot, there are three different outcomes:

1. **Pass**: The function or feature tested is present in the snapshot and it works as the test anticipated according to the *BFC*. The snapshot does not present a *BIC*.

2. **Fail**: The function or feature tested is present in the snapshot but it does not work as the test anticipated according to the BFC. This snapshot is considered as candidate to be the *BIC* or *FFM*.

3. **Not-Runnable**: The function or feature tested is not present in the snapshot, thereby the test cannot run.

There are four different scenarios to illustrate how to apply the hypothetical test into the

linear vision precedence of the $BFC(i)$ in order to identify whether the *BIC* exists. In these scenarios it is considered that the *TSB* have 100% of coverage and that it can run indefinitely across the history of the source code. Thus, the *TSB* is passed to all the snapshots (ancestral commit set) in search for the one that fails; if found, it will be consider it as a candidate for the *BIC* or *FFM*.

Figure 5.3 shows how to apply the hypothetical test to the linear vision precedence when there is a *BIC* and the *TSB* can be run in all the snapshots. To locate the *BIC*, the *TSB* is passed into all the snapshots of the *ACS*. The *BIS* is the first that fails. It can be known that the *BIS* is the *BIC* because the snapshot before passes with the same environment that the *BFS*.

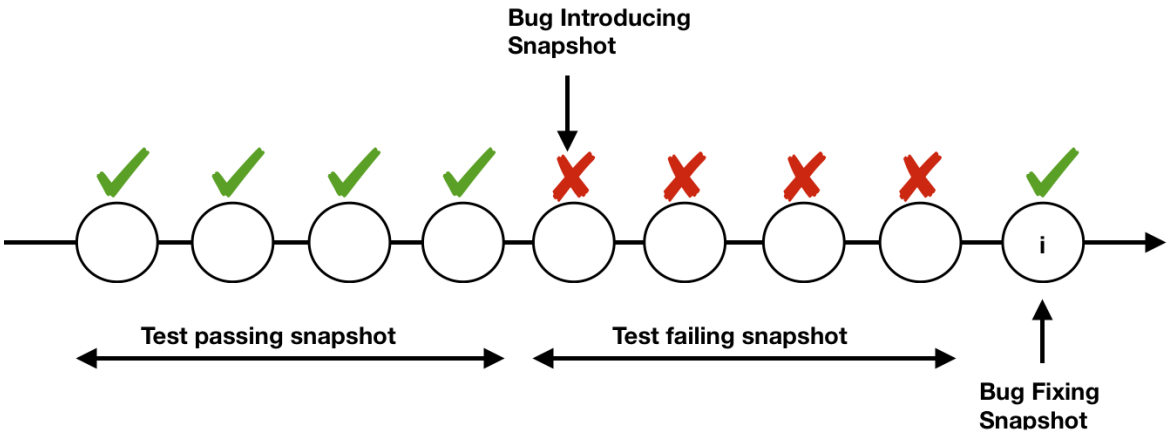


Figure 5.3: The Bug Introducing Snapshot is the Bug-Introducing Commit

Figure 5.4 shows how the hypothetical test is applied to the linear vision precedence when there is a *BIC* but the *TSB* test cannot be run after a snapshot. This is because the tested function or feature is not present in that moment. Here, the first *BIS* identified is also the *BIC* because when it introduced the function or feature tested, without any other external changes, it was buggy.

Figure 5.5 shows how the hypothetical test is applied to the linear vision precedence when there is not a change in the *ACS* that introduced the bug and the *TSB* can run indefinitely across the VCS of the project. The *BFC* fixed a bug caused by a change in the environment or a external change, thus the *TSB* passes in the *BFS* but not in older snapshots. However, when the *TSB* is run in the antecesor snapshots replicating the “new” environment or the

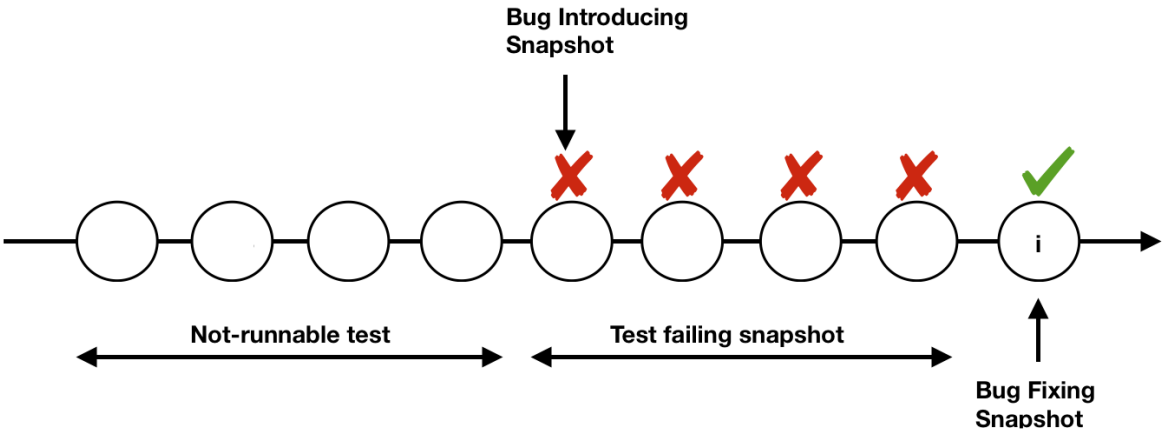


Figure 5.4: The Bug Introducing Snapshot is the Bug-Introducing Commit

external change, it passes in all the posterior snapshots to the external change, and it fails in the ancestor snapshots. In this scenario the first *BIS* before the *BFC* will be the *FFM*, there was not *BIC* in the *ACS*.

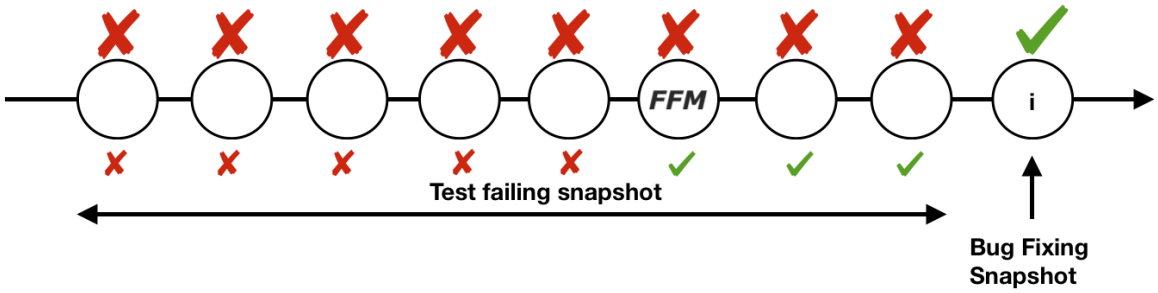


Figure 5.5: The Bug Introducing Snapshot is the the First-Failing Moment

Figure 5.6 shows how the hypothetical test is applied to the linear vision precedence when the *BIC* was in the project from the beginning and the *TSB* can run indefinitely across the VCS of the project. The *TSB* is passed into all the snapshots of the *ACS* but it will alway fail. The *BIC* is the first snapshot in the *ACS*. Our hypothesis is that this scenario is not common.

5.2.2 Criterion to apply to the Test Signaling a Bug

As mention earlier, when the history of a project is linear, researchers can test the $ACS(i)$ one by one and run the *TSB*. On the opposite, when the history of the project is not linear, it may have multiple branches to be tested. Furthermore, as the failure might be present in

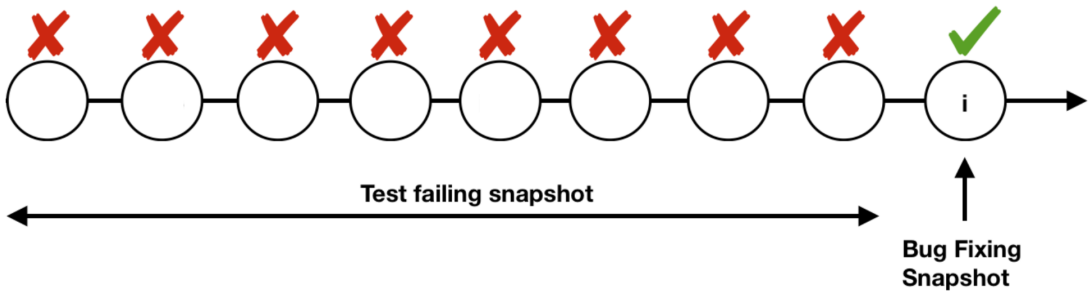


Figure 5.6: The first snapshot is the Bug-Introducing Commit

some simultaneously existing branches and absent from other branches the concept of the *FFM* may be uncertain. However, at first glance, and assuming that theoretically the test can be run for all snapshots in the $ACS(i)$, the outcome of the *TSB* still find sections in one or more branches where the test would fail. Thus, the model is still able to find the suspicious snapshots to be the *BIC* or *FFM*.

It may be possible that in some scenarios, the proposed model should define the *BIC* as the one which is topological "the first one" in the faction of continuously failing snapshots. It also may occur that, the presence of multiple parallel branches cause two or more commits in parallel start to fail until the *BFC*. However, it may be an indication that the bug was introduced into the source code, independently, in several branches or it was copied (or written equally by chance) in several branches. Our hypothesis is that these scenarios are not common, however, we need to include them in the theoretical model and verify this hypothesis in the future work.

Thus, to accommodate this case, we could extend the notion of *BIC* to "the set of *BIC*", which would be those "commits corresponding to the first snapshot to fail, continuously until the *BFC*, in several branches that lead to the *BFC*".

Furthermore, depending how the *TSB* fails there may be different situations, the researchers should decide whether the *BIS* is a *BIC*, a *FFM*, both, or undecided.

1. **Undecidable**: When we cannot be sure to find the *FFM* or the *BIC*. In this class, we may not find the *FFM* between all the $ACS(b)$ using the hypothetical test, for example due to the lack of information.

2. **The *BIC* is the *FFM***: When we can find the *BIC* between all the $ACS(i)$ using the

hypothetical test and furthermore, it is the *FFM*.

3. **There is only *FFM***: When the bug was not caused by a change belonging to the $ACS(i)$ but, another external factors cause the failure such as: (1) There is a change in the dependencies of the project; (2) There is a change in the requirements of the project; or (3) There is a bug in APIs used in the project. In this class, we will find the *FFM* using the hypothetical test and it will be the first time that the malfunction manifests itself in the project.

Nevertheless, it would appear that automatizing this process is tedious and complex, because all the external dependencies used in the project makes it more complicated to build an isolated test to run in each previous change. However, we still rely this can be possible thanks to studies such as the one performed by Bowes *et al.*, which provides a core list of testing principles that focuses on different quality aspects other than effectiveness or coverage. For our research, the most interesting principle is the test (in)dependency which describes that a test should be able to run in any order and in isolation. In addition, the test should not rely on other tests in anyway. Allowing practitioners to add new tests without keeping in mind dependencies or effects they might have on existing tests [Bowes et al., 2017].

5.3 Algorithm to Find the BIC and the FFC

Figure 5.7 presents the decision tree in order find the *BIC*, whether it exists, and the *FFM*. This decision tree is implemented in our model and includes the current shortcomings in the state-of-the-art approaches based on backtracking the lines that have been modified in the *BFC* in order to find the *BIC*.

1. **Lack of guidelines when SZZ identifies more than one pc**: The decision tree can distinguish when there are more than one previous commit and what needs to be done to identify the *BIC* or the *FFM*.

2. **Only new added lines in the BFC**: The decision tree can distinguish when there are only new lines added in the *BFC*. In this case, the PC Set is computed by identifying the pc of the lines surrounding the new additions.

3. **Modifications are not related to the root cause of the bug**: The *TSB* does not test the functions or features that are not related to the root cause of the bug.

4. **More than one issue are addressed by the BFC**: The *TSB* only implement the test for the functions or features that caused the bug.

5. **Lines correct at the time of their insertion**: The *TSB* always fails with the BFS environment in the snapshots, identifying the *FFM*.

6. **Dormant Bugs**: The *TSB* will identify the first snapshot in which the test fails, identifying the *BIC*.

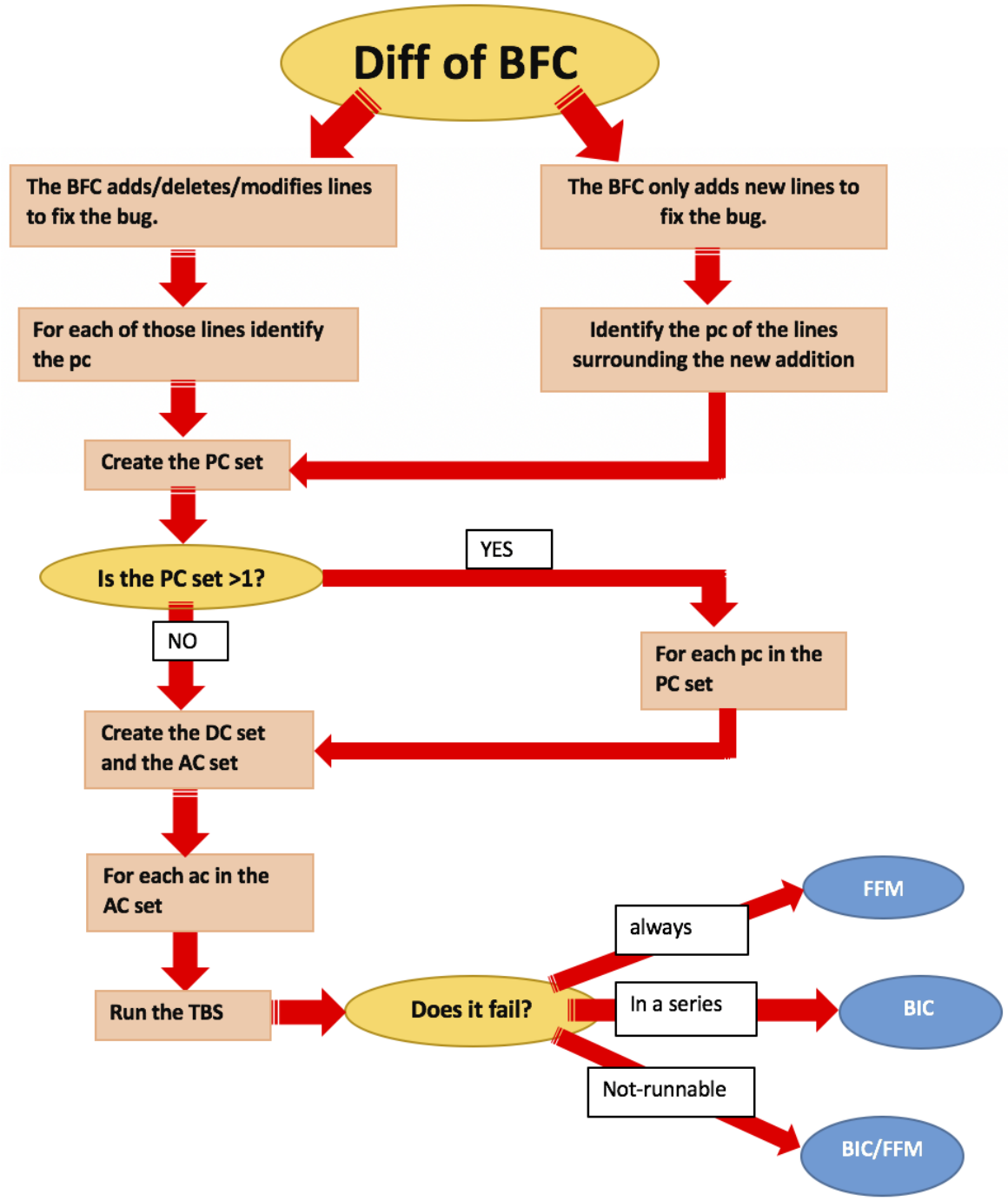


Figure 5.7: Decision tree to identify the BIC and the FFC

Chapter 6

Empirical Study on the Applicability of the Theory

This chapter presents an empirical study on the applicability of the theoretical model that was presented before in Chapter 5. The empirical study was carried out through analyzing the public data sources from two OSS projects, Nova and ElasticSearch. This study can be extended to others projects as long as they use VCS and bug-tracking systems. The goal of this Chapter is the application of the proposed model, that defines the code change that introduced a bug in the system or determines whether it exists given a Bug-Fixing Commit (*BFC*). Furthermore, this model enables to define the "gold standard" to be defined where commits in the repositories are Bug-Introducing Commits (*BICs*).

This chapter is divided into three sections. The first part is a brief introduction to the projects and arguments its election as case studies to identify the origin of bugs.

The second part clearly defines the methodology to manually analyze and identify the *BIC* given a *BFC*. This methodology explains how the raw dataset has been filtered and details in stages the approach to identify the *BIC* given a specific *BFC*.

Lastly, the third part presents the results after applying the theory into Nova and ElasticSearch. The obtained results give more insight into the problem of correctly identifying the *BIC* by comparing the "gold standard", with the performance of some state-of-the-art approaches. It computes the *real* false positives and false negatives.

Table 6.1: Main parameters of the Nova project, June 2018.

Parameter	Size
Companies Contributing	216
Commits	32474
Contributors	1602
LOCs	4581836
Resolved Bugs	1930

6.1 Case studies and datasets

There are two case studies that are explained in subsections 6.1.1 and 6.1.2. Both projects have similar characteristics that are interesting and worthwhile to study, however, they also have some differences that may enable the results to be extended to other open source projects.

6.1.1 Nova

Nova is the most active module of OpenStack project in terms of contributions. OpenStack is a cloud computing platform that is used to build a SaaS (software as a service) with a huge development community thereby making it an interesting project to study. OpenStack has a considerable growth of approximately 350 times in code size and 250 times in number of commits since its first release in November 2009. It counts with more than 7,900 contributors, and significant industrial support from several major companies such as Red Hat, Huawei, IBM, HP, SUSE, etc. OpenStack is mainly written in Python. Currently it has more than 328,800 commits with more than 47 million lines of code[1]. All its history is saved and available in a version control system[2], an issue tracking system (Launchpad[3]) and a source code review system (Gerrit[4]). Table 6.1 provides a summary of the Nova project. There are 1,602 distinct authors identified in the project from more than 200 different companies.

In addition to the enormous diversity of people and companies contributing to Nova, the project has other characteristic that make it a good case of study.

- **Research friendly**: Both the bug tracking system and the version control system (*VCS*)

[1] `http://stackalytics.com`
[2] `https://wiki.openstack.org/wiki/Getting_The_Code`
[3] `https://launchpad.net/openstack`
[4] `https://review.openstack.org/`

are a trustful data source for retrieving bug-fixing information. In addition, the policy of adding the link of the Bug-Fixing Commit into the bug report information is helpful when linking and analyzing the two data sources.

- **The openness of dataset**: all of the data analyzed in this chapter and through the thesis are publicly available. This supports the replicability of the experiments which is a fundamental part of Empirical Software Engineering.

- **linearVCS**: OpenStack policy tries to maintain a linear history This makes analyzing Nova efficient since the cases where a bug might be present in some simultaneously existing branches can be reduced.

- **Programming Language**: Nova uses Python as programming language and it is an interpreted language that do not need to be compiled before running. In addition, Python is dynamically typed which means that the type for a value is decided at runtime and not in advance. These specific characteristics of Python might affect the way that bugs are introduced into the source code of a project.

Nova Dataset: Launchpad works with issue reports called tickets. These tickets describe bug reports, feature requests, maintenance issues, and even design discussions. We randomly retrieved 125 closed tickets from Launchpad that had a commit merged into the source code and were reported during 2015. However, to correctly identify the *BIC* of a *BFC*, it is necessary that a bug exists. For that reason, we only analyzed tickets that report real bugs. Thus, to ensure that the tickets described real bug reports, two different researchers were assigned to analyze them. This procedure was not a trivial task; it is very labor intensive as it has to be done manually. As the task is repetitive, we developed a web-based tool[5] described in Rodriguez *et al.* [Rodríguez-Pérez et al., 2016] that helps with the classification process. This tool offers relevant information to decide if a ticket is describing a bug report or not, this information was extracted automatically from the project repositories. As it offers a web-based interface, the tool allows for collaboration. Traceability and transparency in the identification of bug reports is also possible as all decisions are stored.

Each ticket was (manually) categorized into one of following three groups:

[5] `http://bugtracking.libresoft.info`

Table 6.2: Main parameters of the ElasticSearch project, June 2018.

Parameter	Size
Commits	39402
Contributors	1032
LOCs	1187732
Resolved Bugs	4958

1. *Bug Report*: The ticket describes a bug report.

2. *Not Bug Report*: The ticket describes a feature, an optimization/refactoring of the code, changes in test files, or any other situation... but not a bug report.

3. *Undecided*: The ticket presents a vague description and cannot be assigned without doubts to any of the previous groups.

Subsequently, the tickets agreed[6] upon by both researchers as part of the bug report were included into the final dataset. Then, the bug reports were linked to their *BFCs*. The final dataset counts up to 60 bug reports that are linked with *BFCs*.

6.1.2 ElasticSearch

ElasticSearch is a distributed Open Source search and analytics engine written in Java. It is continuously evolving since its first release in 2010 and currently counts with more than 3,900 commits. This project was chosen because it has a similar number of commits to OpenStack and its policy of labeling issues is very strict. In the project, the developers use the label "bug" when the issue describes a bug according to their opinion. Thus, researchers might rely on the developers' opinion, and they might be sure that the issues retrieved from the issue tracking system are real bug reports and as a consequence, their (*BFCs*) are fixing real bugs. The code of ElasticSearch is hosted in GitHub[7], and its issue list is available in GitHub[8] as well. Table 6.2 provides a summary of the ElasticSearch project. There are 1,000 distinct authors identified in the repository and more than 4,500 closed bug reports.

[6]Please, consider the reading of the paper for further information about the analysis and the ratio of agreement between the researchers

[7]`https://github.com/elastic/elasticsearch/`

[8]`https://github.com/elastic/elasticsearch/issues`

In addition to the enormous diversity of people and companies contributing to ElasticSearch, the project has other characteristic that makes it a good case study.

- **Research friendly**: The code and bug reports are hosted in GitHub and it is a trustful data source for bug-fixing information. In addition, the policy of adding the link of the bug report number or the pull request number into the *BFC* is helpful when linking and analyzing the two data sources.

- **The openness of dataset**: all of the data analyzed in this Chapter and in the thesis is publicly available. This helps with the replicability of the experiments which is a fundamental part of Empirical Software Engineering.

- **No linearVCS**: Contrary to OpenStack, ElasticSearch repository counts with more than 160 branches. This might make the project more complicated to analyze.

- **Programming Language**: ElasticSearch uses Java as the programming language. Java encourages error-free programming as it is strictly typed, it performs run-time checks and is independent of hardware. These specific characteristics of Java might affect the way that bugs are introduced into the source code of a project.

Dataset ElasticSearch :

ElasticSearch works with issue reports labeled as "bug" which discards the necessity of filtering those issues that are not describing real bug reports. Thus, since ElasticSearch and Nova have similar amounts of commits, we randomly retrieved the same amount of closed bug reports that are in Nova dataset, with a total of 60 bugs which were reported during 2015 and which had a commit merged into the code source. Then, these bug reports were linked to their *BFCs*, thereby the final dataset counts up to 60 real bug reports that are linked with real *BFCs*.

6.2 Methodology

The methodology used in this empirical study is divided into two parts, the first part is the filtering stage where the researchers ensure datasets fulfill the requirements for analysis under

the theory of bug introduction. While the second stage describes each of the steps from the Bug-Fixing Commit until the identification (or not) of the *BIC* and the *FFM*.

Figure 6.1 provides an overview of each step involved in our study and their outcomes.

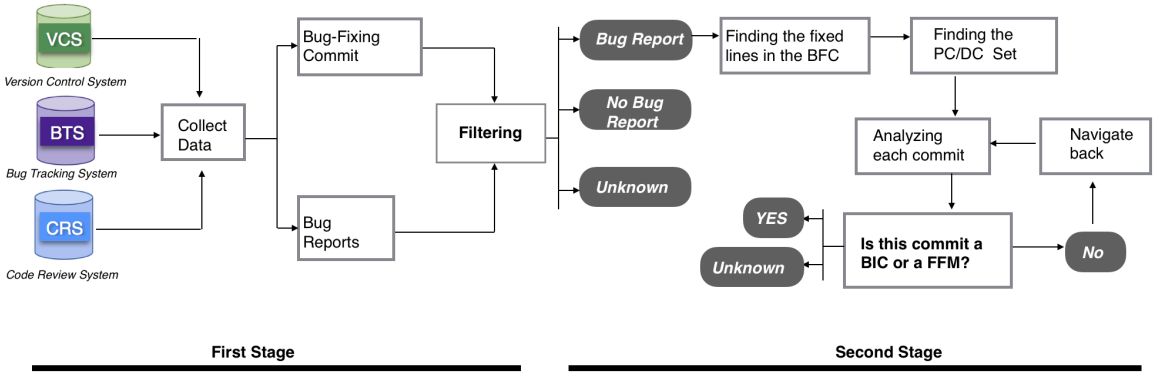


Figure 6.1: Overview of the steps involved in our analysis

6.2.1 First Stage: Filtering

This stage ensures that the bug reports stored in the datasets can be applicable to our model. Despite the policy of bug labeling in ElasticSearch and the agreement on classifying bug reports from other issues in Nova, this stage analyzed carefully each piece of information in the bug reports to ensure the "gold standard". Thus, the datasets should only store bugs that are considered bugs at the moment of bug fixing.This also ensures that if there are issues reported as bugs but are discovered to be new feature requests or improvement suggestions, they are excluded.

6.2.2 Second Stage: Identifying the First Failing Moment and the Bug Introducing Commit of a Bug-Fixing Commit

The input of this stage is a set of "real" bug reports from Nova and ElasticSearch. These bug reports describe "real" bugs in the moment of their fix. In this stage, we manually determine if a given change was the first one to show a malfunction.

Analyzing the information from the bug reports and the VCS we might identify the *BIC* and the *FFM* of a *BFC*. The identification of a *BIC* means that the bug introduction moment

coincides with a commit, i.e., a change that can be witnessed in the version control system to either source code or configuration files. In some situations, the failure is not directly caused by a change visible in the version control system, but rather, the failure is due to changes in the context or environment. The moment when the failure manifests itself is the *FFM*. In some, but not all cases the *BIC* coincides with the *FFM*.

Since the *BIC* and *FFM* can be any of the ancestor commits, it is necessary to manually analyze them in order to ensure the credibility of the "gold standard". To better understand this process, the steps are detailed in the following paragraphs. To be noticed, that some of the concepts used in the next paragraphs have been described in Section 5.1.1 that explains the proposed terminology.

Finding the lines that fixed the bug, $LC(c)$:

At this stage, the source code that fixed the bug must be identified; a generic bug is referred to as b. Therefore, it is important to:

- Identify the change(s) that fixed the bug, $BFC(b)$. In most of the analyzed cases this set was always singleton, meaning that in each bug there was a unique *BFC*. However, in the bug report #1442795[9] there were two different *BFCs* that fixed the bug. To simplify this process, the methodology assumes that the set of $BFC(b)$ is singleton, and in the case where more than one *BFC* fixed the bug, both *BFCs* should be analyzed.

- Find the lines that the $BFC(b)$ modified to fix the bug, $LC(c)$, where c is the *BFC*. In the bug report, sometimes there is a lot of information related to the code review process and the *BFC*. By applying *"git diff"* it is possible to identify what lines have been added, modified or deleted between the version after the $BFC(b)$ and the previous one. There is also the possibility to visualize these changes, between the version after the $BFC(b)$ and the previous one, using the friendly visualization that GitHub provides.

- Filter out lines that are not code: Lines that have been modified, but do not contain source code (e.g., comments or blank lines) are not considered, and as a consequence they are not analyzed. In addition, there was a *BFC*[10] that closed two different bug

[9] `https://bugs.launchpad.net/nova/+bug/1442795`

[10] `https://github.com/elastic/elasticsearch/commit/beaa9153a629c0950182e4e8c4f8eedd1c63`

reports. In this case, the lines related to the bug that was not under analysis were filtered out.

Determining, for each of those lines, $LC(c)$, what immediate previous commit last changed the lines:

Each individual line touched by the $BFC(b)$ has only one previous commit. However, there could potentially be as many previous commits as modified lines in the $BFC(b)$, (see the genealogy tree in subsection 5.1.2). Thus, taking this genealogy tree as reference, the result is a set of all previous commits of the bug b, we will refer to it as $PCS(b)$.

Analyzing each of these previous commit and their descendant commits to determine the *BIC* and the *FFM*:

This analysis uses information available in the description of the ticket as well as from the logs of the $BFC(b)$ and commits in the $PCS(b)$. After understanding the bug and the changes that fixed the bug, it is necessary to identify the *BIC*, whether it exists, as well as the *FFM*. Thus, the identification of the *BIC* starts with the analysis of the $PCS(b)$, for each of the commits belong to the $PCS(b)$, the lines changed are analyzed in order to find which one introduced the bug. At this point, there are three different scenarios and the behavior is different depending on the following conditions:

1. *The commit introduced the bug*: This commit is the *BIC* because it introduced the error at the moment of their writing. It was possible to identify the *BIC* which is also the *FFM* because it is the first commit that manifested the wrong behavior. According to the theoretical model, the *TSB* passes in the *BFS* and the *BIS* is one of the snapshot belongs to the $PCS(b)$.

2. *The commit does not insert the bug*: In this case there will be two possible outcomes:

   - The lines in the commit are correct, they do not insert the error at the time of their writing and other external factors caused the bug. There is no *BIC* in this scenario and the analysis should focuses on understanding whether this pc is first moment when the bug manifested itself. According to the theoretical model, the *TSB* is

run in all of the $ACS(b)$ and it passes in the *BFS* but it always fails in the ancestor snapshots. However, if the *TSB* is run in the ancestor snapshots replicating the change that fix the bug, it passes up to the snapshot previous to the *FFM*. Thus, the first *BIS* that passes in the sequence of *BIS* before the *BFS* is the *FFM*.

- The lines in the commit are syntactic sugar or semantically equivalent modifications (refactoring): This means that the commit retains the same behavior as before, thereby it is required to navigate back into, the $ACS(b)$ and start again in the first point of this list.

3. *It is not sure whether the commit introduced the bug*: In this case, it is important to continue navigating back into the $DCS(b)$, if this commit introduced for the first time the lines of the $LC(c)$ and it is not clear whether they are buggy, the commit is classified as "undecided". This means that after the analysis, the *BIC* was unable to be found manually despite its possible existence.

6.2.3 Outcomes

At the end of the process there are three possible outcomes for each bug report analyzed. The outcomes are related to the identification of the *BIC* and *FFM* given a *BFC*. These outcomes are explained in the following list:

- A *BFC* has a *BIC*: In this case it is sure that there is at least one *BIC* among the previous commit set, descendant commit set or ancestor commit set. However, during the manual analysis it may occur that:

  1. The *BIC* was able to be identified manually or,
  2. The *BIC* was unable to be identified manually.

  Furthermore, in this scenario, the *BIC* is also the *FFM*.

- A *BFC* does not have a *BIC*: In this case, it can be sure that any line contained the bug when it was committed into the source code, and due to other factors such as modifications in internal resources, changes in external resources, buggy code in third-parties or changes in the requirements that invalidate previous assumptions the bug

manifested itself. It can also be confirmed that none of the ancestor commits introduced the error, thereby none of them can be blamed as the cause of the bug. In this scenario, we only can identify the *FFM* and it may occur that:

1. The *FFM* was able to be identified manually or,
2. The *FFM* was unable to be identified manually.

- It is not clear whether or not a *BFC* has a *BIC*: Some bug reports do not detail enough information about the failure and the *BFC* is not sufficient in deciding whether or not there is a *BIC*. Also, some commits have changes that are very complex to understand and we cannot decide whether they introduced the bug or not.

6.3 Results

This section presents the results of the empirical analysis in order to locate the *BIC* and the *FFM* in two case studies, Nova and ElasticSearch. Furthermore, there is a comparison between the results obtained after applying the proposed model with the results obtained after applying an enhancement of the SZZ algorithm [Kim et al., 2008].

6.3.1 First Stage

During the first stage, 4 bug reports were removed from the initial set of 120 bug reports. Then, there are 57 bug reports in Nova and 59 bug reports in ElasticSearch that are considered into the next stage. The reason to removed the bug report #1185290[11] was the discordances in the comments between the developers of Nova. These are the comments where the discordance is obvious:

- "I am not sure that I consider this a bug. Without –all-tenants=1, the code operates under your own tenant. That means that –all-tenants=1 foo should really be a no-op without –all-tenants=1."
- "I disagree, mainly because the structure of the requests and code path should largely be transparent to the user. I would suggest that specifying –tenants should imply you

[11] `https://bugs.launchpad.net/nova/+bug/1185290`

are doing a query across –all-tenants=1unless the –tenants specified is the same as what is contained in OS_TENANT_NAME (the unless part is debatable)"

Furthermore, other reasons were also discovered, for instance the bug #1431571[12] reported a bug in a test file. It was removed because a bug in a test file does not mean that the source code of the project may contain a bug. It was also discovered that some bug reports such as #7740[13] describe hypothetical scenarios. In these cases, the bug report details a possible bug in the future. These bug reports were removed from the analysis because although the developers described them as bug reports, it is understood that when the *BFC* was submitted, the bug had not occurred in the project.

6.3.2 Second Stage

In this stage we identify the *BICs*, and how they are related to *BFCs* by applying the proposed model. This model enables the "gold standard" to be defined where the *BICs* in Nova and ElasticSearch caused the *BFCs*. This chapter answers two different questions:

The first part presents the results after applying the model to the datasets. Thus, to demonstrate how the model works, we gathered a set of *BFCs* from two projects, and used them to manually identify their *BICs*, when they exist. As mentioned before, the definitions of *BIC* and *BFC* were applied based on the existence of a *TSB*. In case we were unsure of the existence of the *BIC*, its corresponding *BFC* was removed from the analysis, since the empirical study is focused in the cases when a *BIC* can be found or when it do not exist for certain.

The second part presents the effectiveness of the SZZ algorithm. Once we have the "gold standard", we can compute how this algorithm, and others, "really" performs. This evaluation is an important contribution to the current literature because, as far as we know, any previous studies have carried out this empirical analysis.

6.3.3 Explanation of the Datasets

This section explains the datasets of Nova and ElasticSearch that were manually analyzed to identify the *BIC* and the *FFM*.

[12] `https://bugs.launchpad.net/nova/+bug/1431571`

[13] `https://github.com/elastic/elasticsearch/issues/7740`

The datasets of Nova and ElasticSearch contain 57 bug reports linked to 59 *BFC*. In Nova there are 9 *BFCs* with only new added lines to fix the bug, almost the same number that ElasticSearch which has 8 *BFCs* with only new additions. The total number of previous commits identified in Nova was 124 and, 112 in ElasticSearch.

Figure 6.2 presents the average of previous commits, files and test files in each bug report of Nova and ElasticSearch. On average, both projects are similar around 2 previous commits per *BFC*, although it is slightly higher in Nova with 2.17 previous commit per *BFC*. However, the average of files touched by the *BFC* is higher in ElasticSearch, with 3.18 files[14] per *BFC*. The average of test files modified in a *BFC* is almost the same in both projects with 1.53 in Nova and 1.60 in ElasticSearch.

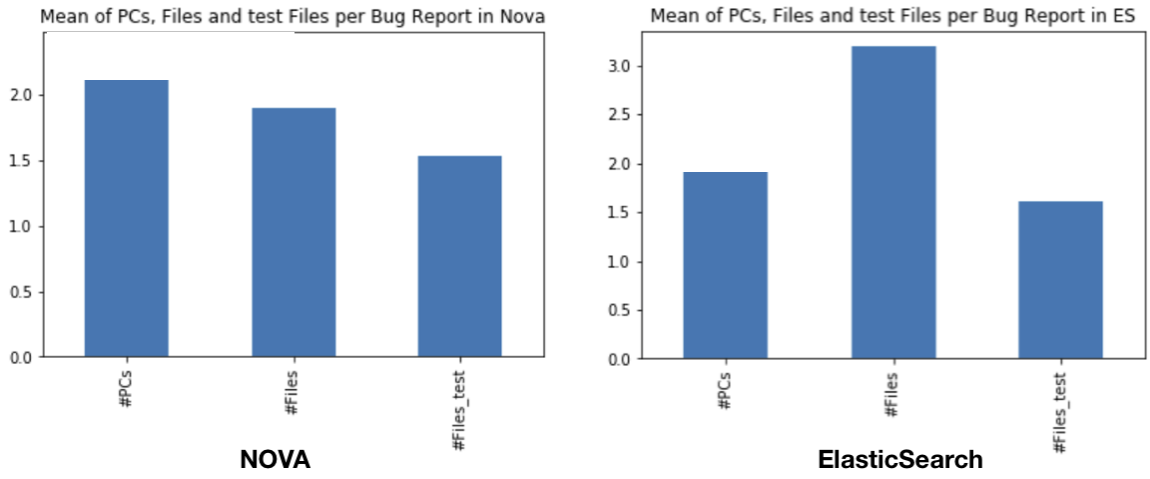


Figure 6.2: Mean of previous commits, files and test files per Bug Report in Nova and ElasticSearch

6.3.4 RQ1: What is the frequency for a Bug-Fixing Commit being caused by a Bug-Introducing Commit?

Table 6.3 shows the number of *BFCs* that have been caused by *BICs* and the number of *BFCs* that have not been caused by a *BICs* in the datasets. The trend in the results of both projects is similar as both present higher percentage of *BFCs* caused by a *BIC*. However, the percentage of *BFCs* that were not caused by a *BICs* is more representative in Nova with 21% of the *BFCs*

[14] the test files are not included here

Table 6.3: Percentage of Bug-Fixing Commit (*BFC*) with a Bug-Introducing commit (*BIC*), without a Bug-Introducing Commit (*NO_BIC*) and Undecided in Nova and ElasticSearch (*ES*).

	BIC	NO_BIC
Nova	45 (79%)	12 (21%)
ES	54 (91%)	5 (9%)

Table 6.4: Reasons why a Bug-Fixing Commit is not caused by a Bug-Introducing Commit

	Nova	ElasticSearch
Co-evolution Internal	5 (42%)	2 (40%)
Co-evolution External	2 (17%)	1 (20%)
Compatibility	1 (8%)	0 (0%)
Bug in External API	4 (33%)	2 (40%)

being fixing a bug that was not introduced in the system. On the contrary, this percentage is lower in ElasticSearch, where only the 9% of the *BFCs* were not caused by a *BIC*.

RQ1.1: Which reasons may explain that a Bug-Fixing Commit is not caused by a Bug-Introducing Commit?

Additionally, in those cases where the *BFC* does not present a *BIC*, we have looked for its root cause. Table 6.4 shows the main reasons for a bug not having a *BIC*. Furthermore, they are explained below:

- *Co-evolution Internal*: Changes in the source code related to satisfy the new requirements of the project. Since internal resources have been modified or requirements have been changed invalidating previous assumptions. The bugs caused by the co-evolution internal can be explained from the point of view that the code is not reflecting a new change or requirement of the project and the source code is manifesting the failure.

- *Co-evolution External*: A bug is caused by some external change(s). External resources

have been modified without a previous notification and the source code of the project starts to fail. The bugs caused by the co-evolution external are easiest to understand than the previous ones, because they can be explained from the point of view that a change/update in the source code of an external artifact used in the project has caused the bug.

- *Compatibility*: Bugs are caused by an incompatibility between software and hardware or an incompatibility with some operating systems. We consider that these bugs do not have a *BIC* because the failure is not present always, it manifests itself depending on compatibility factors such as the hardware used.

- *Bug in External API*: A bug in the API of a external library to the project has caused a bug in the source code of the project. In this scenario the *BIC* exists but it cannot be identified in the *ACSBFC* because this change is not in the VCS of the project.

These reasons explain why the *BIC* cannot be identified given a *BFC*. In Nova the principal reason with a 42% occurrence rate was the *Co-evolution* of the lines changed to fix the bug with the internal requirements of the projects. The second reason with a 33% of the cases where the presence of bugs in external APIs that are used in Nova. Then, the third most frequent reason is the *Co-evolution* of the lines changed to fix the bug that is related to third-parties code or external artefacts. Finally, the *Compatibility* reasons explain why in 8% of the cases the *BFC* was not induced by a *BIC*. On the other hand, in ElasticSearch the percentages are equally distributed with a 40% occurrence rate in *Co-evolution internal* and *Bug in External API*, and 20% of the cases belongs ti *Co-evolution External* with the exception that bug caused by the incompatibility of hardware and software were not found.

RQ1.2: Could the location of a bug be modeled on the Bug-Introducing Commit and First-Failing Moment?

Table 6.5 shows the percentage of *BIC* and *FFM* that have been manually identified by using the model. By definition, and in theory, our model is able to identify the *FFM* and the *BIC*, and whether it exists, but in practice, since the identification needs to be done manually, sometimes the *BIC* or the *FFM* cannot be identified for certain. To be noticed, these *BFC* that were classified into "Undecided" in Table 6.3 are not considered in Table 6.5 since researcher

Table 6.5: Percentage of Bug-Introducing Commit and First Failing Moments identified after applying the theoretical model.

	BIC = *FFM*	only *FFM*	Undecided
Nova	34 (60%)	12 (21%)	11 (19%)
ES	38 (64%)	5 (9%)	16 (27%)

were not sure whether they have a *BIC* or not. Thus in Nova, from the 45 bugs caused by a *BIC*, we correctly identified 34 *BICs* which are also *FFMs*. From the 12 bugs manifested without a *BIC*, we successfully identified 4 *FFMs*. In the dataset of ElasticSearch, from the 54 bugs caused by a *BIC*, we correctly identified 38 *BICs* which are also *FFMs*. Form the 5 bugs manifested without a *BIC*, we were unable to identify manually the *FFMs*. Due to the complexity to identify and ensure correctly the *BICs* and *FFMs*, both projects have the "Undecided" category, which implies that we have suspicious commits to be the *BIC* and *FFM* but we cannot be completely sure because of their complexity or lack of information.

RQ1: 79%-91% of the Bug-Fixing Commits analyzed were caused by a Bug-Introducing Commit,*BICs*. However, 9%-21% of the bugs were caused by other factors different from introducing faulty code in the source code. The main reason why a bug is not caused by a *BIC* in both projects is the co-evolution internal.

6.3.5 RQ2: What are the specifications that define the effectiveness of an algorithm used to locate the origin of a bug?

After applying the model and identifying the number of *BICs* and *FFMs* in our datasets, we obtained the "gold standard" where the number of *BFCs* with a *BIC* and without a *BIC* can be known for certain. For that reason, we are able to compute the true/false positives and true/false negatives for an algorithm, and for that purpose the original SZZ algorithm [Śliwerski et al., 2005b] was selected, but we removed the changes in the comments and blank lines from the analysis when using this algorithm. Also, we analyze the SZZ-1 [Kim et al., 2006c].

The SZZ algorithm does not compute *BFC* with only new lines added, and in the "gold

Table 6.6: Results of True Positives, True Negatives, False Negatives, Recall and Precision for the SZZ and SZZ-1 algorithms assuming that the algorithm flags all of the commits belong to a set of $PCS(b)$ as *BIC*.

	True Positives	False Positives	False Negatives	Precision	Recall	F1-Score
Nova (SZZ)	25 (26%)	54 (56%)	17 (18%)	0.32	0.60	0.42
Nova (SZZ-1)	28 (30%)	51 (55%)	14 (15%)	0.35	0.58	0.44
ES (SZZ)	26 (27%)	59 (61%)	12 (12%)	0.31	0.68	0.43
ES (SZZ-1)	27 (28%)	58 (60%)	11 (12%)	0.32	0.71	0.44

standard" of Nova there are three *BFC* with only new lines whereas in ElasticSearch there are seven. These cases are the false negatives of SZZ. Furthermore, the average of previous commits per *BFC* is around two, this means that the SZZ algorithm may identify more than one possible *BIC* per *BFC*. In this case, given the $PCS(BFC)$, we compute a true positive when a *BIC* exists and it is identified by the SZZ, and the rest of previous commits are computed as false positives.

The "gold standard" of Nova consists of 34 *BFCs* with a *BIC* and 12 *BFCs* without a *BIC*. While the "gold standard" of ElasticSearch is made up of 38 *BFCs* with a *BIC* and five *BFCs* without a *BIC*. When applying the SZZ algorithm to the set 46 *BICs* of Nova, it returns a set of 79[15] possible *BICs*. When the algorithm is applied to the 43 *BFC* of ElasticSearch, it returns a set of 85[16] possible *BICs*. Table 6.6 presents the percentage of true/false positives and false negative for the SZZ and SZZ-1 as well as the precision, recall and F1-score in each project.

In the literature there is no clear heuristic to follow by researchers when the SZZ identifies a set of possible *BICs* for a *BFC*. In Table 6.6 it is assumed that regardless of whether this case occurs, the algorithms flag all of the commits belonging to that set as possible *BICs*. However, some researchers in the literature state that when more than one commit is flagged, the *BIC*

[15]Three of the *BFCs* in the "gold standard" have only new lines, thus the SZZ cannot be applied. We compute these cases as false negatives because.

[16]seven of the *BFCs* in the "gold standard" have only new lines, thus the SZZ cannot be applied. We compute these cases as false negatives.

Table 6.7: Results of True Positives, True Negatives, False Negatives, Recall and Precision for the SZZ-1 algorithm assuming that the algorithm only flags the earlier commits that belongs to a set of $PCS(b)$ as *BIC*.

	True Positives	False Positives	False Negatives	Precision	Recall	F1-Score
Nova (SZZ-1)	25 (45%)	14 (25%)	17 (30%)	0.64	0.60	0.66
ES (SZZ-1)	16 (28%)	20 (34%)	22 (38%)	0.44	0.42	0.43

should be the earlier one in time. With this assumption, the number of true/false positives and true/false negatives of the SZZ-1 is once again computed using the "gold standard" of Nova and ElasticSearch, we selected only SZZ-1 because it performs better results than SZZ. Now, when applying the SZZ to the 46 *BFC* of Nova, it returns a set of 43 possible *BICs*. Furthermore, when SZZ is applied to the 43 *BFC* of ElasticSearch, it returns a set of 36 possible *BICs*. Table 6.7 presents the percentage of true/false positives/negatives for SZZ-1 as well as the precision, recall and F1-score in these scenarios.

RQ2: SZZ-1 performs better results than the original SZZ, but when applying the heuristics it still computes 25% of false positives in Nova and 34% in ElasticSearch. Assuming that the earliest commit caused the *BIC* only improves the F1-score in Nova.

RQ2.1: Which reasons caused that a previous commit identified by the algorithms was not the *BIC*?

Additionally, in those cases where the previous commit identified by the algorithm was not the *BIC*, we have looked for its cause. And without being exhaustive, we have identified some of the most common reasons why the previous commits analyzed were not the *BIC*, this reasons are following explained:

- *Variable Renaming*: It is a modification in the source code that changed the names of variables. When the algorithms identify a previous commit like this, it means that the line was already buggy or that this change is not related to the bug.

- *Equivalent change*: It is a modification of the source code to improve code quality

and efficiency such as optimization and refactoring. When the algorithm identifies a previous commit commit like this, it means that the bug was already in the logic of the program because the code optimization did not changed the behavior.

- *API changes*: It is a modification in the APIs of the program, normally the addition of a new argument. When the algorithm identifies a previous commit like this, and it is not the *BIC*, it means that the lines did not contain the bug, but the evolution and necessities of the project make that the line manifested the bug.

- *Obsoleted Code*: It is source code that have been useful in the past, but is no longer used. When the algorithm identifies a previous commit like this, it means that it is not the *BIC* because it removed obsolete code.

- *Changes done by the* BFC: The *BFC* may decide to change some parts of the code that are not related to the bug.

In the empirical study we quantitatively and qualitatively investigate the existence of *BIC* given a *BFC* by using the information stored in the control version system, bug tracking system and code review system of the software projects. For the quantitative analysis, we compare the true positives *BIC* and false negatives *BIC* manually analyzed across the results obtained from SZZ-based techniques. From the qualitative analysis, we inspect subset of the false negatives and false positives *BICs* to enumerate the most common causes for these two groups.

Chapter 7

Results and Discussion

This chapter aims to summarize the main results obtained in Chapter 4 and Chapter 6. The threats to validity and the potential automation of the theoretical model presented in Chapter 5 is also discussed.

We first discuss the threats to validity which raise potential questions that are answered in the following sections. Then, there are three subsections which present the results according to the objectives of this dissertation.

The first subsection presents the discussion of the SLR on the credibility and reproducibility of the SZZ algorithm. The importance of this discussion lies on the impact of this algorithm in the Empirical Software Engineering (*ESE*) as it has been cited more than 600 times since its publication in 2005.

The second subsection uses the results from the SLR to discuss the theoretical model to identify the change that introduced a bug. This model is mainly motivated by the shortcomings of the SZZ previously identified and the necessity of better understand software evolution.

Finally, the last subsection discusses the applicability of the theoretical model as well as the implications of the results from the empirical study where this model was applied. This model identifies the Bug-Introducing Commit (*BIC*) and whether it exists for a given Bug-Fixing Commits (*BFC*) based on the assumption of a hypothetical test that can be run indefinitely through the entire history of the software project.

7.1 Threats to validity

Wohlin *et al.* claims that there are four main types of validity threats in Empirical Software Engineering research: construct, internal, external and conclusion [Wohlin et al., 2012].

Construct validity: This type of threat is in regards to the validity of the metrics used in the empirical research method and whether they claim to be measurable. In the context of this thesis, this validity aspect will focus on the measures and concepts used and the potential problems that may cause bias in the results of the metrics during retrieval.

Internal validity: Internal validity is the degree to which a causal conclusion based on a study is warranted, which is determined by the extent a study minimizes systematic errors. An attempt was made to minimize this threat by following the procedures for performing SLRs described in [Madeyski and Kitchenham, 2017] and by reducing as much as possible the interaction with tools that can introduce bias in the results. Furthermore, replications packages are offered thereby that third parties can inspect our sources.

External validity: External validity is the degree to which results in this dissertation can be generalized to other contexts. In the empirical study, we selected two case studies: Nova, and ElasticSearch; with its particularities. Thus, this thesis cannot claim that our results can be generalized to other projects. However, researchers should not undermine the value of these case studies.

Conclusion validity: Conclusion validity is related to how sure the conclusions reached in regards to the relationships in our data are reasonable; this does not affect our approach.

7.1.1 Construct validity

Construct validity is the degree to which a research work represents what it is expected to measure. In the context of this thesis:

1. In the Systematic Literature Review:

- The impact of the SZZ is measured by the number of publications per year that use it. As well as the types and diversity of venues that published these publications.
- The reproducibility of the publications that use the SZZ is measured by the availability of a detailed description and data set or of a replication package.
- Since the studies have not been replicated or reproduced in this thesis, the effect on reproducibility is to be considered. However, to mitigate this, an upper limit of publications that may be reproducible are listed.

2. In the Empirical Study:

   - The *BFC* is a change that fixed a bug which was previously described in a bug report. As mentioned before, there may be cases where commits detected as the fix of a bug turn out to be false because the bug report did not describe a real bug. To mitigate this threat, in Nova the bug reports were analyzed by two researchers to ensure that their description was a real bug. Then, in a second round, the obtained datasets were again reviewed to filter out the uncertain cases.
   - The *BIC* is the commit that, involuntary, introduced a bug in the project. To accurate identify the *BIC*, it was manually selected by backtracking the fixed lines of a *BFC*. If the *BFC* only had new lines, we inspected and tracked back their surrounding lines. In case of doubt whether a commit is a *BIC*, it was directly rejected from the final "gold standard".
   - *FFM* is the change that manifests the failure the first time. It is identified manually and based on the hypothetical test. In case of doubt whether a commit is a *FFM*, it was rejected from the "gold standard".

We manually analyzed 187 out of 1070 papers during the SLR and 257 previous commits during the empirical study, thereby human errors might have occurred. This thesis provides replication packages with the raw data used in the SLR and the empirical study. Thus, researchers have the possibility to improve, check and replicate the work of this thesis.

7.1.2 Internal validity

The internal threats to validity are:

1. In the Systematic Literature Review:

   - There might be a selection bias due to having chosen Google Scholar and Semantic Scholar as the source of all publications on SZZ; other publications may exist that are not indexed by Google Scholar or Semantic Scholar.
   - There might exists publications that use the SZZ algorithm and not cite the original publication or its improvements.
   - The maturity of the field might also affect whether the study has been done in a too early a stage to draw conclusions. However, that more than 10 years is enough time to allow to extract valid lessons from its study.

2. In the Empirical Study:

   - The limited sample size of bug reports used in this research is the main threat to its validity. There may be the possibility that a prior unknown tendency occurs in the selection of these 120 random bug reports. Our analysis requires a lot of manual human effort, so meaningfully increasing the number of bug reports is difficult. However, it should be noted that our numbers are in the order of magnitude of similar studies: for instance, Hindle's *et al.* [Hindle et al., 2008] considered 100 commits in his study, Da Costa *et al.* [da Costa et al., 2016] considered to manually analyze 160 bugs as a whole and 80 *BICs*. Finally, Williams and Spacco [Williams and Spacco, 2008] manually analyzed 25 *BFCs* that contained a total of 50 changed lines which were mapped back to a *BIC*.
   - We are not experts in OpenStack and ElasticSearch, and although we have discussed and removed the uncertain cases, our inexperience may have influenced the results of the analysis when identifying whether a bug has been in the system from the very beginning or whether the code has always been clean.
   - A random script was used to extract the bug reports from Launchpad and GitHub that were reported during 2015. There could be unintended bias in the data throughout the course of this year, for instance the phase of the project.
   - There could be some lax criteria involving the subjective opinion of researchers. To mitigate this, the Bug-Fixing Commit that was not sure was discussed. Fur-

thermore, the "Undecided" category was added to mitigate possible false positives from our dataset.

- Commits classified into "Undecided" group were rejected as our aim was to build a trustable dataset in which we were sure that the bugs reports were "real" bug reports and that the *BICs* and *FFMs* were as very accurate as possible. There may be the possibility that those commits could have relevant information, and as a consequence, some data that would vary the results.

7.1.3 External validity

The most important external threats are related to peculiarities of the projects:

1. In the Systematic Literature Review:

   - The results about reproducibility using SZZ cannot be generalized to *ESE* research because it was a case of study.

2. In the Empirical Study:

   - The use of diff is the most extended way of providing diff information when looking for the difference between two files. However, other ways of providing diff information should be considered.
   - This thesis only has selected two different programming languages, Java and Python. It is possible that the study of different programming languages turns out in different results.
   - The use of Nova and ElasticSearch as the case studies implies a better understanding of how bugs appear in these projects. However, Nova and ElasticSearch are projects with a very rapid evolution and a very active community of developers. It could be possible that other projects with fewer commits per year have different results. A higher number of projects would enrich the study.

7.2 Discussion

There are three different discussions, based on what has been presented in Chapter 4, Chapter 5 and Chapter 6.

7.2.1 Discussion: Reproducibility and Credibility of the SZZ

Chapter 4 details the Systematic Literature Review on the use of reproducibility and credibility of SZZ algorithm. It was demonstrated to be a widely used algorithm in *ESE* research. The SZZ is proven to be significantly relevant, and that it is not limited to a niche audience. However, it has become an important component of publications in top journals and prominent conferences in many different topics of Software Engineering.

During the SLR, it was observed that the limitations of SZZ are well known and documented. Improvements have been proposed, with unequal success up to this moment. The first part of the limitations –related to linking fix commits and bug tracking issues – has been improved significantly, the enhancements for the second part that has to do with finding the bug introducing change are still limited and accuracy has room for improvement.

Even if limitations have been widely documented, from our study, it can be observed that this has not made ESE practices stronger. From the detailed study of the threats to validity of publications using SZZ, SZZ-1, and SZZ-2, it is clear that most publications are not reporting the limitations, and interestingly enough, limitations to the first part –which have shown to be less relevant– are discussed more often than the second part. The fact that 38% of the publications use the *original* SZZ is indicative of this regard. It is therefore recommended for researchers who develop modifications of the SZZ algorithm, to publish the software implementation in development sites such as `GitHub`, so other researchers can *fork* the project. These *forks* can be easily traced, and the authors will be able to ask for a specific citation to their solution if other researchers make use of it.

The reproducibility of the publications is discovered to be limited, and replication packages are seldom offered. The results presented in our research are in line with previous research [Amann et al., 2015], although not as *bad* as the ones found for MSR in 2010 [Robles, 2010]. In any case, we think they are not satisfactory enough, and some reflections should be raised on scientific methods used in our discipline.

Even if using one of the improved versions of the algorithm helps with the accuracy of the SZZ approach as pointed out in [Rahman et al., 2012]: "Accuracy in identifying Bug-Introducing Commit may be increased by using advanced algorithms (Kim et al. 2006, 2008)", they are seldom used and only 22% of the publications use one of the two (improved) revisions of SZZ. It seems that researchers prefer to *reinvent the wheel*; 49% use an ad-hoc *modified* version of SZZ in their publications instead of using other researcher's improvements. One possible reason for this is that papers that describe the SZZ algorithm or any of its improvements do not provide a software implementation. Thus, researchers have to implement it from scratch for their investigation. Our results show that in such a situation, what is often practiced is the adaption of base SZZ algorithm with the addition of modifications, resulting in an 'ad-hoc' solution. For all 'ad-hoc' solutions identified, there is an absence of rationale on why other enhancements to SZZ have not been implemented. Another major problem when using improvements to SZZ is that they have not been given a version/label. Even if a revision of SZZ is used, publications often refer to it as SZZ, making it difficult to follow, to reproduce, to replicate and to raise awareness on this issue.

This thesis provides a simple way to measure the ease of reproducibility and credibility of research papers. Although this measure was formulated only to studies that make use of the SZZ, we think that it can be adapted to other ESE studies easily. Thus, authors can easily assess whether their papers offer a reproducible and trustable work (i.e., with scores above or equal to 5). A Although authors were often addressed directly, the reviewers in the scientific process must not forget their responsibility. It can be observed that authors are often detail-focused when presenting their research; reviewers often have the required vision to evaluate the studies objectively and retains the ability to take these details into consideration. This helps authors to raise the level of their research to another standard. Thus, it is recommended for reviewers to always question the listed points in Chapter 4 and adapt them to the context of the research when performing a review of a paper which uses heuristics and assumptions..

It was observed that full comprehensive reports on reproducibility are seldom found. Only 15% of the total analyzed papers are classified as *good and excellent quality with respect to reproducibility*. The research community should direct more attention to these aspects; we believe that too much attention is put on the final result(s) (the *product* of the research: new knowledge) as opposed to the research process. As researchers in the field of software engi-

neering, we know that both, a high-quality product and process, are essential for a successful advancement for the long term [Kan, 2002].

To summarize, it is considered important to highlight some of the lessons learned after performing the SLR. The SZZ algorithm is based on heuristics and assumptions, thus to provide more trustable results, it is recommended for researchers to specify (and argue) the use of the methods/algorithms that mitigate the limitations of their studies. They must also be aware of the risk of every assumption used and, if needed, provide a manual analysis of the results. For studies where the study size is large, researchers can select a random sample to validate it manually. In addition to carrying out empirical studies, the authors must be conscious that the best method to produce reproducible studies is to include a replication package that can be publicly available together with its publication (ideally for an indefinite time). They also have to be aware that some characteristics in their studies, such as software environments, might change, causing both programs and data to become obsolete. As a result, a detailed description of the elements, methods, and software used during the study is also valuable.

7.2.2 Discussion of the Theoretical Model

Chapter 5 addresses the issue of identifying changes that introduced bugs. Currently, the lion's share of the work is based on methods and techniques which rely on the inherently flawed assumption that the lines of code that have been modified to fix a bug are also the lines that previously introduced the bug. While the questionable nature of this assumption is known, this thesis makes an important contribution by detailing a new model to explain how bugs appear in software products. This proposed model does not blame all identified changes as *BICs*, but instead, it searches for when bugs manifest themselves for the first time, and how that can be determined by running a test.

After analyzing 120 bug reports in this study, we noticed that determining where, when and how a bug is introduced is not a trivial task. There are situations where many researchers were involved to clarify the essence of the malfunction. In fact, it is in some cases difficult to even determine whether a bug was present in the code at the time of a given commit's submission. For example, there are cases where the current automatic approaches are unable to determine the point of introduction of a bug, as no previous commit can be identified. One

of those cases is when the *BFC* only added new lines in the source code: in such case, there is no way of identifying the previous commit as defined, since there is no previous commit before the addition of lines. In this case, the description of the bug report as well as the surrounding lines of the new additions in the *BFC* could clarify the reasons for the new lines to fix the bug. For instance, whether the new lines were not included in some ancestor commit causing the bug, or the new lines were included in the *BFC* to fix a bug related to satisfy the evolution of requirements and characteristics of the project.

Thanks to the results of the SLR, the limitations, and problems that affect SZZ-based algorithms when identifying the *BICs* were quantified. Thus, the proposed model integrates these scenarios and is able to deal with them. An explanation of how the model can address each of the limitations is given.

1. Identification of more than one previous commit: The model does not focus on identifying the previous commit set of a *BFC*. It instead tests in all the ancestor snapshots of the project the behavior or the functionality that was reported as a bug, to find the first time that bug manifested itself in the software.

2. Scenarios when only new lines are used to fix the bug: The model does not look for lines that have been modified or deleted to fix a bug. It instead considers all the ancestor commit set to test when the malfunction manifested itself for the first time.

3. External changes or changes in the environment: The model can detect bugs caused by changes in the environment or external changes by using a test that checks whether the functionality that was fixed in the *BFC*, and detailed in the bug report, behaves correctly in the ancestor snapshots. This means that the hypothetical test will pass or fail when it is running into previous snapshots with the state of the new environment fixed in the *BFC*. Thus, when the snapshots passes the *TSB* means that the environment or external change was already implemented, while if the *TSB* fails means that in this snapshot the new environment or external change was not implemented. In this scenario, the *BIC* cannot be identified since any previous change introduced the bug. Consequently, the malfunction was caused by something external to our project, and only the *FFM* can be found.

4. Multiple modifications over a line: In our model, it does not matter how many times

a line has been modified since it does not blame the last change as the *BIC*. The test looks for the first time that the bug was introduced in the project.

5. Weak semantic level: As in the example above, the proposed model deals with the semantic level of changes because it is not focused on identifying the last change; it instead tests the behavior of the code.

6. Scenarios when a *BFC* fixed more than one bug: In these cases, our model can design a specific test for each bug, looking for the first time that the test fails when testing each bug.

7. Compatibility reasons: The model identifies the first time that the bug was manifested in the source code using the *TSB* to identify the *BIS*.

8. Dormant bugs: The model identifies the first time that the bug was introduced into the source code using the *TSB*. Thus, it does not matter how long the defect had been in the project.

Many researchers have based their methods for locating *BICs* in the versions of the SZZ algorithm or similar algorithms that lack the means to deal with these limitations. They formulate heuristics that for example, remove the *BFCs* with only new lines to fix the bug because they cannot track back these lines, or they remove longer *BICs* or older commits because they are unlikely to be the *BIC*. Sometimes, the consequence may be mislead results and lose accuracy when identifying *BICs*.

The SZZ is currently the best-known and easiest algorithm used to identify *BICs*. As a result, some studies have used datasets to feed their bug prediction or classification models with results obtained from the SZZ. Ray *et al.* that studied the naturalness of buggy code [Rahman et al., 2014]. Massacci *et al.* evaluated most existing vulnerabilities of discovery models on web browsers and took many datasets where at least the built-in used the SZZ approach [Massacci and Nguyen, 2014]. Abreu *et al.* obtained a dataset using the SZZ and studied how the frequency of communication between developers affects the fact to introduce a bug in the source code [Abreu et al., 2009].

In general, with this new method it may be possible to distinguish between two relevant moments given a *BFC*, the *BIC* and the *FFM*. The first moment does not always exist, because

there could be external changes or the instance whereby the internal requirements could have evolved which may have caused the failure. However, there is always a *FFM* which indicates the first moment when the project manifests itself the bug. To obtain more accurate results in areas such as bug prediction, bug categorization, and bug detection, the approaches used to identify when a bug is introduced should distinguish between these two moments, the *FFM* and the *BIC*. The empirical case of study states that the current state-of-the-art approaches identify from 26% and up to 45% of "real" *BIC* (true positives). Thus, these results are promising to start thinking about implementing new methods based on the theoretical model proposed in this dissertation, or at least, to consider re-formulating correctly the definition of bug introducing. The bug introduction process cannot be considered as a static problem; the current methods need to be able to distinguish between buggy lines and clean lines at the moment of their insertion. This dissertation has demonstrated that other external and internal reasons rather than buggy lines are causing the failure of the systems and that it is not fair to blame a change as the cause when in fact, it did not insert the error as it fulfilled with the functionalities required, environments, necessities, and requirements of the project in that moment. Although more lights have been given to the problem of emulating software faults realistically, to achieve greater bug localization automation, a more concerted effort is needed in testing to find ways or techniques to address the re-built problem and to build a test that can be automated or partially automated to find the *BIC* or *FFM*.

7.2.3 Discussion of the Empirical Study Results

The empirical study of Chapter 6 supports the necessity of a new model to identify the changes that introduced bugs as it is not always obvious how errors are introduced into the source code with the current state-of-the-art approaches. This dissertation introduces a new model that might solve this necessity in Chapter 5. Its implementation has been empirically studied in Chapter 6, we have carried out a qualitative study on a set of bug reports to identify whether a *BIC* exists and, in this case, to pinpoint where it was introduced given a *BFC*.

The identification of the lines that are responsible for inserting the error is a quite complicated process. Out of 120 bug reports from the datasets of Nova and ElasticSearch, 116 bug reports with their respective Bug-Fixing Commit were able to pass to the second stage. In this second stage, researchers manually analyzed whether or not the changes identifies from

the *BFC* introduced the bug. In both projects, the percentage of changes that introduced a bug into the source code was higher than the others categories, with 79% of occurrence rate in Nova and 91% in ElasticSearch. However, the percentage of *BFC* that were not caused by introducing a *BIC* is more representative in Nova with 21% of occurrence rate, while in ElasticSearch the 9% of the *BFC* did have a *BIC*. We hypothesize that a possible reason can be attributed to the programming language. While Python is a dynamic language that can run without compilation, Java is strictly typed and performs run-time checks, which may decrease the category of *BFC* that were not induced by a *BIC*.

During the manual analysis, we discover some reasons which may explain why a *BFC* was caused by the insertion of buggy lines. In Nova, the principal reason was the *Co-evolution Internal* with a 42% occurrence rate, followed by 33% of the cases where bugs are in external APIs. The third most frequent reason was the *Co-evolution External*, with a 17% occurrence rate and finally, the less frequent reason was *Compatibility* with a 8% of the cases. On the other hand, in ElasticSearch the percentages are equally distributed with a 40% occurrence rate in *Co-evolution Internal* and *Bug in External API*, 20% of occurence in *Co-evolution Exteranal* and bugs caused by the incompatibility of hardware and software were not found. This anecdotal classification should be further investigated since it can help researchers to identify different patterns, probably hidden, which can explain better how bugs are introduced and manifested in the source code. It also does not claim that these categories can be extended to other projects. We draw attention to the fact that there are other different reasons that cause bugs, and it should be better studied to improve current classifications. To be noted that, although this manual analysis was mostly conducted by the author of this dissertation, in case of doubts, other researchers were involved to discuss and analyze the root cause of the bug. It may be possible that some differences in judgment about the classification of the root cause exist whether this study is replicated in the future. In fact, even if the location and time stamp of a *BIC* are known, we need to (1) understand the bug and how to fix it in the case we have indeed found the actual *BIC*, or (2) determine that what we assume to be the *BIC*, is in fact not the *BIC*, but simply the *FFM* because the bug was not present at this moment.

After examining 120 bug reports in this study, we notified that attempting to agree on the classification of the root causes of defects is a tedious and sometimes subjective process. Thus, this dissertation suggests an initial classification, without trying to make any strong

statement. From the results, it was observed that from 9% to 21% of the analyzed *BFC* were not induced by a *BIC*. This results can be compared to the results obtained by Wan *et al*, they defined a category for bugs caused by environmental and configuration issues as *"Environmental and configuration bugs correspond to the bugs that lie in third-party libraries, underlying operating system, or non-code parts"* and found that 11.42% of the reported bug studies fall into this category [Wan et al., 2017]. Thus, our initial classification may help to understand better the reasons why a *BFC* was not caused by a *BIC*.

This dissertation discusses a myriad of options in terms of what a *BIC* can and cannot be, and also how an SZZ algorithm will fail, depending on the specific implementation/version of the SZZ algorithm. An essential element of the empirical study is the "gold standard", which has the characterizations of *BICs*. The study was able to quantify the "real" number of false positives, false negatives and true positives in the performance of the SZZ algorithm, and as far as we know, nobody has attempted to calculate it before. According to the results presented in Chapter 6 Section 3.4, in the best scenario, the SZZ and SZZ-1 compute 55% and 35% of false positives in Nova with a F1-Score of 0.44 and 0.66 respectively. In ElasticSearch, the number of false positive in SZZ and SZZ-1 61% and 34% with a F1-score of 0.43 in both algorithms. These results may appear to be contradictory with the previous results where ElasticSearch has a fewer percentage of bugs that were not introduced by *BICs*, and it may cause a lower number of false positives than Nova. However, the reason why ElasticSearch computes more false positives may be because the presence of *BFC* with a *PC* set greater than 1 is higher than in Nova. Thus, it explains why the heuristics of the SZZ fail more frequently, and almost in half of these cases. The assumption made about that the earlier commit belongs to the *PC* set is the *BIC* is frequently wrong in these cases. When there is a *PC* set with more than one pc, the SZZ algorithm makes heuristics to identify which commit from the set is the *BIC*; in this case, the SZZ algorithm looks for the earlier commit and blames it as the *BIC*. Another reason that explains why Nova computes higher recall than ElasticSearch is the number of *BFC* with only new lines. This causes the number of false negatives to increase since SZZ does not include these *BFCs* in their analysis.

Other studies such as [Kim et al., 2006c], [Williams and Spacco, 2008], and [da Costa et al., 2016] also manually analyzed samples of SZZ data, although they reported much higher percentages of correct SZZ results than this thesis. This is because these studies were not searching

for the "gold standard"; they do not distinguish between *BICs* and *FFMs*, and they do not define what a bug is. As results, they do not contemplate other scenarios which exist and other factors such as changes in external APIs or co-evolution changes. While this dissertation analyzes the whole context of a *BFC*, these studies only focus in verifying the bug-fixing hunks, thus they could omit changes in the API or changes due to the evolution of the code. On the other hand, there is also the possibility that the specific projects selected in these studies had a fewer percentage of false negatives due to different cases of bug introduction. This is one of the reasons why we propose a further work to extend the analysis to a varied set of projects.

Our research provides empirical evidence in the assumption that the previous commit is where the cause of a bug is located is not true for a significant fraction of bugs up to 25% in the best scenario. SZZ-based algorithms cannot find the change that introduced the bug mainly due to the impossibility to trace further some lines. Da Costa *et al.* studied the impact of refactoring changes on different implementations of the SZZ and observed that 19.9% of the bug-introducing lines changed in a *BFC* are related to refactoring changes, and 47.95% of the bug-introducing lines contained equivalent changes, in both cases these algorithms fail to identify the *BIC* as they cannot track further these lines.

In addition, the results also show that our proposed model, theoretically, fulfills with the fundamental concern of identifying the *BIC* when it exists. The results also indicate that in the best scenario, the use of SZZ presents a 54% of true positives. These results suggests that the existing techniques based on the SZZ algorithm are generating several false positives and should be improved or reformulated in order to consider all the limitations mentioned in this dissertation. However, further investigation on the automation of the model needs to be conducted, as well as further study on how external factors and the evolution of requirements affect the manifestation of bugs in the projects.

Chapter 8

Conclusions and Future Research

This chapter recapitulates the initial research goals and contributions claimed in Chapter 1 and describes the main conclusions of this thesis. Furthermore, the future research work is also detailed in section 8.2.

8.1 Conclusions

Software bugs are important information to understand the process of bug insertion and bug manifestation. The complete understanding of bug insertion means to correctly identify changes that introduced bugs in software products. This understanding needs to distinguish between when a bug is inserted from when a bug is first manifested. Understanding how bugs are inserted helps in different areas of software engineering such as bug prediction, bug proneness, bug detection or software quality. Thus, this thesis covers the entire research cycle of bug introduction: from investigation and identification of the shortcomings of the state-of-the-art approaches to the design of a model, and its empirical application.

Specifically, in this dissertation we have observed that the basic distinction between the bug introduction moment and the bug manifestation moment has not been adequately delineated in the current literature of software bugs. As a consequence, a widely used algorithm to identify where bugs are introduced suffers from severe reliability and credibility concerns as shown in Chapter 4. In Chapter 4, we have performed a case study of ESE practice by means of conducting a SLR on the use of the well-known SZZ algorithm. Our sample of publications consists of 187 publications that make use of the SZZ algorithm, out of 458 pub-

lications that cite the seminal paper. This SLR sheds some light on some of the problems that ESE research faces when assumptions, as it is the case in SZZ, are made in research methods: publications are primarily non-reproducible, limitations are only occasionally reported, improved versions are seldom used and difficult to identify, and researchers prefer to use their own enhancements.

Thus, to address these challenges, Chapter 5 presents a model to identify the changes that introduced bugs in the system. The model includes an overview of the challenges of bug introduction identification that can be used to define what is a bug and how practitioners may identify when it was inserted by assuming that there exists a hypothetical test that can check the correct behavior of the project in different snapshots. Chapter 5 also formulates a comprehensive nomenclature to argue about bug introduction and formally describes the theory of bug introduction. To our knowledge, this is the first study aiming to formulate a nomenclature for the bug introduction process.

The application of the theoretical model into practice may be complicated without a full automatization, however, Chapter 6 describes an empirical study where the model was applied in a manual process leading to identification of Bug-Introducing Commits (*BICs*). The empirical study demonstrates the challenges and shortcomings of existing SZZ-based approaches and provides means of constructing a "golden standard" for their evaluation. It also demonstrates, empirically, that the correct identification of changes that introduced bugs is far from perfect and further research is needed because the results indicate that other factors such as the internal evolution of the requirements or external changes cause that up to 9%-21% of the bug were not introduced in the project.

To summarize, a plethora of research studies focus on helping practitioners to identify bugs in software products by detecting, predicting and understanding them. However, we have demonstrated that some bugs are not really inserted in the project, and hence this distinction is not accounted for in such studies. Moreover, when a change in the project inserted the bug, the current techniques compute several false positives when attempting to identify it because they make many assumptions. To deal with this problem we have quantified the limitations of these techniques and we have provided a solution. This solution describes a new model to determine when a change introduced an error and when the project manifested this error for the first time. We demonstrated its accuracy over two software project and provide a

framework to evaluate the different versions of the SZZ-based techniques. The contributions of this dissertation open multiple directions for further research in several areas of software engineering which are related to software bugs.

8.2 Future Work

After manually identifying the Bug-introducing commits (*BIC*) and the First-Failing Moments (*FFM*) using the criteria detailed in the proposed model, the next logical step is to automatize the theory as much as possible in order to automatically find the *BIC* and *FFM* given a Bug-Fixing Commit (*BFC*). As mentioned earlier, there will be some scenarios where the Test-Signaling Bug (*TSB*) may be unable to implement due to reasons related to dependencies and environments used at some previous point in the history of the project. Thus, as future work, I would like to study the extent to which this occurs in an optimal project where all the dependencies would be under control, where each one of the previous environments of the project can be most likely replicated. We hope that the automation of the proposed model is also interesting from a practical point of view, because it would provide software projects with a valuable tool for better understanding what is a bug and how it is introduced, and therefore design measures for mitigation.

Another future line is to select a bigger sample size to carry out a classification to study the frequency with which a *BFC* presents a *BIC* depending on the bug report description and the software change in the *BFC*. This study will provide us with more in-depth knowledge on whether there are patterns that exist which have been hidden in the current literature. This study could help to better design integration tests, to better identify real *BICs* or to check for these cases.

Finally, another interesting future line concerns with the reproducibility and replicability of previous studies based on the use of SZZ-like techniques in order to predict, detect and classify bugs. There is a need for boosting reproducibility and replicability to investigate how it should be addressed by the ESE community, this will help to ascertain the impact of this thesis in other domains as well as the necessity of improving and better defining what is a bug and how it can be located.

Appendix A

Replicability of the Results

Some authors such as Shull *et al.* [Shull et al., 2008] and Basilli *et al.* [Basili et al., 1999] have dealt with the task of Mining Software Repositories *MSR*. This is a complex task because of the high amount of time spent, the necessity of developing some specific tools and the management of datasets. Furthermore Robles [Robles, 2010] has raised some questions related to this problem in the ESE.

In first place, there are several elements that may be of interest for reproducibility:

A.1 SLR

Original data source: The original data sources used along this thesis can be found in `http://gemarodri.github.io/Thesis-Gema`.

Extraction methodology: The methodology is briefly detailed in Chapter 4, section 3. While the scripts used can be also found at `http://gemarodri.github.io/Thesis-Gema`

Study parameters: The initial filter applied over the raw data set is detailed in Chapter 4, Section 3.

Results dataset: The results can be found in Chapter 4. In addition, they are also available at `http://gemarodri.github.io/Thesis-Gema`

Persistence: it is expected to have access to the website as long as GitHub exists.

A.2 Empirical Study

Original data source: The original data sources used along this thesis can be found in `http://gemarodri.github.io/Thesis-Gema` or along Chapter 6, Section 1.

Extraction methodology: The methodology is briefly detailed in Chapter 6, Section 2. While the scripts created and needed can be found at `http://gemarodri.github.io/Thesis-Gema`

Study parameters: The initial filter applied over the raw data set is detailed in Chapter 5, Section1 and Section 2.

Results dataset: The results can be found in Chapter 6, Section 3. Furthermore, they are also available at `http://gemarodri.github.io/Thesis-Gema`

Persistence: it is expected to have access to the website as long as GitHub exists.

To be noted, this thesis should change, if this happens, the last version of this thesis is available at: `http://gemarodri.github.io/Thesis-Gema`.

Appendix B

Resumen en Castellano

B.1 Introducción

Los sistemas de software siempre han tenido errores, su búsqueda ha ocupado y ocupará gran parte de las tareas diarias de los desarrolladores de software. Los sistemas de software se encuentran en continua evolución con cambios continuos en el código fuente, y además se han vuelto más complejos a lo largo del tiempo, ya que necesitan manejar una cantidad significativa de datos, integrar módulos de terceros y ejecutar multiplataformas. Como consecuencia, desarrollar y probar sistemas de software es actualmente un gran desafío para los ingenieros de software.

Por lo tanto, la aparición de defectos es inevitable, aunque es común usar técnicas para detectarlos y prevenirlos con anterioridad, la tasa promedio de defectos arreglados en la industria cuando se trata de crear software es de aproximadamente de 1 a 25 errores por cada 1,000 líneas de código [McConnell, 2004]. Esto revela la necesidad de comprender cómo se introducen los errores en el código fuente para minimizar la posibilidad de que se introduzcan en los sistemas de software. Esto hace que sea especialmente importante comprender los procesos que llevan a la introducción de errores, la manifiestación de errores y a su posterior arreglo. La mejor práctica para entender cómo se insertan los errores en el código fuente se basa en el estudio de sus informes, causas y soluciones. Específicamente, el estudio de los cambios que arreglan un error es un interesante ejercicio que permite a los investigadores comprender la importancia de entender como se introducen los errores en diferentes áreas de Ingeniería de Software. Por ejemplo, determinar

por qué y cómo se introduce un error puede ayudar a identificar otros posibles cambios que introducen errores [Śliwerski et al., 2005b], [Kim et al., 2006c], [Zimmermann et al., 2006], [Thung et al., 2013], [Sinha et al., 2010]; puede ayudar a descubrir patrones de introducción de errores que podrían conducir al descubrimiento de métodos para evitarlos [Nagappan et al., 2006], [Zimmermann et al., 2007], [Hassan, 2009], [Hassan and Holt, 2005], [Kim et al., 2007]; puede ayudar a identificar quién es el responsable de insertar el error, que tiene el potencial de ayudar en el autoaprendizaje y los procesos de evaluación por pares [Izquierdo et al., 2011], [da Costa et al., 2014], [Ell, 2013]; puede ayudar a comprender cuánto tiempo está presente un error en el código, lo que permite la evaluación de calidad [Chen et al., 2014], [Weiss et al., 2007], y así sucesivamente. Debido a estas razones, este campo de estudio ha estado activo durante las últimas décadas.

Hay mucha información sobre cómo se informan y se gestionan errores en los sistemas de seguimiento de errores, como Bugzilla, Launchpad o Jira. Esta información proporciona una descripción textual sobre los *síntomas* del error, así como los medios para reproducirlos o enumerar los módulos afectados por el error. Esta información se puede vincular al sistema de código fuente que proporciona el *tratamiento* mediante cambios en el código fuente para corregir el error.

Aunque hay mucha información disponible, realmente encontrar cuándo se introducen los errores es difícil, hasta el punto de que no está claro cómo definir *“ Cuándo ”* que se introduzca un error. El término, *“ error ”* no se encuentra claramente definido, por lo que los investigadores simplemente están adoptando suposiciones actuales que se encuentran presentes en la literatura. Estas suposiciones establecen que las líneas modificadas para corregir el error probablemente sean la causa de dicho error.

Sin comprender que significa tener un error en el sistema y cómo se introduce, es difícil desarrollar un enfoque exitoso para identificar cuándo se inserta un error en el software. Es particularmente importante aclarar que nuestra definición de error considera que un error solo se introduce en un sistema cuando se introducen líneas defectuosas en el código fuente. Y el momento en que se inserta el error se debe distinguir del momento en que un sistema manifiesta el error por primera vez. Aunque ambos momentos pueden ser iguales, el momento de la introducción del error se puede identificar mediante el uso de una prueba de test hipotética que verifica si el código en el momento de su escritura presenta los *síntomas* descritos en el

informe de error, por tanto:

- Cuando falla la prueba, significa que las líneas contenían errores en este momento, y podemos estar seguros de que el error se insertó en el momento en que se insertaron las líneas.

- Cuando pasa la prueba, significa que las líneas estaban limpias en este momento y que en ese momento no hubo un momento de introducción de errores. Esto se debe a que un sistema tiene diferentes necesidades en diferentes puntos a lo largo del tiempo y los requisitos del sistema cambian y es posible que no se administren adecuadamente; por ejemplo, una línea limpia insertada en el momento A puede no causar la falla del sistema, pero cuando el sistema alcanza el momento B, la línea puede desencadenar el error que causa la falla del sistema. Esto es posible porque el sistema usa dependencias externas que habrán cambiado entre el momento A y B. Otra posible explicación sería que alguna biblioteca externa contiene un error que se insertó en el momento B. Una tercera explicación sería debido a la evolución del sistema que provocó cambios en otras partes del código, afectando así al código fuente en el momento al alcanzar el momento B.

Además, otros factores también complican el desarrollo de un enfoque exitoso para identificar el primer momento de falla del sistema. Por ejemplo, cuando varias modificaciones sobre la misma línea podrían encubrir la verdadera evolución de una línea de código, ocultando así la causa del error [Servant and Jones, 2017] o cuando solo hay nuevas adiciones de líneas para arreglar el error [da Costa et al., 2016]. Por otra parte, los enfoques pueden disminuir la precisión cuando los cambios que se arreglan no son modificaciones no esenciales [Herzig and Zeller, 2013] o cuando un cambio no está relacionado con las líneas del código que solucionó el error [German et al., 2009].

En la actualidad, muchos investigadores solo han realizado estudios para identificar el cambio que insertó el error a través de tracear las líneas modificadas que corrigierion el error. Sin embargo, este enfoque es ineficaz respondiendo a preguntas relevantes como son *"¿qué causó el error? "* y *"¿esta línea era errónea en el momento de su inserción? "*. Porque hasta el momento, no se ha encontrdado ningún modelo significativo que pueda validar si la línea identificada, por uno de los métodos actuales, contenía el error cuando se insertó en el código

fuente.

Para investigar y comprender mejor cómo aparecen los errores en los productos de software, esta tesis presenta un modelo que define el cambio de código que introdujo un error. Este modelo también establece la distinción necesaria entre el momento de introducción del error y la primera vez que el error se manifesta en el sistema. Estos momentos no solo se enfocan en estudiar el *tratamiento*, sino que también analizan los *síntomas*. Siguiendo la evolución de las líneas que se han cambiado para corregir el error, el contexto y la historia del error puede ser más fácil de entender, y por lo tanto es teóricamente posible descubrir si las líneas modificadas para corregir el error introdujeron el error o por el contrario, para confirmar que existen otras razones diferentes a la introducción de líneas erróneas que provocaron la aparición del error en el sistema. El conocimiento de esta información puede ser muy útil en muchos campos de la ingeniería de software. Por ejemplo, al calcular varias métricas, como la experiencia del autor y su actividad en el proyecto [Izquierdo et al., 2011], [Izquierdo-Cortázar et al., 2012]. Además, permite mejorar técnicas avanzadas que aprovechan dicha información para aprender patrones de estos cambios, para motivar el diseño y desarrollo de mejores mecanismos [Harris et al., 2010], o para ayudar a automatizar predictores de errores que estiman la probabilidad de que dado un cambio, éste inserte errores [Rao and Kak, 2011], [Thung et al., 2012], [Zimmermann et al., 2007].

B.2 Antecedentes

En la literatura existen dos corrientes diferenciadas sobre el estudio de las *causas potenciales* que hacen que los desarrolladores introduzcan los errores. Por un lado, existen técnicas que se basan en el estudio de los cambios que arreglan un error, específicamente, identificando las líneas que han sido modificadas para encontrar que cambio previo las introdujo. Sin embargo, las otras técnicas usadas por los investigadores se basan en el análisis de trazas en el código fuente, es decir, estos métodos analizan la asociación entre los fallos de un programa y la ejecución de alguno de los elementos de un programa.

A continuación se describe una breve introducción al estado del arte actual. En el Capítulo 2 se realiza una descripción más detallada.

B.2.1 Siembra del error

B.2.2 Identificar el cambio que arregló e error

Ness y Ngo fueron los primeros en estudiar los cambios que podían introducir errores, para ello usaron una búsqueda simple lineary binaria, que tenía como propósito aislar el cambio que provocó el arreglo aplicando cambios cronológicos al programa hasta que la version arreglada presentase el mismo comportamiento erróneo que que la siguiente versión del programa [Ness and Ngo, 1997]. Una de las limitaciones de esta técnica se producía cuando un conjunto de cambios provocaba el error, para solventar esta limitacioón, Zeller propuso la automatización de la técnica delta debugging, esta técnica determina el mínimo conjunto de cambios que inducen a arreglos [Zeller, 1999].

Purushothaman y Perry midieron la probabilidad, menos del 4%, de que un cambio pequeño introdujese errores, nombraron a estos tipos de cambios como *dependencias*, que son cambios en las líneas de código que fueron cambiadas por un cambio previo, asumiendo que si el cambio posterior en esas líneas era para arreglar un error, entonces quien lo introdujo fue el cambio previo, ya que era erróneo [Purushothaman and Perry, 2004]

Cuando existe un cambio que arregló el error (*BFC*): Śliwersky *et al.* describió como identificar este tipo de cambios en proyectos que usaban sistemas de versión de archivos. Además propusieron el algoritmo *SZZ*, este algoritmo es uno de los más usados en la literatura actual y se basa en identificar los cambios que arreglan errores para analizar las líneas que han sido modificadas o eliminadas, asumiendo que el último cambio realizado en esas líneas antes del arreglo fue el cambio que introdujo el error [Śliwerski et al., 2005b].

La popularidad en el uso de este algoritmo, provocó que algunos artículos estudiasen como mitigar las limitaciones que presentaba el SZZ. Kim *et al.* sugirió una nueva implementación del SZZ que se basaba en la técnica de *annotation graph* para identificar las líneas que habían sido afectadas por el cambio que arregló el error. Además, en este artículo los autores mejoraron la técnica eliminando del análisis algunos casos como modificaciones realizadas por el *BFC* en líneas en blanco, en los comentarios o cambios del formato [Kim et al., 2006c]. Por otro lado, Williams y Spacco propusieron una nueva versión del algoritmo SZZ. Esta versión usa diferentes pesos (weights) para mapear la evolución de

una línea. Además, esta técnica también ignora los comentarios en blanco y los cambios de formato en el código fuente [Williams and Spacco, 2008]. Sin embargo, a pesar de estas mejoras, el algoritmo seguía teniendo limitaciones y seguía identificando erróneamente cambios que no causaban errores como los cambios que provocaron el posterior arreglo. Por ese motivo, Da Costa *et al.* crearon un marco para eliminar del análisis los cambios poco probables de introducir el errores. Este marco se basa en una serie de heurísticos como las fechas en las que los supuestos cambios que provocaron el error fueron realizados, las fechas en las que los errores fueron reportados en el sistema, etc [da Costa et al., 2016].

Cuando no existe un cambio que arregló el error: La localización de errores basada en espectro es una técnica usada para identificar el origen del fallo. Reps *et al.* usa ésta técnica, cuya entrada es un dos conjuntos de rangos, para ejecuciones exitosas y fallidas, y produce candidatos (líneas, métodos, bloques...) que explica las posibles razones del fallo [Reps et al., 1997]. Abreu *et al.* investigó la precisión de diagnóstico en la localización de errores como una función de varios parámetros. Sus resultados indicaron que el rendimiento superior de un coeficiente particular es en gran parte independiente del diseño del caso de prueba [Abreu et al., 2007].

Otra de las técnicas que se usa para localizar el cambio que introdujo el error se denomina el vecino más cercano, en inglés conocido como *Nearest Neighbor*. Renieres y Reiss usaron esta técnica para localizar el fallo [Renieres and Reiss, 2003]. La técnica tiene dos partes principales: En primer lugar, selecciona una única ejecución fallida y después, calcula la ejecución pasada con la mayor cobertura de código similar. En segundo lugar, crea el conjunto de todas las sentencias que se ejecutan en la ejecución fallida pero no en la pasada ejecución.

Zeller y Hildebrandt desarrollaron por primera vez el algoritmo de Depuración Delta, más conocido en Inglés como el *Delta Debugging algorithm*, que compara los estados de fallo y éxito de una ejecución del programa, y usa la búsqueda binaria para localizar causa del fallo [Zeller and Hildebrandt, 2002]. Despˇes, Gupta *et al.* combinó el Delta Debugging con la técnica de rebanamiento estático para identificar el conjunto de declaraciones que probablemente contengan código defectuoso [Gupta et al., 2005]. Finalmete, Cleve y Zeller [Cleve and Zeller, 2005] describieron y usaron la técnica de transiciones de causa *Cause Transitions* para comparala con el Nearest-Neighbor. Sus resultados sugieren que, bajo el

mismo conjunto de sujetos, se comporta mejor la técnica de Cause Transitions.

B.2.3 El uso del Algorithmo SZZ:

Sin tratar de ser exhaustivos, en esta sección ofrecemos una serie de artículos donde los autores han usado el algoritmo SZZ para diferentes fines. A continuación se muestran cinco diferentes categorías dependiendo del propósito del artículo en el que se usó el SZZ.

Prediccioón de Errores Feng *et al.* usaron el SZZ para recolectar datos defectuosos y construir un modelo de predicción de errores universal [Zhang et al., 2014]. Jiang *et al.* propuso una nueva técnica para predecir errores en futuros datos, esta técnica produce un modelo personalizado para cada desarrollador [Jiang et al., 2013]. Hata *et al.* desarrollo un sistema de control de versiones detallado para Java, con el finde llevar a cabo predicciones mas exhaustivas. Los autores coleccionaban los módulos erróneos usando el algoritmo SZZ [Hata et al., 2012]. Kim *et al.* analizó la historia de versiones de siete proyectos para predecir los ficheros y entidades más propensos a errores [Kim et al., 2007]. Zimmermann *et al.*, también usó el algoritmo para predecir errores en grandes sistemas como Eclipse [Zimmermann et al., 2007]. Nagapan *et al.* usó el SZZ para predecir la probabilidad nuevas entidades defectuosas tras la distribución de la nueva versión del software [Nagappan et al., 2006]. Yang *et al.* utilizaron el algoritmo SZZ para estudiar la probabilidad de que un cambio que arregló un error, a su vez introdujese otro error en el futuro [Yang et al., 2014]. Rosen *et al.* desarrolló una herramienta basada en el estudio del algoritmo SZZ que identifica y predice los cambios más peligrosos en el software, ya que pueden provocar errores en el futuro [Rosen et al., 2015]. Kamei *et al.* introdujeron el concepto just-in-time (JIT) para asegurar una mayor calidad del software, los autores usaron este concepto para construir un modelo que predice si un cambio es probable que introduzca errores [Kamei et al., 2013].

Clasificación de errores Pan *et al.* usa varias métricas obtenidas aplicando la técnica del "slicing"[1] para clasificar cambios como erróneos o libres de error, los cambios erróneos son identificados usando el SZZ [Pan et al., 2006]. Kim *et al.* estudió cómo clasificar los cam-

[1]slicing hace referencia a cada una de las "rebanadas" del software, esta técnica es usada para identificar todo el código fuente de un programa que puede afectar de algún modo el valor de una variable dada.

bios en archivos como defectuosos o libre de errores usando las características de los cambios realizados en el código fuente para arreglar el error [Kim et al., 2008]. Thomas *et al.* introdujeron un marco diseñado para combinar múltiple configuraciones de clasificadores, mejorando de esta manera el rendimiento del mejor clasificador [Thomas et al., 2013]. Ferzund *et al.* presentaron una técnica para clasificar los cambios realizados en el software como libre de errores o erróneos, usaban el SZZ para identificar de los trozos de código que fueron modificados anteriormente y que introdujeron errores [Ferzund et al., 2009]. Kim y Ernst propusieron un algoritmo de priorización de alertas basado en la historia de control de versiones de un proyecto para ayudar a mejorar la priorización de herramientas de búsqueda de errores. Este algoritmo se basa en SZZ para identificar líneas de archivos relacionadas con errores [Kim and Ernst, 2007].

Localizacion de errores Asaduzzaman *et al.* usaron el algoritmo SZZ en Android para identificar los cambios que introdujeron errores para después, usar esa información y buscar problemas de mantenibilidad del proyecto [Asaduzzaman et al., 2012]. Schröter *et al.* construyeron un conjunto de datos para el proyecto Eclipse que contiene información sobre los cambios que introdujeron errores y los cambios que arreglaron esos errores [Schröter et al., 2006]. Kim *et al.* desarrollaron una herramienta para encontrar fallas usando la memoria de los arreglos de errores, este método se centra en el conocimiento sobre los cambios que arreglaron los errores. La herramienta usa análisis estadístico para aprender patrones de error específicos de un proyecto mediante el análisis de la historia de un proyecto para luego sugerir correcciones [Kim et al., 2006a]. Wen *et al.* propusieron otra herramienta para localizar errores basada en la recuperación de información y utiliza el algoritmo SZZ para extraer los cambios que introdujeron errores y evaluar de este modo la herramienta [Wen et al., 2016].

Entender la Evolución del Software Kim y Whitehead analizaron los proyectos ArgoUML y PostgresSQL para calcular el tiempo que tarda un error en ser arreglado después de haber sido introducido en el código fuente [Kim and Whitehead Jr, 2006]. También, Kim *et al.* estudió las propiedaddes y la evolución de los patrones de cambios realizados en siete diferentes sistemas de software escritos en C, usaban el SZZ para identificar los cambios que introdujeron errores [Kim et al., 2006b]. Eyolfson usa el SZZ para estudiar si el momento del día en

el que se introduce un cambio y la experiencia del desarrollado que lo introduce, afecta a la probabilidad de introducir más errores en el código [Kamei et al., 2010]. Izquierdo *et al.* usa el algoritmo SZZ algorithm para estudiar si los desarrolladores arreglan los errores que han introducido [Izquierdo et al., 2011], además, los autores también usan este algoritmo para estudiar la relación entre la experiencia de los desarrolladores y la introducción de errores en la comunidad de Mozilla [Izquierdo-Cortázar et al., 2012]. Rahman y Devanbu estudian los factores que tienen más impactyo en la calidad del software, como la propiedad del código, la experiencia, la estructura organizacional y la distribución geográfica. En este estudio, el algoritmo SZZ se usó para identificar las líneas de código asociadas con los cambios que introdujeron los errores [Rahman and Devanbu, 2011].

Estudios Empíricos Nguyen y Fabio Massacci realizaron un estudio para validad la confiabilidad de los datos de la versión vulnerable de NVD. Los autores usaron el algoritmo SZZ para identificar el código vulnerable responsable de la vulnerabilidad [Nguyen and Massacci, 2013]. Bavota *et al.* estudiaron empíricamente en qué medida las actividades de refactorización introdujeron errores en tres sistemas que usaban Java como lenguaje de programación [Bavota et al., 2012]. Kamei *et al.* utilizaron el algoritmo SZZ para revisar los modelos de predicción de errores en la literatura [Kamei et al., 2010]. Fukushima *et al.* evaluó empíricamente el rendimiento de los modelos de predicción de errores basados en el concepto Just-In-Time [Fukushima et al., 2014].

B.3 Modelo teórico Propuesto para localizar errores

Esta sección presenta la definición de un modelo para identificar inequívocamente los cambios que introducen errores (*BIC*). Éste modelo identifica un conjunto de cambios que introducen errores y que se corresponden con un conjunto de cambios que arreglan errores (*BFC*). Además, este modelo incluye definiciones precisas de lo que significa un *BFC* y un *BIC*, definiciones basadas en la suposición de que existe una prueba hipotética con una cobertura de test perfecta y que podría ejecutarse indefinidamente a lo largo de la historia del código fuente del proyecto. El modelo propuesto devuelve verdadero o falso dependiendo de si el error estuvo presente o no en un momento específico del proyecto, de tal manera que podemos usar la prueba o test para comprobar si una determinada funcionalidad esta presente y se com-

porta de acuerdo a las expectativas en cada momento de la historia del proyecto. Por primera vez, un modelo contempla diferentes escenarios que han sido excluidos en la literatura actual, por ejemplo, los *BFC* que presentan solamente nuevas lineas añadidas, o la distinción entre el momento de inserción y el momento de manifestación de un error.

El objetivo principal para describir este nuevo modelo es ampliar las técnicas actuales usadas en la líteratura y garantizar de esta forma que las cambios identificados como responsables de introducir el error en el código fuente en realidad introdujeron el error en algún momento de la historia del proyecto. Finalmente, éste modelo se usará como un marco teórico para la evalución del rendimiento de otras técnicas en la identificación de *BIC*, así como para comparar la efectividad entre los diferentes algoritmos, ya que junto con el modelo propuesto se define un *"estándar de oro"* en que se identifican inequívocamente que cambios son *BIC* en un proyecto. Antes de explicar en detalle la teoría del modelo propuesto, se requiere definir algunos conceptos que sob utilizados para explicar el modelo.

B.3.1 Definiciones:

Buscar el origen del error es una tarea compleja que implica una gran variedad de elementos y conjuntos individuales en su búsqueda. De tal manera que, encontramos la necesidad de formular una terminología que es una de las partes de valor en esta tesis. Esta terminología se puede aplicar a los sistemas modernos de administración de código fuente. La terminología define cada elemento y cada conjunto de elementos que tienen lugar durante el análisis para la identificación del *BIC* a partir de un *BFC*. Para evitar confusiones, a continuación se definen los conceptos con los que vamos a trabajar:

Cambio atómico *(at)*: Es una operación que aplica un conjunto de cambios distintos como una operación única. En esta tesis, el cambio atómico se refiere a un cambio mínimo de una línea.

Cambio atómico anterior: Dado un cambio atómico at, nos referimos a at' como la última modificación que cambió la línea l de un fichero f. Por lo tanto, la relación de precedencia entre un cambio atómico y su cambio atómico previo es la siguiente:

$$at' \rightarrow at$$

Commit *(c)*: Un cambio observable que registra uno o más cambios atómicos en el código fuente de un sistema de software. Estos cambios generalmente se resumen en un parche que es un conjunto de líneas que un desarrollador agrega, modifica o elimina en el código fuente. Los commits actualizacian la versión actual del directorio árbol.

Líneas modificadas: Por definición, una confirmación puede cambiar cero[2] o más líneas de código; y nos referimos a ellas como *líneas cambiadas* de un commit y las denotamos como $LC(c)$.

Precedencia entre commits: Relación entre *cambios atómicos* de un commit con sus *cambios atómicos previos* en un fichero f, dónde el commit previo de un commit es el cambio atómicos anterior de un conjunto de cambios atómicos. Nos referimos a esta precedencia entre commits como *commit previo(*pc*)* de una commit en f y la designaremos como:

$$pc'(c) \rightarrow pc(c)$$

paragraph textbf Conjunto de commits previos: Conjunto que incluye los diferentes commits previos a un commit. Formalmente:

$$PCS(c) = \bigcup pc'(c)$$

Commit Descendiente: Dado un commit c y un fichero f, una confirmación descendiente de c es una de las confirmaciones que pertenece a la cadena de commits de precedencia de c en f, nos referiremos a él como dc.

Set de Commits Descendientes: Conjunto de commits descendientes para un commit determinado; nos referimos a él como $DCS(c)$. Debemos tener en cuenta que *set de commits previos* contiene solo los commit previos a un commit, mientras que el *set commits descendientes* contiene todos los commits que han modificado, de alguna manera, las líneas cambiadas en c durante toda la historia del fichero f.

Ancestor Commit: Es cualquiera de los commits anteriores a un commit determinado, nos referiremos a él como ac.

[2]se cambian cero líneas cuando solo se agregan nuevas líneas en un commit.

Conjunto de Ancestor Commits: Conjunto de los ancestor commits de un commit determinado; lo llamamos $ACS(c)$. Debemos tener en cuenta que a partir de un commit específico del repositorio, el *conjunto de ancestor commits* contiene todos los commits de ese repositorio.

Inmediatamente Ancestor Commit: Es el commit inmediatamente anterior a un commit determinado en el conjunto de ancestor commits, nos referiremos a él como iac.

Instantánea o Snapshot: Representa el estado completo del proyecto en algún momento de la historia. Usando git como ejemplo, dado un commit c, la instantánea correspondiente sería el estado del repositorio después de escribir en la terminal de comandos "git checkout c". La evolución del software se puede entender como una secuencia de instantáneas, cada una correspondiente a un commit, en el orden mostrado por " git log" (orden de confirmaciones en la rama considerada).

Commit que arregla un error (*BFC*): Commit dónde se solucionó un error. Dado un error b se puede requerir uno o más commits para reparar el error, definimos el conjunto de commits que arreglan un error como el siguiente conjunto: $BFC(b)$. En general, esperamos que este conjunto sea único, es decir, que un error se corrija en un único commit, a pesar de que se pueden necesitar varios commits para corregir un error. Además, un commit que corrige un error solo existe si realmente es un error en el momento de la corrección, porque para descubrir qué cambio introdujo el error, es necesario que el error analizado sea realmente un error.

Snapshot de la solución del error (*BFS*): instantánea del código correspondiente a un *BFC*.

Test Signaling a Bug (*TSB*): Una prueba utilizada para indicar que hay un error presente. Se define como una prueba hipotética, que se puede ejecutar en cualquier instantánea del código, devolviendo *True* si la prueba pasa, lo que significa que la instantánea no contiene el error, y *False* si la prueba no pasa, lo que significa que la instantánea contiene el error.

Prueba que falla en la snapshot *(T-S)*: instantánea para la cuál el *TSB* falla.

Prueba que pasa en la snapshot *(T + S)*: instantánea por la cuál el *TSB* pasa.

Snapshot introductor de errores (*BIS*): Primera instantánea en la secuencia continua más larga de T-S, que termina justo antes de *BFS*. Es decir, hay una secuencia continua de instantáneas para las que falla la prueba, comenzando en el *BIS* y terminando justo antes del *BFS*. Como la prueba falla desde la instantánea hasta la corrección, podemos saber que la prueba falló en esa secuencia y, dado que esta es la primera instantánea con el fallo de la prueba, podemos decir que ésta es la primera instantánea "con el error presente".

Cambio que introdujo el error (*BIC*): Un commit específico perteneciente al *BIS* que introdujo la(s) línea(s) errónea(s) en el momento de su inserción, y el error se propagó en el sistema a través de los commits posteriores hasta que el *BFC* arregló el error.

First Failing Moment (*FFM*): El primer commit correspondiente al *BIS* que manifiestó el error pero que no introdujo líneas erróneas en el momento de su inserción.

B.3.2 Explicación del modelo propuesto:

Con demasiada frecuencia, cuando se analizan proyectos, los investigadores usan `git` como sistema de gestiín de código fuente *(SCM)*. El SCM registra *cambios observables* en un archivo o conjunto de archivos. Los cambios observables son alteraciones de archivo(s) causados por adiciones, eliminaciones o modificaciones de una o más líneas en el código fuente. Gracias al SCM, los investigadores pueden rastrear de forma manual o automática la eliminación y modificación de líneas desde un momento específico hasta su origen, y también pueden identificar qué líneas son nuevas adiciones en cada commit. Navegando hacia atrás en la relación entre las líneas alteradas de cada cambio observable y su cambio anterior, se puede construir un *árbol de precedencia de cambios observables* o *árbol genealógico*. La figura B.1 muestra un ejemplo del cambio observable i realizado en un ficherp f que corrigió el error b. Rastreando cada línea modificada o eliminada de f, podemos dibujar el árbol genealógico de las líneas involucradas en i. Es importante saber que las nuevas líneas añadidas

no se pueden rastrear, pero se tiene en cuenta en el modelo. Los recuadros negros representan los diferentes commits, los puntos representan un conjunto de solo nuevas líneas, las flechas muestran la precedencia entre commits, y el color de las líneas se relaciona con la acción realizada, eliminar (rojo), añadir (verde) o modificar (negro).

Usando la terminología definida anteriormente, nos vamos a referir al cambio observable i como el *BFC*. A partir de las líneas cambiadas en el *BFC*, $LC(BFC)$, podríamos dibujar su árbol genealógico en el que, gracias a la precedencia entre los commits, hay una relación genealógica. Visualmente a partir de esta relación, distinguimos entre commits de la primera generación *(i-1a, i-1b, i-1c)*, segunda generación *(i-2a, i-2b, i-2c, i -2d)*, y tercera generación *(i-3a)* del *BFC*. Por extensión, el conjunto de commits previos del *BFC* es la primera generación de commits y el conjunto de commits descendientes es la primera, segunda y tercera generación de los commits.

$$PCS(\textbf{\textit{BFC}}) = (i-1a, i-1b, i-1c)$$

$$DCS(\textbf{\textit{BFC}}) = (i-1a, i-1b, i-1c), (i-2a, i-2b, i-2c, i-2d), (i-3a)$$

Cuando se observan los commits en la rama *master* de un repositorio de proyectos, es posible que no se tenga un claro acceso visual al árbol genealógico de una commit determinada, pero se tiene una visión aplanada de todas los ancestor commits de un commit determinado. En esta visión plana, los commits van precedidos por otros cambios que conforman una visión linearde precedencia, donde se pueden encontrar los commits genealógicos. Un concepto importante a tener en cuenta, es que esta precedencia no está establecida por fechas, sino por versiones anteriores en el SCM. Esto ocurre debido a la forma en que funciona un sistema descentralizado de gestión de código fuente (DSCM). Bird *et al.* Explicó cómo podrían divergir los repositorios locales de dos desarrolladores que colaboran a la vez en un proyecto que usa git, lo que hace que cada repositorio contenga commits que pueden no están presentes en el otro repositorio. Por lo tanto, en el momento de combinar ambos repositorios locales, el usuario puede seleccionar entre muchas opciones relacionadas con la secuencia de commits, tales como rebase, merge, remove, squash, etc. Estas acciones pueden alterar el orden natural de los commits, lo que las inhibe de ser ordenados cronológicamente en el tiempo [Bird et al., 2009b]. Continuando con el ejemplo anterior, La Figura B.2 muestra la visión linearde precedencia del commit i en la rama *master* del repositorio de un proyecto.

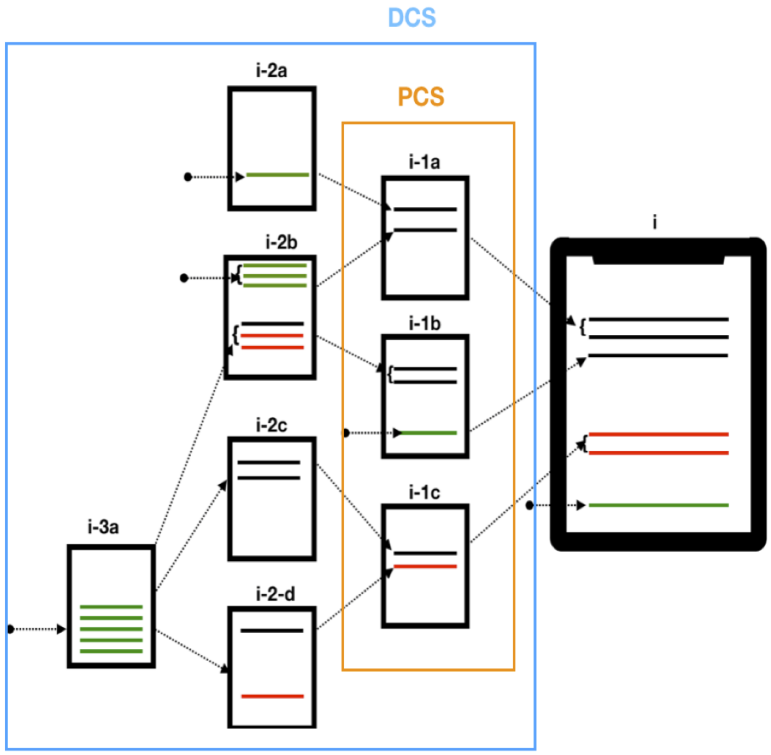


Figure B.1: Árbol genealógico del cambio observable i, cada confirmación muestra una relación de precedencia con sus compromisos descendentes.

Los commits están representados por círculos, y los cambios que pertenecen al árbol genealógico de i se pueden encontrar en naranja si son pc o en azul o si son dc; los commit restantes son los ac dónde el commit anterior al i es el inmediatamente ancestor commit iac. El conjunto de *ACS* fueron añadidos al proyecto; sin embargo, no tienen una relación de precedencia con las líneas modificadas en i y no aparecen representados en la figura.

Para aquellos que están familiarizados con `git`, podemos comparar la Figura B.1 con el resultado obtenido tras usar `git blame` en las líneas modificadas en *BFC* y la Figura B.2 se correspondería con el resultado de usar `git log` en una confirmación específica de la rama principal de un repositorio en un proyecto.

Desde un punto de vista objetivo, no es importante saber *CUÁNDO* se insertó el error en el tiempo, sino *QUÉ* lo insertó. Esto se debe a que después de insertar las líneas erróneas en commit previo o una ancestral commit de un *BFC*, el error permanece en el sistema y, además, se propaga a través de cada cambio nuevo en el fichero. Por lo tanto, determinar el primer cambio que manifiesta el error implica navegar en las líneas del árbol genealógico

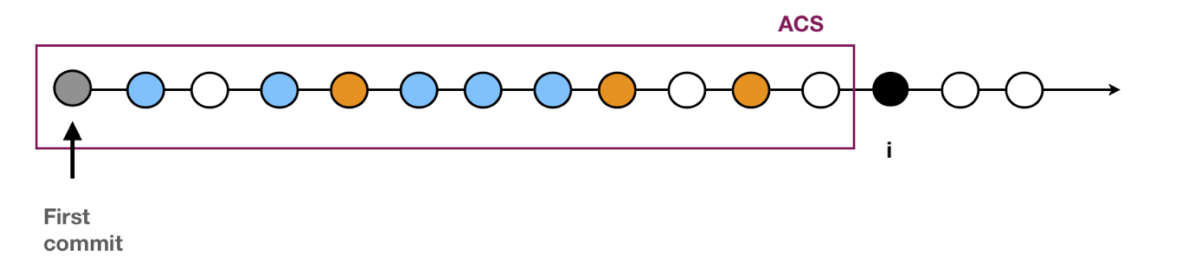


Figure B.2: Visión de precedencia linearen la rama master del commit que arregla el error. Los commits coloreados pertenecen a $PCS(i)$ (naranja) y *DCS*i (azul), el commit negro es *BFC* y el commit gris es el commit inicial del proyecto. Se debe tener en cuenta que los commits no se ordenan en orden cronológico porque no asumimos una prioridad establecida por fechas.

y desde un punto de vista teórico, habrá un cambio en la. visión lineardе precedencia que manifiesta el error en el sistema por primera vez. Este cambio será el primer cambio que falla y se denominará *FFM*. Además, el *FFM* puede ser el cambio que introdujo el error en función de si la(s) línea(s) insertadas contenían el error. Cuando ningún cambio insertó la(s) línea(s) erróneamente, no se puede encontrar el *BIC* y como resultado, solo se puede identificar el *FFM* para explicar en que commit falló por primera vez el sistema. Algunos ejemplos son cuando cambios (externo/interno) afectaron de alguna manera las línea(s) del código fuente causándo el error del sistema y su manifestación.

B.4 Como encontrar el *BIC* y el *FFM*

Para descubrir el cambio que introdujo el error con la mayor precisión posible, se recomienda hacerlo manualmente rastreando cada línea modificada del código fuente de un *BFC* hasta encontrar el *momento* donde se insertó el error. Es necesario utilizar la información del sistema de revisión de código y del sistema de control de versiones para asegurarse que en ese momento se insertó el error. Si en base a esta información se observa que en ese momento el commit insertó la(s) línea(s) que contenían el error, el cambio se considera un *BIC*. Por el contrario, si la información recopilada explica que hubo otro cambio que causó el error en el sistema como por ejemplo un cambio en *el entorno o en el contexto*, el cambio no se considera un *BIC*, si no que se considera un *FFM*.

Teóricamente y bajo las condiciones óptimas, este proceso se podría automatizar. Para encontrar el momento en el que se introdujo un error, se usaría una prueba de señalización de un error (*TSB*) que tendría como resultado *True* cuando la instanánea en la que se prueba el test pasa, y *False* cuando la prueba falla en las instantánea analizada del proyecto. A pesar de la alta complejidad para automatizar este proceso, existe una manera fácil de encontrar el *BIC* o *FFM* usando el modelo propuesto y se basa en buscar manualmente la primera instantánea que no pasa el *TSB*. Esta instantánea contiene el commit que será el candidato perfecto para ser el *BIC* o el *FFM*.

El modelo propuesto se centra en identificar el *BIC* para un *BFC* dado. Para mostrar cómo funciona, se aplican las definiciones de *BIC* y *BFC* basadas en la existencia de un *TSB*. El *TSB* se aplica en todas las instantáneas del proyecto anteriores al *BFC* para identificar si hay un commit que introdujo el error arreglado en en *BFC*. Teniendo en cuenta que *TSB* tiene una cobertura del 100% y que puede ejecutarse indefinidamente a lo largo de la historia del código fuente de un proyecto, el modelo propuesto puede averiguar el *BIC* o el *FFM* de un *BFC* analizando los cambios realizados para arreglar el error. Este *TSB* se ejecuta en todo el conjunto de commits antecesores de las líneas modificadas en el *BFC*. Por lo tanto, cuando la prueba *TSB* busca la instantánea que falla por primera vez; si se encuentra, el modelo la considerará como candidato para ser el *BIC*.

B.4.1 Resultados del *TSB*

El resultado del *TSB* varía dependiendo de cada instantánea, y existen tres posibles resultados al ejecutar el *TSB*:

1. **Pasa**: La función o característica probada está presente en la instantánea y funciona como la prueba esperaba, no hay *BIC*.

2. **Falla**: La función o característica probada está presente en la instantánea pero la prueba no funciona como se esperaba. Esta instantánea se considerada una candiata a *BIC*.

3. **No-ejecutable**: La función o característica probada no está presente en la instantánea y por tanto la prueba no puede ejecutarse en la instantánea.

Hay tres escenarios diferentes que ilustran cómo aplicar el test hipotética en el conjunto

de $ACS(i)$ para identificar si existe el *BIC*. En estos escenarios, se considera que la *TSB* tiene un 100% de cobertura y puede ejecutarse indefinidamente a lo largo de la historia del código fuente. Por lo tanto, el *TSB* se pasa a todas las instantáneas previas al *BFC* para buscar la instantánea que falla; si se encuentra, se considerará como candidato a ser el *BIC*.

La Figura B.3 muestra cómo aplicar la prueba hipotética cuando hay existe un *BIC* y el *TSB* se puede ejecutar en todas las instantáneas previas al *BFC*. Para identificar el *BIC*, el *TSB* se ejecuta en todas las instantáneas del conjunto de commits ancestrales, y el *BIS* será la primera que falle (i-1b). Se puede saber que el *BIS* es un *BIC* porque pasa la instantánea anterior (i-2c).

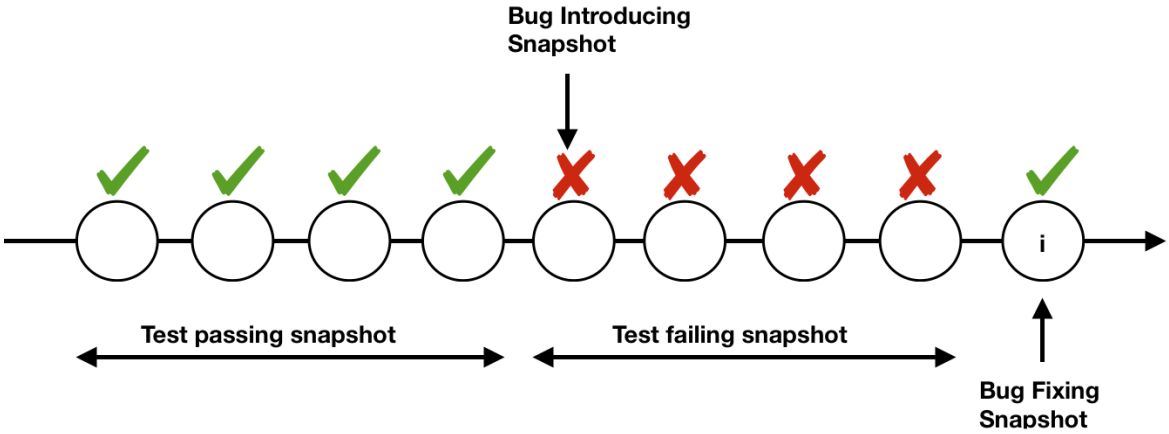


Figure B.3: El Bug Introducing Snapshot es el *BIC*

La Figura B.4 muestra cómo se aplica la prueba hipotética cuando existe un *BIC* pero la prueba *TSB* no se puede ejecutar después de una instantánea. Esto se debe a que la función o característica probada no se encuentra presente en ese momento. En este caso, el primer *BIS* identificado es el *BIC* porque cuando introdujo la función o característica probada, ésta contenía errores.

La figura B.5 muestra cómo se aplica la prueba hipotética cuando no hay *BIC* y el *TSB* se puede ejecutar indefinidamente a lo largo del historial del código fuente. La *TSB* siempre falla con el entorno *BFS* en las instantáneas, pero si se establece el entorno anterior, pasará en la instantánea descendente. Por lo tanto, el primer *BIS* antes de *BFC* será el *FFM*, no existe *BIC* en el conjunto de commits antecesores.

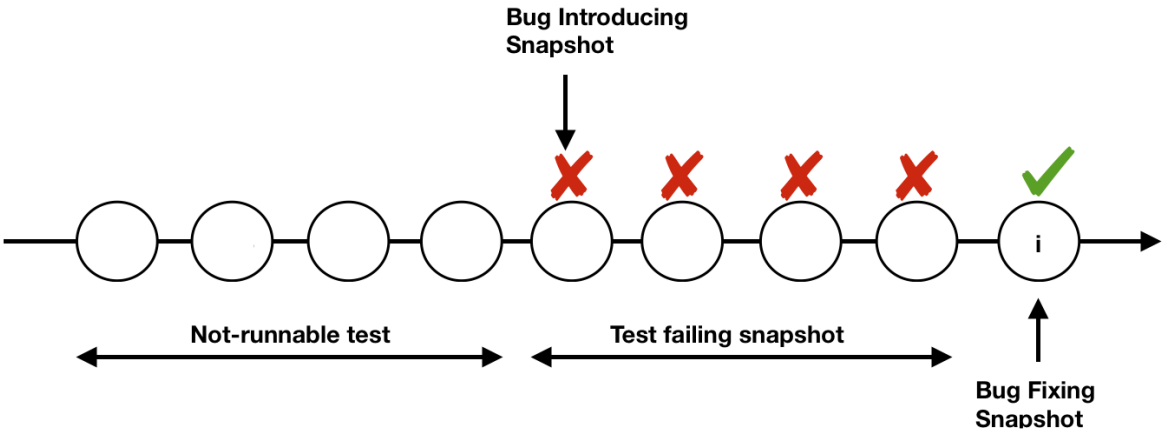


Figure B.4: El Bug Introducing Snapshot es el *BIC*

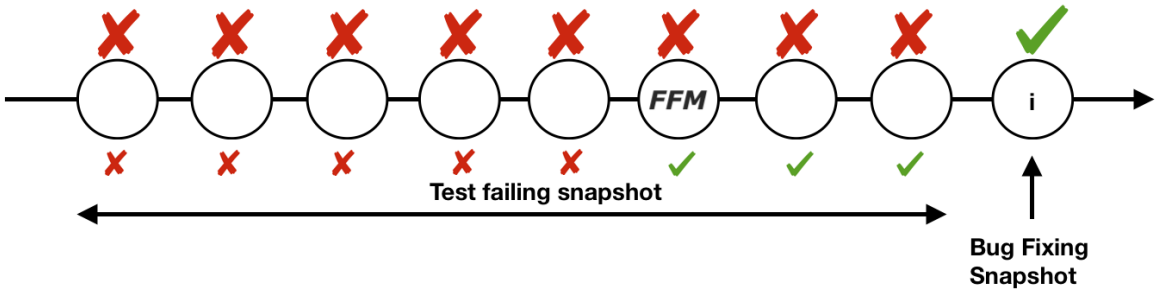


Figure B.5: El Bug Introducing Snapshot es el *FFM*

B.4.2 Criterio para aplicar el *TSB*

Cuando la historia de un proyecto es lineal, los investigadores pueden usar uno por uno todo el conjunto de commits pertenecientes al $ACS(i)$ para ejecutar el *TSB* en la rama master. Pero cuando la historia del proyecto no es linearsi no que puede tener múltiples ramas, encontrar el *BIC* o *FFM* puede llegar complicarse. Esto se debe a que el error podría estar presente en algunas de las ramas existentes y ausente en otras, y el concepto de *FFM* puede llegar a ser incierto. Sin embargo, a primera vista, y suponiendo que teóricamente la prueba se puede ejecutar para todas las instantáneas en el $ACS(i)$, los resultados del *TSB* aún podrían ser válidos para identificar el *FFM* o *BIC* ya que todavía encuentran secciones en una o más ramas donde la prueba fallaría.

Es posible que, en algunos escenarios, el modelo propuesto defina el *BIC* como el topológicamente primero en la sección de múltiples instantáneas con errores. Puede suceder que dos o más

commits en paralelo comienzan a fallar hasta llegar al *BFC*, sin embargo, se puede considerar que esto es una indicación de que el error fue introducido en el código fuente, independientemente, en varias ramas o fue copiado (o escrito idénticamente por casualidad) en varias ramas. Nuestra hipótesis es que estos escenarios no son comunes y por el momento podríamos no centrarnos en ellos, pero al menos debemos mencionarlos en el modelo teórico.

Por lo tanto, para definir estos casos, podríamos extender la noción de *BIC* a "el conjunto de *BIC*, que sería "los cambios correspondientes a la primera instantánea que falla, continuamente hasta llegar al *BFC*, en varias ramas que conducen hasta el *BFC*".

Además, los investigadores pueden decir el tipo de *BIS* en base a si es un *BIC*, un *FFM*, ambos, o indecisos.

1. **Indeciso**: Cuando no podemos encontrar el *FFM* o el *BIC*. En esta situación, no podemos estar seguros de encontrar el *FFM* entre todos los commits pertenecientes al conjunto $ACS(b)$ usando el *TSB*

2. **El *FFM* es el *BIC***: Cuando podemos encontrar el *FFM* entre todos los commits del conjunto $ACS(b)$ usando la prueba *TSB* y además podemos estar segurod de que ese commit es la causa del error porque introdujo las líneas erróneas en el código fuente.

3. **solo es un *FFM***: Cuando el error no fue causado por un cambio perteneciente al conjunto $ACS(b)$ si no que, un cambio en el entorno o un cambio en las dependencias del proyecto causó el error y usando el *TSB* podemos encontrar la primera instantánea que manifiesta el error.

Automatizar este proceso es tedioso y complejo, debido a la necesidad de recrear todas las dependencias externas utilizadas en el proyecto en cada una de las instantáneas previas a un *BFC*. Sin embargo, seguimos confiando en que la automatización puede ser posible en base a declaraciones de estudios anteriores como el realizado por Bowes *et al.*. Este estudio proporciona una lista básica de principios de prueba que se centra en diferentes aspectos de calidad, además de la efectividad o la cobertura. Para nuestra investigación, el principio más interesante es la (in)dependencia de prueba que describe que una prueba debe poder ejecutarse en cualquier orden y de forma aislada. La prueba no debe depender de otras pruebas de todos modos, para permitir a los profesionales agregar nuevas pruebas sin tener en cuenta las dependencias o los efectos que puedan tener en las pruebas existentes [Bowes et al., 2017].

B.5 Objetivos y Problema

El objetivo principal de esta tesis es desarrollar un modelo teórico que ayude a identificar inequívocamente el cambio que introdujo la línea o líneas erróneas en el código fuente de un programa, a partir del arreglo del error. Para ello, esta tesis realiza un análisis minucioso sobre las técnicas actuales usadas para identificar la causa del error, detallando el problema actual que impide que estas técnicas identifiquen correctamente el cambio en el que el error fue introducido en el código fuente. En concreto, esta tesis cuantifica las limitaciones encontradas en el algoritmo SZZ. Desde hace mas de diez años, este algoritmo es el más utilizado para encontrar el origen del error a pesar de que tanto investigadores como desarrolladores y profesionales son conscientes de sus limitaciones. La falta de otro modelo que explique como encontrar el verdadero origen de un error, así como la falta de definición de lo que significa un fallo, causan que todos los estudios que analizan el momento en el que un error es introducido empiecen con la misma premisa "Las líneas modificadas que arreglaron el error son potencialmente culpables de introducir el fallo en el sistema". Esta premisa trata a los errores como una variables estáticas que permanece en el sistema desde que se introducen, cuando en realidad los errores no deben ser estudiados como variables estáticas, ya que factores externos como cambios el las APIs o la evolución interna del sistema causan que una línea correcta en un un momento determinado empiece a manifestar el error en un momento posterior.

Para resolver la falta de definiciones y las limitaciones yacentes en los algoritmos actuales. Esta tesis propone un modelo teórico para definir qué cambios introducen errores en el código fuente. Este modelo se basa en la suposición que existe una prueba de test perfecta que puede ejecutarse indefinidamente en toda la historia del proyecto con el fin de descubrir cuándo se insertó el error en el código fuente del proyecto. Además, este modelo puede usarse como marco para validar los resultados obtenidos en otrros enfoques. Establecer este marco es uno de los principales valores de la tesis porque la literatura previa ha podido calcular con exactitud la precisión y el recall "real" de los algoritmos actuales usados para identificar el *BIC*.

A continuación se puede encontrar un listado de las contribuciones principales de la tesis:

1. Revisión de la literatura asistemática en el uso del algoritmo SZZ:

   (a) Una visión general del impacto que el algoritmo SZZ ha tenido hasta ahora en el área de ESE.

   (b) Una visión general de cómo los estudios que usan el algoritmo SZZ abordan la reproducibilidad en su trabajo de investigación.

   (c) Un análisis de cómo estos estudios manejan las limitaciones del algoritmo SZZ

2. Un modelo teórico para identificar inequívocamente los cambios de introducción de errores

   (a) Una definición detallada del Bug-Introducing Change y First-Failing Change

   (b) El criterio usado para aplicar el modelo.

3. Estudio empírico sobre la aplicación del modelo propuesto

   (a) La frecuencia de *BFC* inducida por *BIC* en Nova y ElasticSearch

   (b) El conjunto de datos "estándar de oro"

4. Cuantificación del algoritmo SZZ.

B.6 Metodología

La metodología utilizada en este estudio empírico se divide en dos partes, la primera parte es la etapa de filtrado donde los investigadores se aseguran que los conjuntos de datos extraídos de Nova y ElasticSearch cumplen con los requisitos para ser analizados usando la teoría de la introducción de errores descrita anteriormente. La segunda etapa describe cada uno de los pasos que hay que seguir desde que se selecciona un *BFC* hasta que se identifica el *BIC* y el *FFM*.

La figura B.6 proporciona una visión general de cada paso involucrado en nuestro estudio, así como los resultados que se obtendrían.

B.6.1 Primera Etapa: Filtrado

Esta etapa garantiza que el modelo propuesto se pueda aplicar a los informes de errores extraídos anteriormente. Uno de los requerimientos necesarios en esta etapa es que los in-

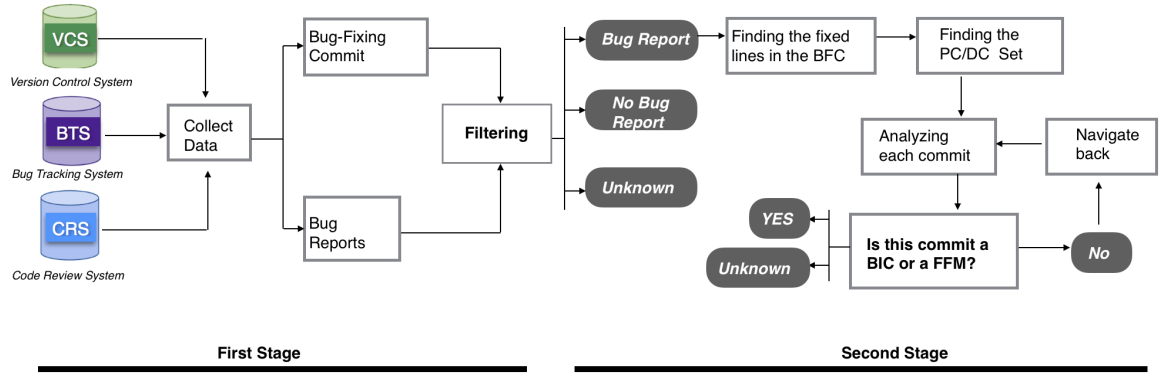


Figure B.6: Descripción de los pasos involucrados en nuestro análisis

formes de error que pasen a la siguiente etapa describan realmente errores, y a pesar de la estricta política de etiquetado de errores presente en ElasticSearch y la clasificación manual para distinguir informes de errores de otro tipo de errores llevada acabo en Nova, esta etapa se encarga de analizar cuidadosamente cada información en los informes de errores para garantizar que el "estándar de oro" obtenido al final del estudio es lo más preciso posible. Consecuentemente, el conjunto de datos tanto en Nova como en ElasticSearch, solo debe almacenar errores que se consideran errores en el momento de su correción. Por ejemplo, hay una prueba (hipotética) que falla justo antes del *BFC* pero no falla inmediatamente después. Esto asegura que si existen problemas que se describen como errores pero se descubre que son nuevas peticiones para añadir nuevas características o sugerencias de mejoras, serán excluidas.

B.6.2 Segunda etapa: identificar el *BIC* y el *FFM*

Esta etapa tiene como entrada un conjunto de informes de errores de Nova y ElasticSearch que describen errores reales en el momento de su corrección. En esta etapa se require identificar manualmente el *BIC* y el *FFM* correspondiente a un *BFC* dado, aunque en algunas ocasiones, no se puede identificar ningún *BIC* como el inductor del *BFC*. La identificación de un *BIC* dado un *BFC* significa que el error estaba presente en el momento de escribir las líneas en el código fuente. El *BIC* puede estar en el commit previo o en cualquiera de los commits descendientes o antecesores, por tanto se requiere analizar manualmente cada uno de ellos para garantizar la credibilidad en el "estándar de oro". En los casos en que el error no está

presente en el momento de insertar las líneas en el código fuente, significa que no podemos identificar ningún *BIC* y es necesario encontrar el *FFM*. Para entender mejor este proceso, los pasos seguidos se detallan a continuación donde se hace uso de la terminología que se encuentra descrita en la sección 5.1.1.

Encontrar las líneas que corrigieron el error

En esta etapa, se debe identificar las líneas de código fuente que corrigió el error; nos referimos a un error genérico como b. Es importante:

- Identificar los commits que corrigieron el error entre el conjunto $BFC(b)$. En la mayoría de los casos analizados, este conjunto siempre fue unitario, lo que significa que para cada error había un *BFC* único que lo arreglaba. Sin embargo, encontramos algunas excepciones, por ejemplo, el informe de error #1442795[3] tenía dos *BFC* diferentes que corrigieron el error. Para simplificar este proceso, la metodología asume que el conjunto $BFC(b)$ es unitario, y en el caso donde más de un *BFC* corrigió el error, la metodología analizará ambos commits indistintamente para identificar el *BIC*.

- Encuentra las líneas que fueron modificadas por el $BFC(b)$ para corregir el error, nos referiremos al conjunto de líneas añandias, modificadas y eliminadas por un commit como $LC(c)$ donde c es el *BFC*: El *BFC* se encuentra vinculado a un informe de error, en este informe se encuentra disponible toda la información relacionada con el proceso de revisión del código. Aplicando la herramienta *"git diff "* es posible identificar qué líneas se han agregado, modificado o eliminado entre la versión después del *BFC* y la anterior. También existe la posibilidad de visualizar los cambios realizados por el *BFC* usando la herramienta web de GitHub donde visualmente se observan de una forma muy intuitiva los cambios realizados en el *BFC*.

- Filtrar las líneas modificadas o eliminadas que no son código fuente: Algunas líneas que se han modificado o eliminado pueden no contienen código fuente (por ejemplo, comentarios o líneas en blanco). Éstas líneas no se consideran en el posterior anális. Además, como caso excepcional, en el conjunto de *BFC* analizados se encontraba un

[3] `https://bugs.launchpad.net/nova/+bug/1442795`

BFC[4] que arregló dos informes de errores diferentes al mismo tiempo. En este caso, se eliminaron el conjunto de $LC(c)$ relacionadas con el informe de error que no se estaba analizando.

.

Determinar qué commit cambió por última vez cada una de las líneas $LC(c)$.

Cada una de las líneas modificadas por un *BFC* tiene un único commit previo. Sin embargo, podría haber tantos commits previos como líneas modificadas en por un *BFC*, (la figura del árbol genealógico representado en la subsección 5.1.2 muestra visualmente este concepto). Consecuentemente, el resultado obtenido en este paso es un conjunto de todos los commits previos del error b, $PCS(b)$.

Analizando cada una de los commits previos, y sus commits descendientes, para identificar el *BIC* y el *FFM*

En este paso se utiliza la información disponible en la descripción del ticket, en los registros del *BFC* y en commits pertenecientes al conjunto de $PCS(b)$ para analizar si existe o no un *BIC* y si manualmente podemos identificarlo. Después de entender el error y los cambios que arreglaron ese error, es necesario identificar el *BIC* en caso de que exista, así como el *FFM*. Consecuentemente, la identificación del *BIC* comienza con el análisis del conjunto de $PCS(b)$, para cada uno de los commits pertenecientes al conjunto de $PCS(b)$ se analizan las líneas modificadas para encontrar si alguno de los commits previos introdujo el error. En este punto, hay tres escenarios diferentes y el comportamiento es difiere según las siguientes condiciones:

1. *El commit insertó las líneas que contienen el error*: Este commit es el *BIC* porque insertó las líneas defectuosas en el código fuente en el momento de su escritura. El *BIC* que a su vez es el *FFM* se pudo identificar porque es el primer commit que manifestó el comportamiento incorrecto. De acuerdo con el modelo teórico, el TSB ejecutado en el *BFC* pasa y el *BIS* es una de las instantáneas previas, es decir, es uno de los commits previos del *BFC*.

[4] `https://github.com/elastic/elasticsearch/commit/beaa9153a629c095`

2. *El commit no inserta las líneas que contienen el error*: en este caso, hay dos resultados posibles:

   - Las líneas del commit son correctas ya que no insertaron el error en las líneas del código fuente en el momento de su escritura y otros factores externos causaron que esas líneas se volvieran defectuosas. En este escenario, no hay un *BIC* en y el análisis debe centrarse en comprender si este commit es el *FFM*. De acuerdo con el modelo teórico, la TSB se ejecuta en todo el $ACS(b)$ y pasa en el *BFC* pero siempre falla en las instantáneas de los ancestros. Sin embargo, si se cambia el entorno, el TSB falla en el *BFC* pero pasa en la instantáneas anteriores, y el primer *BIS* que falla en la secuencia continua de *BIS* es el *FFM*.

   - Las líneas del commit son cambios sintáctico y modificaciones semánticas equivalentes (refactorizaciones): Esto significa que el commit conserva el mismo comportamiento que antes, por lo tanto, es necesario volver a navegar, en el conjunto de $DCS(b)$ y volver al primer punto de esta lista para identificar el *BIC*.

3. *No es seguro que el commit insertase el error*: en este escenario, es importante continuar navegando por el conjunto de $DCS(b)$, si el commit insertó por primera vez en las líneas descendentes al conjunto de lineas del $LC(c)$ y no podemos estrar seguros de que esas l'ineas contengan el error, el commit se clasifica como "indecisa". Esto significa que después del análisis, el *BIC* no se pudo encontrar manualmente a pesar de que podemos asegurar su existencia, o que no podemos asegurar el *FFM* o *BIC* debido a la falta de información.

B.6.3 Resultados de las etapas

Al final del proceso, hay tres resultados posibles para cada informe de error analizado. Los resultados se basan en la identificación inequívoca del *BIC* y del *FFM* dado un *BFC*. Estos resultados se encuentran explicados a continuación:

- Un *BFC* fue inducido por un *BIC*: en este caso es seguro que al menos hay un commit que introdujo las línea(s) errónea(s) entre el conjunto de commits previos, o conjunto

de commits descendientes o el conjunto de commits de ancestros. Sin embargo, durante el análisis manual puede ocurrir que:

1. El momento se pudo identificar manualmente o,
2. El momento no se pudo identificar manualmente.

Además, en este escenario, el *BIC* es también el *FFM*.

- Un *BFC* no fue inducido un *BIC*: en este caso es seguro que ninguna línea insertada en el código fuente contenía el error cuando se introdujo el commit, y otros factores como por ejemplo, los cambios en las necesidades internas del proyecto, o cambios en los recursos externos consumidos por el proyecto, provocan que el código se vuelva defectuoso. También se puede confirmar que ninguno de los commits antecesores insertó las líneas incorrectas, por lo que no se puede culpar a ninguno de ellos como el *BIC* y solo se podrá identificar la primera vez que manifiesta el error en el código. Además, en este escenario puede ocurrir que:

  1. El momento se pudo identificar manualmente o,
  2. El momento no se pudo identificar manualmente.

- No está si un *BFC* due inducido por un *BIC*: algunos informes de errores no detallan suficiente información sobre el error y el *BFC* no es suficiente para decidir si hay o no un *BIC*. Además, algunos commits son muy complejos para comprender sus cambios y no se puede decidir si insertaron o no el error.

B.7 Resultados

Hay tres secciones diferentes para discutir los resultados obtenidos en el Capítulo 4, Capítulo 5 y Capítulo 6.

B.7.1 Reproducibilidad y credibilidad del algoritmo SZZ

En el Capítulo 4 de esta tesis se detalla el estudio de la revisión sistemática de la literatura (*SLR*) sobre el uso del algoritmo SZZ en estudios empíricos de ingeniería del software

Table B.1: Los tipos o publicaciones más frecuentes que utilizan completamente el algoritmo SZZ (N = 187). *#diferentes* cuenta los diferentes lugares en cada grupo, *#publications* cuenta el número total de publicaciones en ese tipo de lugares.

Tipo	# Diferentes	# publicaciones
Revistas	21	42
Conferencias & Simposiums	40	102
Workshops	13	13
Tesis universitarias	20	30

(*ESE*).La SLR demostó que el algoritmo es ampliamente usado en la ESE y que posee una alta relevancia, no se limita solamente a una audiencia específica, si no que se ha difundido por numerosas publicaciones en revistas y conferencias importantes en varias areas de la Ingeniería del Software. Tabla B.1 muestra los diferentes medios de publicaciíon que han usado el algoritmo SZZ. Hay cuatro diferentes medios: Tesis universitarias, artículos en workshops, artículos en conferencias y symposiums, y artículos en revistas.

Durante la SLR se ha observado que las limitaciones del algoritmo SZZ son bien conocidas y documentadas. Se han propuesto mejoras para mitigar las limitaciones que presenta, pero hasta el moment no han llegado a tener demasiado éxito. Además, mientras que las limitaciones de la primera parte del algoritmo, relacionadas con enlazar los Bug-Fixing Commits (*BFC*) con lo bug reports, han obtenido significantes mejoras. Las propuestas para mitigar las limitaciones de la segunda parte, relacionadas con encontrar el Bug-Introducing Commit (*BIC*), todavía tiene espacio para mejorar.

La SLR ha demostrado que la mayoría de las publicaciones que usan el algoritmo SZZ, o sus versiones mejoradas: SZZ-1 [Kim et al., 2006c] y SZZ-2 [Williams and Spacco, 2008], no mencionan las limitaciones, y lo que resulta mas interesante es que las limitaciones de la primera parte del algoritmo resultan ser más discutidas que las de la segunda parte a pesar de ser menos relevantes para la veracidad de los resultados obtenidos. La Tabla B.2 muestra cuántas publicaciones han utilizado mejoras en el algoritmo SZZ para mitigar las limitaciones del original. Tenga en cuenta que la columna “Mixed ” de la Tabla B.2 se refiere a los documentos que han utilizado la versión original, y alguna de las versiones mejoradas en el mismo estudio. En la Tabla B.2 se puede observar que solamente el 38% de las publicaciones analizadas usan la versión *original* SZZ. Por tanto, para aumentar la reproducibilidad

Table B.2: Número de artículos que han utilizado el algoritmo SZZ original, las versiones mejoradas del SZZ o algunas adaptaciones para mitigar las amenazas de validación.

	Original SZZ only	SZZ-mejorado only	SZZ-modificado	Mezcla
# artículos	71 (38%)	26 (14%) [a]	75 (40%)	15 (8%)

[a] 22 (12%) de los artículos usan el SZZ-1 y solo 4 (2%) de los artículos usan el SZZ-2.

y credibilidad de los resultados, recomendamos que los investigadores que desarrollen modificaciones del algoritmo SZZ, publiquen el software implementado en sitios como `GitHub`, de este modo otros investigadores pueden *fork* el proyecto. Esos *forks* pueden ser fácilmente trazados, y los autores pueden preguntar por una específica citación a su solución si otros autores hacen uso de ella. Además otra de las ventajas que posee el publicar el software implementado es que la reproducibilidad es más específica, debido a que en ocasiones los autores por falta de espacio no mencionan como funciona o como se implantó el algoritmo que han usado y es posible que este hecho esconda errores al reproducir el estudio.

Hemos encontrado que la reproducibilidad de las publicaciones es limitada, y los paquetes de réplica se ofrecen muy raramente. La Tabla B.3 ofrece el número de estudios que a)contiene un paquete de reproducción, b) han detallado cuidadosamente la metodología para permitir las reproducciones de sus estudios. Hemos clasificado las publicaciones en cuatro grupos: i) publicaciones que ofrecen un paquete de reproducción (*paquete*), ii) publicaciones que detallan la metodología y los datos utilizados (*Environment*), iii) publicaciones que tienen ambos *Ambos*) y iv) Ninguno (*Ninguno*).

De las 187 publicaciones, 43 ofrecen un paquete de reproducción y 96 cuidadosamente detallan los pasos seguidos y los datos utilizados. Además, solo 24 de los artículos proporcionan tanto el paquete de replicación como la metodología y los datos detallados. 72 de los documentos no ofrecen un paquete de reproducción o una descripción detallada de la metodología y los datos usados.

A pesar de que usar las versiones mejoradas del algoritmo SZZ ayuda a aumentar la precisión del algoritmo como señala [Rahman et al., 2012]: “Accuracy in identifying Bug-Introducing Commit may be increased by using advanced algorithms (Kim et al. 2006, 2008)”, raramente se usan – sóle el 22% de las publicaciones usan una de las dos mejoras propuestas para el algoritmo SZZ. Parece que los investigadores prefieren reinventar la

Table B.3: Publicaciones por su reproducibilidad: Filas: *Si* significa el número de trabajos que cumplen cada columna, mientras que su complemento es *No*. Columnas: *Paquete* es cuando ofrecen un paquete de reproducción, *Environment* cuando proporcionan la metodología detallada y un conjunto de datos usados. Tenga en cuenta que *Both* es la intersección de *Paquete* y *Environment*. (N = 187)

	Solo paquete	Solo Environment	Ambos	Ninguno
Si	19	72	24	72
No	168	96	163	115

rueda, el 49% de los artículos analizados usan una versión modificada o "ad-hoc" por ellos mismos del SZZ, en vez de utilizar las mejoras propuestas en otros artículos. Una posible razón para explicar este comportamiento pooría ser que los artículos que describen el SZZ o cualquiera de sus mejoras propuestas, SZZ-1 y SZZ-2, no aportan el código software de la implementación. Por ello, los investigadores tienen que implementar el software desde cero para poder usarlo en sus investigaciones. Nuestros resultados muestran que en dicha situación, lo que se hacer es usar el concepto general que describe el algoritmo SZZ y añadir algunas modificaciones con el propósito de mitigar algunas de las limitaciones presentes en el SZZ original. Para todos las soluciones identificadas como "ad-hoc", no se ha encontrado ninguna razón que explique el por qué no se implementaron las otras mejoras del SZZ. Otro problema importante es el etiquetado del las mejores de SZZ, ya que no han sido ni etiquetadas ni enumeradas. Cuando una versión mejorada del SZZ es usada en los artículos, a menudo lo autores se refieren a ella como SZZ, dificultando el seguimiento, la reprodureproción y la ráplica de sus resultados.

Por otro lado, en esta tesis se ofrece una simple manera de medir la facilidad de reproducción y credibilidad en los artículos de investigación. Esta medida se basa en puntuar cinco características que se analizaron en los artículos. Si las preguntas fueron respondidas positivamente, el artículo fue marcado con un puntaje positivo, de lo contrario con un 0:

1. ¿El estudio informa de las limitaciones del SZZ? (puntaje = 1 punto)

2. ¿Los autores llevan a cabo una inspección manual de sus resultados? (puntaje = 1 punto)

3. ¿El estudio aporta un paquete de reproducibilidad? (puntaje = 2 puntos)

4. ¿El estudio proporciona una descripción detallada de los métodos y datos utilizados? (puntaje = 1 punto)

5. ¿Utiliza el estudio una versión mejorada de SZZ? (puntaje = 2 puntos)

Los artículos que sumen puntos del 0 al 1 se consideran *Pobres* de ser reproducibles y poseer resultados creíbles. Los artículos que sumen puntos del 2 al 4 se consideran *Justos* de ser reproducibles y poseer resultados creíbles. Aquellos artículos que sumen puntos del 5 al 6 se consideran *Buenos* de ser reproducibles y poseer resultados creíbles. Finalmente los artículos que sumen 7 puntos se consideran *Excelentes* de ser reproducibles y poseer resultados creíbles

Durante el análisis también hemos observado que raramente se encuentran informes exhaustivos completos de la reproducibilidad, solamente se han clasificado un 15% de los artículos como *buena y excelente calidad con respecto a la reproducibilidad*. La comunidad de investigación debe poner mayor atención a estos aspectos; nosotros creemos que se está otorgando demasiada atención a los resultados finales (el *producto* de la investigación: nuevo conocimiento) en comparación con el proceso de investigación. Como investigadores en el campo de la ingeniería del software sabemos que ambas partes – una alta calidad del producto y una alta calidad del proceso – son esenciales para obtener progresos exitosos a lo largo del tiempo [Kan, 2002].

Para resumir, consideramos importante resaltar algunas de las lecciones aprendidas después de llevar a cabo la SLR. El algoritmo SZZ está basado en heurísticos y asunciones, por tanto para aportar unos resultados más creíbles, recomendamos que los investigadores especifiquen y argumenten el uso de esos métodos y algoritmos cuyo fin es mitigar las limitaciones presentes en el SZZ en sus estudios. Ser consciente del riesgo de cada asunción usada y si es necesario, validar manualmente una porción de los resultados obtenidos. Además, para llevar a cabo estudios empíricos, los autores deben ser conscientes que para permitir la reproducción de sus publicaciones, el mejor método es incluir un paquete de reproducción que puede estar públicamente disponible junto con su publicación (idealmente para siempre). Por otro lado, ellos también tiene que ser conscientes de que algunas características en sus estudios pueden cambiar, por ejemplo los entornos del software, causando que programa y

los datos se encuentren obsoletos, por lo que es necesario detallar y describir con precisión cada uno de los elementos, métodos y software usados durante el estudio.

B.7.2 Teoría de Inserción del error

Esta tesis propone una solución al problema actual de identificar el Bug-Introducing Commit (*BIC*) dado un commit que arregla un error (*BFC*). Actualmente, la mayor parte del trabajo se basa en métodos y técnicas formuladas bajo la suposición intrínsecamente errónea de que las líneas de código que se han modificado para corregir el error, son las líneas que han introducido el error en primer lugar. La naturaleza problemática de esta suposición es conocida, como se ha demostrado anteriormente en la revisión sistemática de la literatura. Sin embargo, esta tesis hace una contribución importante al tratar de cuantificar el alcance del problema y al detallar un nuevo modelo de introducción de errores para encontrar la primera vez que el software manifiesta un comportamiento incorrecto, sin la necesidad de culpar a un cambio como el Bug-Introducing Commit, si no que identifica el primer cambio que establece el error y comprende ese momento en su contexto y dependencias.

Después de analizar varios informes de errores durante esta tesis, nos hemos dado cuenta que determinar dónde, cuándo y cómo se introdujo un error no es una tarea trivial, en la que a veces se involucran muchos investigadores con el fin de aclarar la naturaleza del error investigado. De hecho, no es fácil determinar si el error estaba presente en el código en el momento de realizar un cambio. Por ejemplo, hay casos en los que las técnicas automáticas actuales no pueden determinar el punto de introducción del error, ya que no se puede identificar ningún commit previo. Uno de esos casos es cuando el commit/cambio que arregla el error (*BFC*) solo introduce nuevas líneas en el código: en este caso, no hay forma de identificar el commit(s) previo(s) tal como lo definimos, ya que no hay ningú commit previo tocando las líneas que han sido añadidas. En este caso, solo la descripción del informe del error o la descripción y el código fuente del *BFC* podría aclarar si las nuevas líneas no se incluyeron debido a un olvido en un cambio ancestor o debido a que se incluyeron para satisfacer algún nuevo requisito o nueva característica del proyecto.

La SLR ayudó a cuantificar las limitaciones y los problemas que afectan a los algoritmos que se basan en el SZZ y son utilizados para identificar el Bug-Introducing Commit. Por tanto, teniendo esto en consideración, hemos propuesto un modelo en el que, por definición,

se integran todos los escenarios que imposibilitan el correcto funcionamiento del SZZ. El modelo propuesto es capaz de lidiar con estos escenarios y dar solutión al problema. A continuación, explicamos brevemente como el modelo propuesto es capaz de abordar cada una de las limitaciones encontradas en la SLR.

1. Identificación de más de un commit previo: Nuestro modelo no trata de identificar los commits previos de las líneas modificadas o eliminadas en un *BFC*, si no que a través de un test comprueba el comportamiento de la funcionalidad que se arregló en el *BFC* en todas las instantáneas anteriores, descendientes y ancestrales del proyecto que esta siendo analizado hasta que encuentre la primera vez que el test falla.

2. La utilización de nuevas líneas para corregir el error en el *BFC*: Nuestro modelo no busca las líneas que se han modificado o eliminado para corregir un error, sino que considera todos los cambios realizados en el *BFC*, puesto que comprueba la funcionalidad que estaba fallando. Por tanto no distingue entre los *BFC* que solo presentan líneas añadidas para descartarlos del análsis.

3. Presencia de cambios en el entorno o la configuración: Con nuestro modelo podemos detectar errores causados ??por un cambio en los entornos, ya sea que podríamos ser capaces de reproducir el entorno anterior y posterior a un *BFC* y observar que con un entorno específico anterior al *BFC* las pruebas de comprobación pasan en las instantáneas previas al *BFC*, pero que con este entorno fallan en la instantánea del *BFC* y al revés. En este caso, habrá un *FFM* pero no un *BIC*.

4. Varias modificaciones sobre una línea: en nuestro modelo, no importa cuántas veces se haya modificado una línea, ya que no señala al último cambio como el *BIC*. La prueba de comprobación busca la primera vez que se inserta el error en la línea que se está probando.

5. Nivel semántico débil: como en el ejemplo anterior, el modelo propuesto trata con el inconveniente de tener un nivel semántico débil porque no se centra en identificar el último cambio, sino que prueba el comportamiento del código.

6. Un *BFC* corrige a la vez más de un error: en estos casos, nuestro modelo puede diseñar una prueba de comprobación específica para cada error, buscando la primera vez que la

prueba falla al comprobar la presencia de un error concreto en instantáneas anteriores al *BFC*.

7. Razones de compatibilidad:

8. Errores latentes: el modelo identifica la primera vez que el error se insertó en el código fuente utilizando el la prueba de comprobación, por tanto, no importa cuánto tiempo haya permanecido el código defectuoso en el proyecto, porque teóricamente encontraremos la primera vez que se introdujo.

Muchos investigadores han basado sus métodos para localizar el cambio que introdujo el error en el algoritmo SZZ o algoritmos similar, estos métodos carecen de medios para hacer frente a las limitaciones anteriormente comentadas, y por tanto tienen que formular algunas heurísticas que, por ejemplo, eliminan el *BFC* con solo nuevas líneas añadidas porque no pueden rastrear esas líneas, o eliminan commit numerosas modificaciones o eliminan los commits más antiguos porque es poco probable que sean el *BIC* [da Costa et al., 2016]. Como consecuencia, estos heurísticos pueden inducir a errores en los resultados, y mostrar un mayor porcentaje de precisión del algoritmo al identificar el *BIC*.

El SZZ es el algoritmo, hasta ahora, más conocido y más fácil de usar para identificar el cambio que introdujo el error o *BIC*. Debido a esto, algunos estudios utilizan el algoritmo SZZ o variantes de él para extraer conjuntos de datos relacionados con el *BIC* y los usan para alimentar sus modelos de predicciń o clasificaciń de errores. Por ejemplo, Ray *et al.* usaron un conjunto de datos que se recopiló usando el algoritmo SZZ para estudiar la naturalidad del código con errores [Rahman et al., 2014]. Massacci *et al.* evaluó la mayoría de los modelos de detección de vulnerabilidades existentes en los navegadores web y usó muchos conjuntos de datos construidos usando el algoritmo SZZ [Massacci and Nguyen, 2014]. Finalmente, Abreu *et al.* usó el conjunto de datos obtenido en [Abreu et al., 2009] para estudiar cómo la frecuencia de comunicación entre desarrolladores afecta el hecho de introducir un error en el código fuente.

En general, este nuevo modelo propuesto, es capaz de distinguir entre dos momentos relevantes dado un commit que corrige un error *BFC*, estos momentos son el momento de introducción del error *BIC* y el momento de manifestación de ese error en el código *FFM*. Mientras que el primer momento no siempre existe, debido a que algunos cambios externos

o la evolución de los requisitos internos han cambiado provocando la falla, siempre hay un primer momento de fallo que indica cuándo el proyecto manifiesta el error corregido por el *BFC*. Para obtener resultados creíbles en las áreas de predicción de errores, categorización de errores y los modelos de detección de errores, es necesario estudiar y distinguir estos dos momentos, ya que como demuestró nuestro estudio empírico los algoritmos o basados en el SZZ o variantes de él identifican desde el 26% hasta el 38% de falsos positivos "reales". Por lo tanto, estos resultados son alentadores para comenzar a pensar e implementar nuevos métodos basados ??en el modelo teórico propuesto en esta tesis, o al menos, considerar reformular correctamente la definición de insertar un error en el código fuente. Ya que hasta ahora, el proceso de insertar un error en el código ha sido considerado como un problema estático, en el sentido que para los investigadores siempre ha estado presente el error que se ha corregido. Nuestra intuición nos lleva a reclamar nuevos métodos que sean capaces de distinguir entre líneas defectuosas que contienen el error en el momento de insertarlas y líneas limpias que no contienen el error en el momento de insertarlas. Esta disertación ha demostrado que existen otras razones externas e internas que provocan un fallo en el sistema, y no es justo culpar a un cambio anterior como el responsable de provocar el error cuando éste cambio era correcto en el momento en el que introdujo, es decir, éste cambio cumplía las funcionalidades, entornos, necesidades y requisitos del proyecto. Aunque hemos conseguido profundizar en el problema de identificar correctamente el cambio que provocó el error, necesitamos dedicar más esfuerzo para lograr una mayor automatización del modelo propuesto, encontrar formas o técnicas que sean capaces de abordar tanto el problema de reconstrucción de un entorno antiguo, como capaces de construir un test que pueda ser automatizado o parcialmente automatizado para encontrar los dos momentos más importantes, el *BIC* y el *FFM*.

B.7.3 Estudio Empírico: Aplicación de la teoría propuesta para localizar el momento de introducción de un error

El estudio empírico detallado en el Capítulo 6 respalda la necesidad de búsqueda de un nuevo modelo para identificar inequívocamente los cambios que introducen errores, ya que no siempre es obvio identificar y entender cómo se introducen los errores en el código fuente con los métodos actuales. Esta tesis presenta en el Capítulo 5 un nuevo modelo que resuelve esta

Table B.4: Porcentaje de Bug-Fixing Commit que han sido inducidos por un Bug-Introducing Commit *BIC*, y que no han sido inducidos por un Bug-Introducing Commit *NO_BIC*.

	BIC	NO_BIC
Nova	45 (79%)	12 (21%)
ES	54 (91%)	5 (9%)

necesidad y su implementación se ha estudiado empíricamente en el Capítulo 6. Dónde el modelo propuesto se ha utilizado para llevar a cabo un estudio cualitativo de un conjunto de informes de errores para identificar, si existe, el cambio que introdujo el error (*BIC*) dado un cambio que arregló el error (*BFC*).

Identificar las líneas que contienen un error en el código fuente de un proyecto no es un proceso tan sencillo como se podría pensar en un primer momento. De los 120 informes de errores extraídos de Nova y ElasticSearch en la primera fase, 116 informes pasaron a la segunda fase, cada uno de los informes de error se encuentra enlazado con el cambio que arregló el error. En la segunda fase, los investigadores analizaron manualmente si el *BFC* fue inducido por un *BIC* los resultados se presentan en la Tabla B.4. La tendencia en los resultados de ambos proyectos fue similar, ya que ambos resultados presentan un mayor porcentaje de *BFC* que fueron inducidos por un *BIC* en lugar de ser inducidos por otras razones. Sin embargo, el porcentaje de *BFC* que no fue inducido por un *BIC* es más representativo en Nova con un 21% de los *BFC* que arregló un error que no se introdujo en las líneas del sistema. Por el contrario, este porcentaje es menor en ElasticSearch, donde solo el 9% de los *BFC* no fueron inducidos por un *BIC*. Además, de los 45 *BFC* inducidos por un *BIC* en Nova, fuimos capaces de identificar manualmente 34 de ellos. Mientras que en ElasticSearch, de los 54 *BFC* inducidos por un *BIC* fuimos capaces de identificar manualmente 38 de ellos. En ambos proyectos se puede observar la dificultad en la tarea de identificar manualmente el origen del error, ya que el porcentaje de duda entre los *BFC* analizados alcanza un 19% en Nova y el 27% en ElasticSearch.

Durante el análisis manual, se descubrieron algunas razones que explican por qué un *BFC* no es inducido por un *BIC* que se han presentado en forma de una clasificación anecdótica. La Tabla B.5 muestra las razones más comunes en ambos proyectos. En Nova, con un porcentaje

de ocurrencia del 42% la razón principal fue la *Co-evolución* de las líneas que se cambiaron para corregir el error con los requisitos internos del proyecto. Es decir, debido a la evolución del proyecto, las líneas que fueron insertadas en un momento puede que no tengan sentido o no cumplan las necesidades actuales del proyecto y se ha manifestado un error en esas líneas, sin embargo, este error no significa que las líneas fuesen erróneas en el momento que fueron insertadas, si no que debido a las nuevas necesidades del proyecto han pasado a manifestar el error. El segundo motivo, con un porcentaje de ocurrencia del 33% es la presencia de errores en APIs externas que son consumidas por nuestro código. La tercera razón más frecuente es la *Co-evolución* de las líneas que se modificando para corregir el error. Por otro lado, en ElasticSearch los porcentajes se distribuyen por igual con un 40% de ocurrencia en *Co-evolución interna* y errores en APIs externas, con la única excepción de que en este proyecto no encontramos ningún error causado por la incompatibilidad de hardware y software. Esta clasificación anecdótica que explica las razones por las cuales un *BFC* no es inducido por un *BIC* debería investigarse con mayor profundidad, ya que puede ayudar a los investigadores a identificar patrones diferentes y quizás ocultos, que ayuden a explicar y entender mejor cómo se insertan y se manifiestan los errores en el código fuente. El propósito de esta tesis no es establecer una única clasificación que explique los motivos por los cuales un *BFC* no es inducido por un *BIC* o que esta clasificación se pued a extender a otros proyectos, si no que el motivo inicial de la clasificación es alertar e informar sobre el hecho de que hay otras razones donde una línea que cuando no se insertó no era defectuosa está causando/manifestando un error. Por tanto, creemos que estos casos deberían analizarse en profundidad para mejorar las clasificaciones actuales basadas en el origen del error.

Después de analizar varios errores durante el estudio empírico, se ha observado que alcanzar un acuerdo en la clasificación de la causa raíz de un error es tedioso y muchas veces se basa en la subjetividad de los investigadores. Cabe señalar que, aunque el análisis manual fue realizado principalmente por el autor de esta tesis, otros investigadores se involucraron para discutir y analizar la causa del error, en caso de dudas. Por lo tanto, esta disertación sugiere una clasificación inicial, sin tratar de hacer ninguna afirmación al respecto, pero se observa a partir de los resultados que del 9% al 21% de los *BFC* analizados no fueron inducidos por *BIC* y esta clasificación puede ayudar a entender mejor los principales motivos. Es posible que si otros investigadores repliquen nuestro trabajo, puedan obtener leves diferen-

Table B.5: Razones por las cuales un Bug-Fixing Commit no es inducido por un Bug-Introducing Commit

	Nova	ElasticSearch
Co-evolución Interna	5 (42%)	2 (40%)
Co-evolución Externa	2 (17%)	1 (20%)
Compatibilldad	1 (8%)	0 (0%)
Error en API Externa	4 (33%)	2 (40%)

cias en la clasificación, puesto que en algunos casos sin una guía detallada, el análisis puede ser subjetivo. Y por ello, actualmente estamos trabajando en minimizar lo máximo posible la subjetividad en la clasificación.

Esta tesis discute un amplio abanico de opciones en términos de lo que puede ser y no puede ser un *BIC* y también, cómo el algoritmo SZZ fallará dependiendo de la implementación/versión específica que se use del algoritmo SZZ. Un elemento esencial del estudio empírico es la obtención del "estándar de oro", ya que tiene las caracterizaciones de los *BIC*. Por lo tanto, el estudio puede cuantificar el número "real" de falsos positivos, falsos negativos y verdaderos positivos en el rendimiento de los algoritmos SZZ y SZZ-1, que hasta donde sabemos, nadie ha intentado cuantificar anteriormente. En base a los resultados presentados en la Tabla B.6 (mas detalles en el Capítulo 6, Sección 3.4,), en el mejor escenario, el algoritmo calcula el 55% de falsos positivos en Nova con un F1-score de 0.44. Mientras que en ElasticSearch, el número de falsos positivos es superior, con un 60% y el F1-Score es igual que en Nova, 0.44. Estos resultados pueden parecer contradictorios con los resultados anteriores donde ElasticSearch tiene un menor porcentaje de *BFC* que no son inducidos por un *BIC* y esto puede hacer que el número de falsos positivos sea menor que en Nova. Sin embargo, la razón por la cuál ElasticSearch computa más falsos positivos se debe a que el número de *BFC* con conjunto commits previos mayor que uno es superior en ElasticSearch que en Nova. Este hecho provocó que la heurística del algoritmo SZZ fallase con más frecuencia en ElasticSearch, y casi en la mitad de estos casos se asumió que el commit mas antiguo en el tiempo perteneciente al conjunto $PCS(e)$ ra el *BIC* y en realidad, esta heurística no se cumplió en

Table B.6: Resultados de verdaderos positivos, verdaderos negativos, falsos negativos, recall y precisión para los algoritmos SZZ y SZZ-1 suponiendo que el algoritmo solo marca uno los compromisos anteriores del conjunto de $PCS(b)$ como *BIC*.

	Verdaderos Positivos	Falsos Positivos	Falsos Negativos	Precisión	Recall	F1
Nova (SZZ)	25 (26%)	54 (56%)	17 (18%)	0.32	0.60	0.42
Nova (SZZ-1)	28 (30%)	51 (55%)	14 (15%)	0.35	0.58	0.44
ES (SZZ)	26 (27%)	59 (61%)	12 (12%)	0.31	0.68	0.43
ES (SZZ-1)	27 (28%)	58 (60%)	11 (12%)	0.32	0.71	0.44

la mayoría de los casos. Otra razón que explica por qué Nova calcula una recall más alta que ElasticSearch es el número de *BFC* con solamente nuevas líneas añadidas, esto causó que la cantidad de falsos negativos aumentase ya que SZZ no incluye estos *BFC* en su análisis.

Otros estudios como Kim *et al.* [Kim et al., 2006c], Williams y Spacco [Williams and Spacco, 2008] y Da Costa *et al.* [da Costa et al., 2016] también analizaron manualmente muestras de datos usando el algoritmo SZZ. Sin embargo, esto autores obtuvieron porcentajes mucho más altos de precisión y recall en los resultados obtenidos en el SZZ que los que se han obtenido en esta tesis. Esto se debe a que éstos estudios no estaban comparando sus resultados con ningún "estándar de oro", y por tanto no diferenciaban entre *BIC* y *FFM*, y no definían exactamente qué era un error. Como consecuencia, estos estudios no contemplaron otros posibles escenarios donde la causa del error era debida a otras razones distintas a la actual premisa, por ejemplo, cambios en APIs externas o cambios de co-evolución. Mientras que en esta tesis analizamos todo el contexto de un *BFC*, éstos estudios solo se enfocan en verificar si dado un *BFC* el algoritmo es capaz de encontrar un posible *BIC*, sin la necesidad de que este *BIC* sea la verdadera razón. Por otro lado, puede ocurrir que debido a los proyectos seleccionados en estos estudios, el porcentaje de falsos negativos disminuya debido a casos muy diferentes de introducción de errores por commit previos. Esta es uno de las razones por las cuales proponemos una línea de trabajo futuro con el fin de extender nuestro análisis a un mayor conjunto de proyectos.

En cualquier caso, nuestra investigación muestra evidencia de que asumir que el commit

previo a las líneas modificadas en un *BFC* es el origen del error no se cumple para una fracción significativa de errores. Además, también muestra que el modelo propuesto en el Capítulo 5, al menos teóricamente, cumple con la preocupación fundamental de identificar inequívocamente el *BIC*, cuando éste existe. Y muestra el porcentaje de falsos positivos presentes en el algoritmo SZZ, que en el mejor de los casos es un 25% en Nova usando SZZ-1 donde se assume que el BIC es el mas temprano en el tiempo de todos los possibles BIC identificados. Sin embargo, es necesario realizar más investigaciones sobre la automatización del modelo, así como investigar más a fondo cómo los factores externos y la evolución de los requisitos afectan a la manifestación e inserción de errores en los proyectos.

B.8 Conclusiones y Trabajo Futuro

Este capítulo recapitula los objetivos iniciales de investigación y las contribuciones establecidas en el Capítulo 1. Además describe las conclusiones principales de esta tesis así como las posibles líneas de investigación futura.

B.9 Conclusiones

Las conclusiones de esta tesis se presentan en relación con el conjunto de objetivos y contribuciones introducidos en las secciónes 1.2 y 1.3 del Capítulo 1. En primer lugar, se presentan las conclusiones obtenidas tras estudiar a través de una revisión sistemática de la literatura el uso del algoritmo SZZ y el problema actual relacionado con la identificación del momento en que se introdujo un error en el código fuente de un proyecto, así como una cuantificación de las limitaciones del algoritmo SZZ. Después, se presentan las conclusiones relacionadas con el modelo teórico propuesto para identificar con mayor precisión el momento de introducción de un error, en caso de que este momento exista. Esta solución se trata de un modelo teórico que distingue entre el momento de introducción y el momento de manifestación del error, para ello se utiliza un hipotético test que comprueba la funcionalidad que ha sido arreglada en otros momentos previos. Por último, se presentan las conclusiones relacionadas con el estudio empírico sobre la aplicación del modelo teórico propuesto para identificar los *BIC* en dos casos de estudio: Nova y ElasticSearch.

Para comprender el problema actual sobre dónde se introdujo el error, se ha realizado una revisión sistemática de la literatura (SLR) en el uso del algoritmo SZZ, que es uno de los algoritmos más conocidos para identificar el cambio que introdujo el error (*BIC*). De las 458 publicaciones que citan el algoritmo, la muestra analizada en la SLR constaba de 187 publicaciones que usaban las dos partes del algoritmo. Esta SLR aporta más conocimiento para entender los problemas de investigación presentes en la ingeniería del software cuando se asumen premisas erróneas, como ocurre con el uso del algoritmo SZZ. Los resultados de la SLR demostraron que las publicaciones que usan el SZZ son en su mayoría no reproducibles, además no es común que los artículos mencionen las limitaciones del uso de éste algoritmo y finalmente, que los autores prefieren usar sus propias mejoras del algoritmo antes que las mejoras propuestas por otros autores. Además, la SLR sirvió para cuantificar las limitaciones y los problemas que afectan a este particular algoritmo cuando se trata de identificar el *BIC*.

Después de entender y cuantificar el problema actual sobre la identificación del *BIC*. Se propuso un modelo teórico que da solución a estas limitaciones al usar el algoritmo SZZ o algoritmos basados en el SZZ. Importantes aportaciones en esta tesis son: la definición de qué es un error, cómo identificar el momento de inserción de un error usando el propuesto modelo y finalmente el criterio que debe aplicarse para usar el modelo. Como ya se ha comentado anteriormente, el modelo propuesto se basa en la idea de un test hipotético que conoce en cada momento cómo debe comportarse el proyecto en un momento específico. El modelo propuesto contempla todos los escenarios que limitan el correcto funcionamiento del SZZ ya que ejecuta un test que comprueba en cada uno de los momentos previos al *BFC*, la funcionalidad que se ha arreglado en el *BFC* y no solo se tiene en cuenta el análisis de las líneas que han sido modificadas par identificar el origen del error. Por tanto, el momento en el que el test falle por primera vez, es el momento en el que se introdujo el error.

Por último, se presentas las conclusiones sobre el experimento empírico que estudia la aplicación práctica del modelo propuesto en dos casos de estudio: Nova y ElasticSearch. Éste estudio ha demostrado que para una larga fracción de los errores analizados se puede identificar si un *BFC* ha sido causado por un *BIC* o no usando la teoría del modelo propuesto. Cuando un *BFC* es causado por un *BIC*, hemos identificado manualmente el 60% de los commits que insertaron el error en Nova y el 68% en ElasticSearch. Además, esta tesis también propone una terminología para entender e identificar cada uno de los elementos que

forman parte del análisis para localizar el origen del error. Hasta donde sabemos, esta es la primera vez que se detalla una terminogía con el fin de explicar e identificar el *BIC* dado un *BFC*.

Para resumir, esta tesis se centra en describir y contextualizar el problema actual de identificar el origen de un error. Para tratar este problema, hemos estudiado y cuantificado las limitaciones que afectan al algoritmo más usado, y hemos propuesto una solución basada en un nuevo modelo teórico que localiza inequívocamente el *BIC*. Hemos aplicado este modelo a dos casos de estudio para localizar el origen del error creando un conjunto que contempla "la gran verdad" y que ha sido usado para comparar el rendimiento real de algoritmos basados en el SZZ.

Si esta tesis sufriese modificaciones, la última versión disponible se encuentra en `http://gemarodri.github.io/Thesis-Gema`.

B.10 Trabajo Futuro

Después de aplicar el modelo propuesto y su criterio para identificar el commit que introdujo el error y el commit que manifiestó el error por primera vez, es comprensible pensar que uno de los trabajos futuros sea tratar de automatizar la teoría propuesta lo máximo posible. De este modo, se obtendría automáticamente el *BIC*, en el caso de que exista, y el *FFM* para cada *BFC* introducido en el modelo. Como se ha mencionado anteriormente, el modelo contempla algunos escenarios que son difícilmente automatizables, en su mayoría debido a la dificultad de reconstruir un estado del sistema que use dependencias que están obsoletas en la actualidad. Por esta razón, el *TSB* no podría ser implementado en ciertas versiones anteriores al *BFC* en el proyecto. Como trabajo futuro me gustaría encontrar un proyecto óptimo donde se detallasen cada una de las dependencias y entornos de un sistema y que además, estuviesen todavía disponibles. En este proyecto estudiaría la automatización del modelo propuesto. Desde un punto de vista práctico, la automatización del modelo propuesto es interesante porque aporta al los proyectos de software una valiosa herramienta para entender mejor que es un error y como fue introducido, y por tanto diseñar medidas para mitigarlo.

Otra futura línea de investigación es seleccionar un tamaño de muestra mayor con el fin de llevar a cabo una clasificacón que estudiaría la frecuencia con la que dado un *BFC*, éste

presenta un *BIC*. La clasificación aportará más conocimientos sobre la posible existencia de patrones que han permanecido ocultos en a literatura actual y que porían ser descubiertos analizando la relación entre los *BFC* y su origen a través de las descripciones del informe de error, los cambios realizados en el código y la aplicación del modelo propuesto. Además, este estudio prodría ayudar a diseñar mejor la integración de test, para identificar con mayor precisión los *BIC*, o al menos, para comprobar que existen en ciertos casos.

Finalmente, otro trabajo futuro interesante sería replicar estudios anteriores basados en analizar el origen del error, y que posean alto impacto en la comunidad pero usando nuestro modelo propuesto. Estos estudios estarían relacionados con la prevención, la detección y clasificación de errores y se quntificaría como afecta en gran escala la asunción en la que se basa el algoritmo SZZ y sus derivados.

Bibliography

[Abreu and Premraj, 2009] Abreu, R. and Premraj, R. (2009). How developer communication frequency relates to bug introducing changes. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 153–158. ACM.

[Abreu et al., 2009] Abreu, R., Zoeteweij, P., Golsteijn, R., and Van Gemund, A. J. (2009). A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792.

[Abreu et al., 2007] Abreu, R., Zoeteweij, P., and Van Gemund, A. J. (2007). On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE.

[Amann et al., 2015] Amann, S., Beyer, S., Kevic, K., and Gall, H. (2015). Software mining studies: Goals, approaches, artifacts, and replicability. In *Software Engineering*, pages 121–158. Springer.

[Ando et al., 2015] Ando, R., Sato, S., Uchida, C., Washizaki, H., Fukazawa, Y., Inoue, S., Ono, H., Hanai, Y., Kanazawa, M., Sone, K., et al. (2015). How does defect removal activity of developer vary with development experience? In *SEKE*, pages 540–545.

[Antoniol et al., 2008] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. (2008). Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, page 23. ACM.

[Anvik et al., 2006] Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM.

[Aranda and Venolia, 2009] Aranda, J. and Venolia, G. (2009). The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st international conference on software engineering*, pages 298–308. IEEE Computer Society.

[Artho, 2001] Artho, C. (2001). Finding faults in multi-threaded programs.

[Asadollah et al., 2015] Asadollah, S. A., Hansson, H., Sundmark, D., and Eldh, S. (2015). Towards classification of concurrency bugs based on observable properties. In *Complex Faults and Failures in Large Software Systems (COUFLESS), 2015 IEEE/ACM 1st International Workshop on*, pages 41–47. IEEE.

[Asaduzzaman et al., 2012] Asaduzzaman, M., Bullock, M. C., Roy, C. K., and Schneider, K. A. (2012). Bug introducing changes: A case study with android. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 116–119. IEEE Press.

[Bachmann and Bernstein, 2009] Bachmann, A. and Bernstein, A. (2009). Software process data quality and characteristics: a historical view on open and closed source projects. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 119–128. ACM.

[Bachmann et al., 2010] Bachmann, A., Bird, C., Rahman, F., Devanbu, P., and Bernstein, A. (2010). The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM.

[Baker and Eick, 1994] Baker, M. J. and Eick, S. G. (1994). Visualizing software systems. In *Proceedings of the 16th international conference on Software engineering*, pages 59–67. IEEE Computer Society Press.

[Basili et al., 1999] Basili, V. R., Shull, F., and Lanubile, F. (1999). Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473.

[Bavota et al., 2012] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., and Strollo, O. (2012). When does a refactoring induce bugs? an empirical study. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 104–113. IEEE.

[Bavota and Russo, 2015] Bavota, G. and Russo, B. (2015). Four eyes are better than two: On the impact of code reviews on software quality. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 81–90. IEEE.

[Baysal et al., 2009] Baysal, O., Godfrey, M. W., and Cohen, R. (2009). A bug you like: A framework for automated assignment of bugs. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 297–298. IEEE.

[Beizer, 2003] Beizer, B. (2003). *Software testing techniques*. Dreamtech Press.

[Bennett and Rajlich, 2000] Bennett, K. H. and Rajlich, V. T. (2000). Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM.

[Bettenburg and Hassan, 2013] Bettenburg, N. and Hassan, A. E. (2013). Studying the impact of social interactions on software quality. *Empirical Software Engineering*, 18(2):375–431.

[Bettenburg et al., 2008] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2008). What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318. ACM.

[Binkley, 1992] Binkley, D. (1992). Using semantic differencing to reduce the cost of regression testing. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 41–50. IEEE.

[Bird et al., 2009a] Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., and Devanbu, P. (2009a). Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM.

[Bird et al., 2010] Bird, C., Bachmann, A., Rahman, F., and Bernstein, A. (2010). Linkster: enabling efficient manual inspection and annotation of mined data. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 369–370. ACM.

[Bird et al., 2009b] Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., and Devanbu, P. (2009b). The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE.

[Bissyande et al., 2013] Bissyande, T. F., Thung, F., Wang, S., Lo, D., Jiang, L., and Reveillere, L. (2013). Empirical evaluation of bug linking. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 89–98. IEEE.

[Blondeau et al., 2017] Blondeau, V., Etien, A., Anquetil, N., Cresson, S., Croisy, P., and Ducasse, S. (2017). What are the testing habits of developers? a case study in a large it company. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 58–68. IEEE.

[Bowes et al., 2017] Bowes, D., Hall, T., Petrić, J., Shippey, T., and Turhan, B. (2017). How good are my tests? In *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*, pages 9–14. IEEE Press.

[Brooks, 1974] Brooks, F. P. (1974). The mythical man-month. *Datamation*, 20(12):44–52.

[Brun et al., 2013] Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. (2013). Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, pages 1358–1375.

[Cao, 2015] Cao, Y. (2015). *Investigating the Impact of Personal, Temporal and Participation Factors on Code Review Quality*. PhD thesis, Universite de Montreal.

[Chandra and Chen, 2000] Chandra, S. and Chen, P. M. (2000). Whither generic recovery from application faults? a fault study using open-source software. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 97–106. IEEE.

[Chen et al., 2014] Chen, T.-H., Nagappan, M., Shihab, E., and Hassan, A. E. (2014). An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 82–91. ACM.

[Chou et al., 2001] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. (2001). An empirical study of operating systems errors. In *ACM SIGOPS Operating Systems Review*, pages 73–88. ACM.

[Ciancarini and Sillitti, 2016] Ciancarini, P. and Sillitti, A. (2016). A model for predicting bug fixes in open source operating systems: an empirical study. In *28th International Conference on Software Engineering and Knowledge Engineering (SEKE 2016), Redwood City, San Francisco Bay, CA, USA*, pages 1–3.

[Clarke and Oxman, 2000] Clarke, M. and Oxman, A. (2000). *Cochrane reviewers' handbook*. Update Software.

[Cleve and Zeller, 2005] Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 342–351. IEEE.

[Copeland, 2005] Copeland, T. (2005). Pmd applied.

[Cruzes and Dybå, 2011] Cruzes, D. S. and Dybå, T. (2011). Research synthesis in software engineering: A tertiary study. *Information and Software Technology*, 53(5):440–455.

[Čubranić and Murphy, 2003] Čubranić, D. and Murphy, G. C. (2003). Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th international Conference on Software Engineering*, pages 408–418. IEEE Computer Society.

[da Costa et al., 2014] da Costa, D. A., Kulesza, U., Aranha, E., and Coelho, R. (2014). Unveiling developers contributions behind code commits: An exploratory study. In *Pro-*

ceedings of the 29th Annual ACM Symposium on Applied Computing, pages 1152–1157. ACM.

[da Costa et al., 2016] da Costa, D. A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., and Hassan, A. (2016). A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering (in press)*.

[Davies et al., 2014] Davies, S., Roper, M., and Wood, M. (2014). Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process*, 26(1):107–139.

[Duraes and Madeira, 2006] Duraes, J. A. and Madeira, H. S. (2006). Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11).

[El Mezouar et al., 2017] El Mezouar, M., Zhang, F., and Zou, Y. (2017). Are tweets useful in the bug fixing process? an empirical study on firefox and chrome. *Empirical Software Engineering*, pages 1–39.

[Ell, 2013] Ell, J. (2013). Identifying failure inducing developer pairs within developer networks. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1471–1473.

[Erlikh, 2000] Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT professional*, 2(3):17–23.

[Ferzund et al., 2009] Ferzund, J., Ahsan, S. N., and Wotawa, F. (2009). Software change classification using hunk metrics. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 471–474. IEEE.

[Fischer et al., 2003a] Fischer, M., Pinzger, M., and Gall, H. (2003a). Analyzing and relating bug report data for feature tracking. In *WCRE*, volume 3, page 90.

[Fischer et al., 2003b] Fischer, M., Pinzger, M., and Gall, H. (2003b). Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE.

[Flanagan et al., 2013] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2013). Pldi 2002: Extended static checking for java. *ACM Sigplan Notices*, pages 22–33.

[Forrest et al., 2009] Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. (2009). A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM.

[Fukushima et al., 2014] Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., and Ubayashi, N. (2014). An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 172–181. ACM.

[German et al., 2009] German, D. M., Hassan, A. E., and Robles, G. (2009). Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology*, 51(10):1394–1408.

[González-Barahona and Robles, 2012] González-Barahona, J. M. and Robles, G. (2012). On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17(1-2):75–89.

[Gousios, 2013] Gousios, G. (2013). The ghtorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 233–236. IEEE Press.

[Gu et al., 2003] Gu, W., Kalbarczyk, Z., Iyer, K., Yang, Z., et al. (2003). Characterization of linux kernel behavior under errors. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 459–468. IEEE.

[Guo et al., 2010] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2010). Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 495–504. IEEE.

[Gupta et al., 2005] Gupta, N., He, H., Zhang, X., and Gupta, R. (2005). Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272. ACM.

[Harris et al., 2010] Harris, T., Larus, J., and Rajwar, R. (2010). Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263.

[Harrold et al., 2000] Harrold, M. J., Rothermel, G., Sayre, K., Wu, R., and Yi, L. (2000). An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing Verification and Reliability*, pages 171–194.

[Hassan, 2009] Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 78–88. IEEE.

[Hassan and Holt, 2005] Hassan, A. E. and Holt, R. C. (2005). The top ten list: Dynamic fault prediction. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 263–272. IEEE.

[Hata et al., 2010] Hata, H., Mizuno, O., and Kikuno, T. (2010). Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Software Engineering*, 15(2):147–165.

[Hata et al., 2012] Hata, H., Mizuno, O., and Kikuno, T. (2012). Bug prediction based on fine-grained module histories. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 200–210. IEEE.

[Herzig et al., 2013] Herzig, K., Just, S., and Zeller, A. (2013). It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press.

[Herzig and Zeller, 2013] Herzig, K. and Zeller, A. (2013). The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 121–130. IEEE Press.

[Hindle et al., 2008] Hindle, A., German, D. M., and Holt, R. (2008). What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108. ACM.

[Horwitz, 1990] Horwitz, S. (1990). *Identifying the semantic and textual differences between two versions of a program*. ACM.

[Hovemeyer and Pugh, 2004] Hovemeyer, D. and Pugh, W. (2004). Finding bugs is easy. *Acm sigplan notices*, pages 92–106.

[Izquierdo et al., 2011] Izquierdo, D., Capiluppi, A., and Gonzalez-Barahona, J. M. (2011). Are developers fixing their own bugs?: Tracing bug-fixing and bug-seeding committers. In *Open Source Software Dynamics, Processes, and Applications*, pages 79–98. IGI Global.

[Izquierdo-Cortázar et al., 2012] Izquierdo-Cortázar, D., Robles, G., and González-Barahona, J. (2012). Do more experienced developers introduce fewer bugs? *Open Source Systems: Long-Term Sustainability*, pages 268–273.

[Janssen et al., 2009] Janssen, T., Abreu, R., and Van Gemund, A. J. (2009). Zoltar: a spectrum-based fault localization tool. In *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution@ runtime*, pages 23–30. ACM.

[Jeffrey et al., 2008] Jeffrey, D., Gupta, N., and Gupta, R. (2008). Identifying the root causes of memory bugs using corrupted memory location suppression. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 356–365. IEEE.

[Jelinski and Moranda, 1972] Jelinski, Z. and Moranda, P. (1972). Software reliability research. In *Statistical computer performance evaluation*, pages 465–484. Elsevier.

[Jiang et al., 2013] Jiang, T., Tan, L., and Kim, S. (2013). Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 279–289. IEEE Press.

[Jones et al., 2002] Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM.

[Jongyindee et al., 2011] Jongyindee, A., Ohira, M., Ihara, A., and Matsumoto, K.-i. (2011). Good or bad committers? a case study of committers' cautiousness and the consequences on the bug fixing process in the eclipse project. In *Software Measurement, 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 116–125. IEEE.

[Juristo and Vegas, 2009] Juristo, N. and Vegas, S. (2009). Using differences among replications of software engineering experiments to gain knowledge. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 356–366. IEEE Computer Society.

[Kagdi et al., 2008] Kagdi, H., Hammad, M., and Maletic, J. I. (2008). Who can help me with this source code change? In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 157–166. IEEE.

[Kamei et al., 2010] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B., and Hassan, A. E. (2010). Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE.

[Kamei et al., 2013] Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773.

[Kan, 2002] Kan, S. H. (2002). *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc.

[Kaner, 2006] Kaner, C. (2006). Exploratory testing. In *Quality assurance institute worldwide annual software testing conference*.

[Kawrykow and Robillard, 2011] Kawrykow, D. and Robillard, M. P. (2011). Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 351–360. ACM.

[Kim and Ernst, 2007] Kim, S. and Ernst, M. D. (2007). Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference*

and the ACM SIGSOFT symposium on The foundations of software engineering, pages 45–54. ACM.

[Kim et al., 2006a] Kim, S., Pan, K., and Whitehead Jr, E. (2006a). Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45. ACM.

[Kim et al., 2006b] Kim, S., Whitehead, E. J., et al. (2006b). Properties of signature change patterns. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 4–13. IEEE.

[Kim and Whitehead Jr, 2006] Kim, S. and Whitehead Jr, E. J. (2006). How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174. ACM.

[Kim et al., 2008] Kim, S., Whitehead Jr, E. J., and Zhang, Y. (2008). Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196.

[Kim et al., 2006c] Kim, S., Zimmermann, T., Pan, K., James Jr, E., et al. (2006c). Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE.

[Kim et al., 2007] Kim, S., Zimmermann, T., Whitehead Jr, E. J., and Zeller, A. (2007). Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society.

[Kitchenham and Charters, 2007] Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering.

[Laprie, 1985] Laprie, J.-C. (1985). Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11.

[LaToza et al., 2006] LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM.

[Le, 2016] Le, D. (2016). Architectural-based speculative analysis to predict bugs in a software system. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 807–810. IEEE.

[Le et al., 2015] Le, T.-D. B., Linares-Vásquez, M., Lo, D., and Poshyvanyk, D. (2015). Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, pages 36–47. IEEE.

[Le Goues et al., 2012] Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE.

[Lehman, 1979] Lehman, M. M. (1979). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221.

[Lehman et al., 1998] Lehman, M. M., Ramil, J. F., and Perry, D. E. (1998). On evidence supporting the feast hypothesis and the laws of software evolution. In *metrics*, page 84. IEEE.

[Lerner, 1994] Lerner, M. (1994). Software maintenance crisis resolution: The new ieee standard. *Software Development*, 2(8):65–72.

[Li et al., 2006] Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., and Zhai, C. (2006). Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM.

[Lu et al., 2005] Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., and Zhou, Y. (2005). Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5.

[Madeyski and Kitchenham, 2017] Madeyski, L. and Kitchenham, B. (2017). Would wider adoption of reproducible research be beneficial for empirical software engineering research? *Journal of Intelligent & Fuzzy Systems*, 32(2):1509–1521.

[Massacci and Nguyen, 2014] Massacci, F. and Nguyen, V. H. (2014). An empirical methodology to evaluate vulnerability discovery models. *IEEE Transactions on Software Engineering*, 40(12):1147–1162.

[McConnell, 2004] McConnell, S. (2004). *Code complete*. Pearson Education.

[McIntosh et al., 2016] McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189.

[Mens, 2008] Mens, T. (2008). Introduction and roadmap: History and challenges of software evolution, chapter 1. software evolution.

[Mens et al., 2005] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. (2005). Challenges in software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22. IEEE.

[Misherghi and Su, 2006] Misherghi, G. and Su, Z. (2006). Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM.

[Mockus and Votta, 2000] Mockus, A. and Votta, L. G. (2000). Identifying reasons for software changes using historic databases. In *icsm*, pages 120–130.

[Monperrus, 2014] Monperrus, M. (2014). A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM.

[Murphy-Hill et al., 2015] Murphy-Hill, E., Zimmermann, T., Bird, C., and Nagappan, N. (2015). The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81.

[Nagappan et al., 2006] Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM.

[Ness and Ngo, 1997] Ness, B. and Ngo, V. (1997). Regression containment through source change isolation. In *Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International*, pages 616–621. IEEE.

[Neto et al., 2018] Neto, E. C., da Costa, D. A., and Kulesza, U. (2018). The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390. IEEE.

[Nguyen et al., 2012] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., and Nguyen, T. N. (2012). Multi-layered approach for recovering links between bug reports and fixes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 63. ACM.

[Nguyen et al., 2010] Nguyen, T. H., Adams, B., and Hassan, A. E. (2010). A case study of bias in bug-fix datasets. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 259–268. IEEE.

[Nguyen and Massacci, 2013] Nguyen, V. H. and Massacci, F. (2013). The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 493–498. ACM.

[Ostrand and Weyuker, 1984] Ostrand, T. J. and Weyuker, E. J. (1984). Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, 4(4):289–300.

[Ostrand and Weyuker, 2002] Ostrand, T. J. and Weyuker, E. J. (2002). The distribution of faults in a large industrial software system. In *ACM SIGSOFT Software Engineering Notes*, pages 55–64. ACM.

[Pan et al., 2009] Pan, K., Kim, S., and Whitehead, E. J. (2009). Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315.

[Pan et al., 2006] Pan, K., Kim, S., and Whitehead Jr, E. J. (2006). Bug classification using program slicing metrics. In *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop on*, pages 31–42. IEEE.

[Panichella et al., 2014a] Panichella, S., Bavota, G., Di Penta, M., Canfora, G., and Antoniol, G. (2014a). How developers' collaborations identified from different sources tell us about code changes. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 251–260. IEEE.

[Panichella et al., 2014b] Panichella, S., Canfora, G., Di Penta, M., and Oliveto, R. (2014b). How the evolution of emerging collaborations relates to code changes: an empirical study. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 177–188. ACM.

[Perry et al., 2000] Perry, D. E., Porter, A. A., and Votta, L. G. (2000). Empirical studies of software engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 345–355. ACM. (F,1).

[Pham, 2000] Pham, H. (2000). *Software reliability*. Springer Science & Business Media.

[Piwowar et al., 2007] Piwowar, H. A., Day, R. S., and Fridsma, D. B. (2007). Sharing detailed research data is associated with increased citation rate. *PloS one*, 2(3):e308.

[Planning, 2002] Planning, S. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*.

[Podgurski et al., 2003] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., and Wang, B. (2003). Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 465–475. IEEE.

[Posnett et al., 2013] Posnett, D., D'Souza, R., Devanbu, P., and Filkov, V. (2013). Dual ecological measures of focus in software development. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 452–461. IEEE.

[Prechelt and Pepper, 2014] Prechelt, L. and Pepper, A. (2014). Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. *Information and Software Technology*, 56(10):1377–1389.

[Purushothaman and Perry, 2004] Purushothaman, R. and Perry, D. E. (2004). Towards understanding the rhetoric of small changes-extended abstract. In *International Workshop on Mining Software Repositories (MSR 2004), International Conference on Software Engineering*, pages 90–94. IET.

[Rahman et al., 2012] Rahman, F., Bird, C., and Devanbu, P. (2012). Clones: What is that smell? *Empirical Software Engineering*, 17(4-5):503–530.

[Rahman and Devanbu, 2011] Rahman, F. and Devanbu, P. (2011). Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM.

[Rahman et al., 2014] Rahman, F., Khatri, S., Barr, E. T., and Devanbu, P. (2014). Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 424–434. ACM.

[Rao and Kak, 2011] Rao, S. and Kak, A. (2011). Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52. ACM.

[Renieres and Reiss, 2003] Renieres, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 30–39. IEEE.

[Reps et al., 1997] Reps, T., Ball, T., Das, M., and Larus, J. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem. In *Software Engineering-Esec/Fse 1997*, pages 432–449. Springer.

[Robles, 2010] Robles, G. (2010). Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories proceedings. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 171–180. IEEE.

[Rodríguez-Pérez et al., 2016] Rodríguez-Pérez, G., Gonzalez-Barahona, J. M., Robles, G., Dalipaj, D., and Sekitoleko, N. (2016). Bugtracking: A tool to assist in the identification of bug reports. In *IFIP International Conference on Open Source Systems*, pages 192–198. Springer.

[Rodríguez-Pérez et al., 2018] Rodríguez-Pérez, G., Robles, G., and González-Barahona, J. M. (2018). Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*.

[Rosen et al., 2015] Rosen, C., Grawi, B., and Shihab, E. (2015). Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 966–969. ACM.

[Rosenthal and DiMatteo, 2001] Rosenthal, R. and DiMatteo, M. R. (2001). Meta-analysis: Recent developments in quantitative methods for literature reviews. *Annual review of psychology*, 52(1):59–82.

[Rosso et al., 2016] Rosso, D., Perez, S., et al. (2016). Purposes, concepts, misfits, and a redesign of git. In *ACM SIGPLAN Notices*, pages 292–310. ACM.

[Sahoo et al., 2010] Sahoo, S. K., Criswell, J., and Adve, V. (2010). An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 485–494. IEEE.

[Schröter et al., 2006] Schröter, A., Zimmermann, T., Premraj, R., and Zeller, A. (2006). If your bug database could talk. In *Proceedings of the 5th international symposium on empirical software engineering*, volume 2, pages 18–20. Citeseer.

[Servant and Jones, 2017] Servant, F. and Jones, J. A. (2017). Fuzzy fine-grained code-history analysis. In *Proceedings of the 39th International Conference on Software Engineering*, pages 746–757. IEEE Press.

[Shippey, 2015] Shippey, T. J. (2015). *Exploiting Abstract Syntax Trees to Locate Software Defects*. PhD thesis, University of Hertfordshire.

[Shivaji, 2013] Shivaji, S. (2013). *Efficient bug prediction and fix suggestions*. PhD thesis, University of California, Santa Cruz.

[Shivaji et al., 2013] Shivaji, S., Whitehead, E. J., Akella, R., and Kim, S. (2013). Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569.

[Shull et al., 2008] Shull, F. J., Carver, J. C., Vegas, S., and Juristo, N. (2008). The role of replications in empirical software engineering. *Empirical software engineering*, 13(2):211–218.

[Sinha et al., 2010] Sinha, V. S., Sinha, S., and Rao, S. (2010). Buginnings: identifying the origins of a bug. In *Proceedings of the 3rd India software engineering conference*, pages 3–12. ACM.

[Śliwerski et al., 2005a] Śliwerski, J., Zimmermann, T., and Zeller, A. (2005a). Hatari: raising risk awareness. In *ACM SIGSOFT Software Engineering Notes*, pages 107–110. ACM.

[Śliwerski et al., 2005b] Śliwerski, J., Zimmermann, T., and Zeller, A. (2005b). When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM.

[Sommerville, 2010] Sommerville, I. (2010). *Software engineering*. New York: Addison-Wesley.

[Sullivan and Chillarege, 1992] Sullivan, M. and Chillarege, R. (1992). A comparison of software defects in database management systems and operating systems. In *FTCS*, pages 475–484.

[Sun et al., 2017] Sun, Y., Wang, Q., and Yang, Y. (2017). Frlink: Improving the recovery of missing issue-commit links by revisiting file relevance. *Information and Software Technology*, 84:33–47.

[Tan et al., 2014] Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., and Zhai, C. (2014). Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705.

[Tan et al., 2015] Tan, M., Tan, L., Dara, S., and Mayeux, C. (2015). Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 99–108. IEEE Press.

[Thomas et al., 2013] Thomas, S. W., Nagappan, M., Blostein, D., and Hassan, A. E. (2013). The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering*, 39(10):1427–1443.

[Thung et al., 2014] Thung, F., Le, T.-D. B., Kochhar, P. S., and Lo, D. (2014). Buglocalizer: integrated tool support for bug localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 767–770. ACM.

[Thung et al., 2013] Thung, F., Lo, D., and Jiang, L. (2013). Automatic recovery of root causes from bug-fixing changes. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 92–101. IEEE.

[Thung et al., 2012] Thung, F., Lo, D., Jiang, L., Rahman, F., Devanbu, P. T., et al. (2012). To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. ACM.

[Tiwari et al., 2011] Tiwari, S., Mishra, K., Kumar, A., and Misra, A. K. (2011). Spectrum-based fault localization in regression testing. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 191–195. IEEE.

[Trockman, 2018] Trockman, A. (2018). Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 524–526. ACM.

[Vahabzadeh et al., 2015] Vahabzadeh, A., Fard, A. M., and Mesbah, A. (2015). An empirical study of bugs in test code. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 101–110. IEEE.

[VanHilst et al., 2011] VanHilst, M., Huang, S., and Lindsay, H. (2011). Process analysis of a waterfall project using repository data. *International Journal of Computers and Applications*, 33(1):49–56.

[Wan et al., 2017] Wan, Z., Lo, D., Xia, X., and Cai, L. (2017). Bug characteristics in blockchain systems: a large-scale empirical study. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 413–424. IEEE.

[Wehaibi et al., 2016] Wehaibi, S., Shihab, E., and Guerrouj, L. (2016). Examining the impact of self-admitted technical debt on software quality. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 179–188. IEEE.

[Weimer et al., 2009] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society.

[Weiss et al., 2007] Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. (2007). How long will it take to fix this bug? In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 1–1. IEEE.

[Wen et al., 2016] Wen, M., Wu, R., and Cheung, S.-C. (2016). Locus: Locating bugs from software changes. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 262–273. IEEE.

[Williams and Spacco, 2008] Williams, C. and Spacco, J. (2008). SZZ revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM.

[Wohlin et al., 2012] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

[Wu et al., 2011] Wu, R., Zhang, H., Kim, S., and Cheung, S.-C. (2011). Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM.

[Yamada and Mizuno, 2014] Yamada, A. and Mizuno, O. (2014). A text filtering based approach to classify bug injected and fixed changes. In *Advanced Applied Informatics (IIAIAAI), 2014 IIAI 3rd International Conference on*, pages 680–686. IEEE.

[Yang et al., 2014] Yang, H., Wang, C., Shi, Q., Feng, Y., and Chen, Z. (2014). Bug inducing analysis to prevent fault prone bug fixes. In *SEKE*, pages 620–625.

[Youm et al., 2015] Youm, K. C., Ahn, J., Kim, J., and Lee, E. (2015). Bug localization based on code change histories and bug reports. In *Software Engineering Conference (APSEC), 2015 Asia-Pacific*, pages 190–197. IEEE.

[Yuan et al., 2013] Yuan, Z., Yu, L., Liu, C., and Zhang, L. (2013). Predicting bugs in source code changes with incremental learning method. *Journal of Software*, 8(7):1620–1634.

[Zeller, 1999] Zeller, A. (1999). Yesterday, my program worked. today, it does not. why? In *ACM SIGSOFT Software engineering notes*, pages 253–267. Springer-Verlag.

[Zeller, 2002] Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM.

[Zeller, 2003] Zeller, A. (2003). Causes and effects in computer programs. *arXiv preprint cs/0309047*.

[Zeller and Hildebrandt, 2002] Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200.

[Zhang et al., 2014] Zhang, F., Mockus, A., Keivanloo, I., and Zou, Y. (2014). Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191. ACM.

[Zhang et al., 2013] Zhang, H., Gong, L., and Versteeg, S. (2013). Predicting bug-fixing time: an empirical study of commercial software projects. In *Proceedings of the 2013 international conference on software engineering*, pages 1042–1051. IEEE Press.

[Zhivich and Cunningham, 2009] Zhivich, M. and Cunningham, R. K. (2009). The real cost of software errors. *IEEE Security & Privacy*, 7(2).

[Zimmermann et al., 2006] Zimmermann, T., Kim, S., Zeller, A., and Whitehead Jr, E. J. (2006). Mining version archives for co-changed lines. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75. ACM.

[Zimmermann et al., 2007] Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. In *Proceedings of the third international workshop on predictor models in software engineering*, page 9. IEEE Computer Society.