Empirical Software Engineering (2024) 29:66
https://doi.org/10.1007/s10664-024-10479-z

Hunting bugs: Towards an automated approach to identifying which change caused a bug through regression testing

Michel Maes-Bermejo[1] · Alexander Serebrenik[2] · Micael Gallego[1] · Francisco Gortázar[1] · Gregorio Robles[3] · Jesús María González Barahona[3]

Accepted: 19 March 2024

Abstract

Context Finding code changes that introduced bugs is important both for practitioners and researchers, but doing it precisely is a manual, effort-intensive process. The *perfect test* method is a theoretical construct aimed at detecting Bug-Introducing Changes (BIC) through a theoretical *perfect test*. This *perfect test* always fails if the bug is present, and passes otherwise.

Objective To explore a possible automatic operationalization of the *perfect test* method.

Method To use regression tests as substitutes for the *perfect test*. For this, we transplant the regression tests to past snapshots of the code, and use them to identify the BIC, on a well-known collection of bugs from the Defects4J dataset.

Results From 809 bugs in the dataset, when running our operationalization of the perfect test method, for 95 of them the BIC was identified precisely and in the remaining 4 cases, a list of candidates including the BIC was provided.

Conclusions We demonstrate that the operationalization of the *perfect test* method through regression tests is feasible and can be completely automated in practice when tests can be transplanted and run in past snapshots of the code. Given that implementing regression tests when a bug is fixed is considered a good practice, when developers follow it, they can detect effortlessly bug-introducing changes by using our operationalization of the *perfect test* method.

Keywords Bug origins · Bug-introducing changes · First-failing change · SZZ algorithm · Software Testing · Regression Testing

Communicated by: Saurabh Sinha

✉ Michel Maes-Bermejo
michel.maes@urjc.es

Extended author information available on the last page of the article

1 Introduction

Bugs are one of the main sources of concern for software developers. Finding faulty code, and fixing it to remove the bug consumes a lot of effort (Kim and Whitehead 2006; Weiss et al. 2007). Learning about how bugs were introduced is both of academic and practical importance. Practitioners would like to know which change introduced a bug when they try to fix it, and which past versions are affected by it. Researchers want to study how bugs were introduced, to find ways of preventing them. However, finding the source code change that introduced the bug [1]) is not easy.

Many methods for finding the change to the source code that introduced the bug (bug-introducing change, BIC) assume that the change should touch some or all of the lines of code that were touched to fix the bug. This is the case, e.g., for all SZZ-based algorithms (Śliwerski et al. 2005). However, a recent work by F. Petrulio et al. (2022) has proven some limitations of this approach. For a dataset with 5,348 bugs where the BIC was available, 1,176 bugs could not be identified with an SZZ implementation (22% of the total). The main reasons of the flaws are (1) changes with only new lines (that because of the way SZZ works cannot be caught) and (2) bugs introduced in changes which are not in previous changes to the lines touched in the fixing commits.

Instead of relying on this assumption, the *perfect test* method was introduced by Rodríguez-Pérez et al. to find what change introduced a given bug (Rodríguez-Pérez et al. 2020). The *perfect test* for a bug is a theoretical construct that fails in any snapshot of the code history affected by the bug, and succeeds in any other snapshot. The BIC, therefore, can be assumed to be the change that produced the first snapshot (of a sequence of snapshots going from the last commit where the test passed to the fix commit) that causes the test to fail. This method provides a clear guideline to decide whether a change introduces a bug or not, but operationalizing it requires having *perfect tests*.

In this paper we study to which extent regression tests can be used as *perfect tests*. Regression tests (Desikan and Ramesh 2006; Wahl 1999) are written to avoid regressions [2], by detecting the reintroduction in future changes of an already fixed bug. Both in closed-source projects from the industry (Ali et al. 2019; Onoma et al. 1998; Engström and Runeson 2010) and open source ones (Schmidt and Porter 2001), regression tests are usually used for the bugs that are fixed (Perscheid et al. 2017), which means that, in theory, if these tests can be used as *perfect tests*, we have a way of automatically detecting when those bugs were introduced. For example, the BICs in the two cases found by Petrulio et al. would be found using the perfect test, since the BICs are not identified by some procedure depending on tracking past changes, but by checking the behavior of the program after any change, determining this way if the bug is present or not

To be useful as *perfect tests*, regression tests should be perfect detectors of the bug, and be "transplantable" to past snapshots [3]. By "transplantable" to a past snapshot we mean that the test should be executed in that past snapshot. Regression tests are usually assumed to be almost-perfect detectors, which is the whole idea of spending effort in writing them. For

[1] In this paper, we will use the term "bug", widely used by practitioners, as synonym for "defect" or "fault", which is more usual in academic literature (IEEE Standard Classification for Software Anomalies 2010; Avizienis et al. 2004; Tan et al. 2014

[2] In this paper, by "regression" we mean a change in the source code that breaks a functionality that was working properly.

[3] A snapshot is a version of the source code of a project, represented by the source code as it is after checking out a commit of its git repository which is identified by the unique hash of the commit (Maes-Bermejo et al. 2022).

the rest of this paper, we will assume that regression tests are perfect detectors, and we will focus on transplantability, which would be the only blocker to decide if they can be used. To transplant a test to a past snapshot, the regression test should be built (compiled, in the case of compiled languages), and run correctly (producing a "fail" or "success" output consistently). We refer to these problems as the "compilability problem" and the "runnability problem" for transplanting tests.

By the "compilability problem" we mean (a) snapshot compilability, or the set of issues that prevent the building of the source code of a past snapshot of a software project, and (b) transplant compilability, or the set of issues that prevent the transplanted test (in our case, the regression test) from being built. The (a) part of this problem has been addressed previously by researchers (Tufano et al. 2017; Maes-Bermejo et al. 2022). However, in our case we also need to study (b), i.e., the compilability of the regression test that has been transplanted into the commit under scrutiny.

By "runnability problem" we mean the set of issues that prevent the transplanted test (in our case, the regression test) from being executed, and to the best of our knowledge this problem has not been studied before.

Our research questions will be, therefore:

$\mathbf{RQ_{1A}}$: "How far can a test be transplanted into the past?" We will study the extent to which we can build and run regression tests in the past. Given that we do not know how far in the past is the BIC, the further we can transplant a test the more probably we could detect the BIC with the *perfect test* method.

$\mathbf{RQ_{1B}}$: "How compilability and runnability problems impact the transplantation of the regression tests to the past?" In this RQ we study the compilability of the source code of past snapshots, the compilability of the transplanted tests code within the snapshots, and the runnability of the transplanted tests in the context of the past snapshots.

$\mathbf{RQ_2}$: "Can the BIC for a given bug be found using its regression test?" We will study whether regression tests are a real and practical operationalization of the theoretical *perfect test* proposed by Rodríguez-Pérez et al. to detect the change that introduced the bug.

To answer these questions we built a tool to implement the *perfect test* method, by automatically compiling and executing regression tests in past snapshots. The tool traverses the history of the code, which is usually not linear but a graph of different branches, transplanting the test to the snapshots previous to the bug fixing change (BFC). We applied the tool on a well-known bug dataset of 835 bugs and their corresponding BFCs, Defects4J (Just et al. 2014). Since Defects4J does not identify the BICs corresponding to the bugs in the dataset, we validated our results manually, by checking whether the identified BIC was really the change that introduced the bug. As a result of this process, we created a new manually validated dataset, based on Defects4J, which we will denominate BIC-RT (**B**ug **I**ntroducing **C**hanges detected by **R**egression **T**ests)

The rest of the paper is structured as follows: Section 2 discusses previous research. Section 3 presents the methodology used in the studies and defines the terminology. The results of applying the methodology are reported in Section 4. Section 5 discusses the results, and explores threats to their validity. Finally, Section 6 draws conclusions and presents further research.

2 Related work

2.1 Transplanting code

Our proposal for identifying BICs is based on transplanting a test present in the snapshot that fixed a bug to earlier snapshots of the same code. Techniques for transplanting pieces of code were applied by ReDeBug (Jang et al. 2012) for fixing clones of a fixed bug, by transplanting the fixing patch. In this respect, the *plastic surgery hypothesis* (Harman 2010), which assumes that changes to source code can be constructed as combinations of other changes already present in the code (grafts) was studied in detail by Barr et al. (2014), showing how in fact patches could in many cases be transplanted to other areas of code. TransplantFix (Yang et al. 2022) is a recent approach that lies in the scope of using transplantation to automatically fix buggy programs. Automated software transplantation has been also used to transfer functionality from one system to another (Barr et al. 2015; Sidiroglou-Douskos et al. 2017) and to generate tests by reusing test code (Zhang and Kim 2017). However, further studies (Castelluccio et al. 2019) show how this practice can lead to regressions in the version where the fix is applied.

To determine which specific piece of code of a change in a bug fix is the part actually fixes the bug, BugBuilder (Jang et al. 2021) transplants each part to the previous versions of the code. It then runs the regression test to check whether the bug is still present. This approach is similar to ours, with the difference that the BugBuilder authors aim to find out which part of a change is the fix, while we intend to find when the bug was introduced, and for that we need to transplant not only to the previous version, but to many others in the past of the bug fix.

2.2 Bug Introduction Changes

The problem of detecting the change that introduced a bug (bug-introducing change, BIC) given the change that fixed it (bug fixing change, BFC) has been extensively studied (Sinha et al. 2010; Davies et al. 2014; Śliwerski et al. 2005; Kim et al. 2006; Williams and Spacco 2008; Kamei et al. 2012; Tantithamthavorn et al. 2013). The usual approach has been to assume that the change that introduced the bug touched the same lines of source code that were touched to fix it. The SZZ algorithm (Śliwerski et al. 2005) was developed based on this assumption. Using the source control management system, it identifies the lines that were edited in the change that fixed a commit, and then which previous changes modified the same lines before the bug was reported. Many variants of SZZ have been proposed, improving the algorithm in different ways. Rodríguez-Pérez et al. (2018) surveyed 187 studies related to SZZ, finding that 38% of them used the original algorithm, while the rest used a derivative. It also found a very low reproducibility for the studies, with many of them not publishing the implementation of the algorithms they used. Fortunately, this situation is changing, and some years later we have several implementations of SZZ derivatives published (Borg et al. 2019; Lenarduzzi et al. 2020; Pokropiński et al. 2022; Rosa et al. 2021).

Another tool to consider for the same purpose is GitBisect [4], which through a binary search, assists the developer to locate the commit that introduced the bug. This tool explores the Git history of a project, asking the developer if the bug is in the current commit or not. It is therefore up to the developer to perform all the build and testing steps manually. An automated

[4] https://git-scm.com/docs/git-bisect

bisection over git bisect was proposed, called "*git bisect run*" [5], which allows the developer to add a script or command to be executed at each step of the tool. As we will see below, our approach proposes a fully automatic process that allows us to obtain much more detailed results from the build and test steps of each commit. Our proposal also solves a limitation of GitBisect: this tool does not consider the graph structure that the commit history of a project may have, while our proposal navigates through the graph with an appropriate algorithm (depth-first search). Some improvements on this technique have been proposed (Saha and Gligoric 2017) by selecting commits and tests to save computational costs.

There are studies that reduce the cost for both bisection and SZZ-like blame models such as (An and Yoo 2021), which filters commits using the coverage of regression tests for the bug, thus reducing the search space for automated bisection.

The algorithm Delta Debugging of Andreas Zeller (Zeller and Hildebrandt 2002; Zeller 2002) uses testing to simplify and isolate the failure in the execution trace of some failing test case. Based on the idea of delta debugging and for the specific case of regression bugs, there are also some techniques that have been proposed. For example, the difference between the last version that worked well and the current version where the bug is present can be used (Saha and Gligoric 2017), or a combination of the information in the issue tracking system and source code management (Khattar et al. 2015).

Recently, the performance of SZZ has been studied (Bludau and Pretschner 2022; Petrulio et al. 2022) in projects that follow the pull-based development model proposed by GitHub (Gousios et al. 2014), showing that in this type of projects it is necessary to consider sets of commits when detecting the change that introduced a bug.

None of these techniques deal with automatically transplanting tests to past versions of the source code. Git bisect and its derivatives do not directly address transplanting code into the past and is limited to a binary search that considers only a linear history model. SZZ and derived techniques try to infer which changes could have introduced the bug by analyzing the history of the source code.

One of the most disruptive studies in the area is that of Rodríguez-Pérez et al. (2020), mentioned above. This study proposes a specific method for deciding if a snapshot has a certain bug or not. For that, it introduces the "perfect test method", a theoretical construct that fails on any snapshot of the code affected by the bug and succeeds otherwise. It is important to notice that this method does not define the BIC based on a method for tracking back changes, but on studying the behavior of the program after a change is applied. Therefore, it has resemblances to the bisection method presented above, but assuming that a perfect test is available, which allows for the automation of the identification procedure. The method determines if a certain snapshot has a bug or not, and therefore, the BIC is simply the first snapshot that has the bug, or in other words, the first snapshot for which the perfect test fails. Our proposal, aims to operationalize the perfect test method by running regression tests on the change history of the project to detect the change that introduced the bug.

3 Methodology

To answer our research questions, we develop a tool that implements the *perfect test* method to find the BIC corresponding to a BFC, by using regression tests as *perfect tests*. A regression test for a bug checks if this bug reappears in changes following the bug fix. Our hypothesis is that a regression test can be used as an approximation to the *perfect test*, which we will

[5] https://lwn.net/Articles/317154/

prove through our tool. Given a BFC and the regression test for its bug, the tool transplants the test to past changes, and tries to execute them, determining if they succeeded or failed. In order to learn how far the tests can be automatically transplanted to past snapshots of the code (RQ_{1A}) and what aspects prevent transplanting the regression test into the past(RQ_{1B}), we run this tool on a dataset consisting of several BFCs and their corresponding regression tests. To learn in which cases the BIC could be found correctly (RQ_2), the tool identifies the BIC using the transplanted regression tests as the *perfect test*.

3.1 The *perfect test* method

As stated by Rodríguez-Pérez et al. (2020), the *perfect test* method to find the BIC corresponding to a BFC assumes that data about changes to the source code (including the BFC and all candidates to be a BIC) can be obtained from a source code management system such as git, in which changes corresponding to fixing bugs can be identified, and related to the description of the corresponding bugs. The *perfect test* method consists of the following steps (Rodríguez-Pérez et al. 2020):

1. Identify a Bug-Fixing Change (BFC) and the description of the corresponding bug.
2. Using the change and the description of the bug, it describe the bug in terms of a *perfect test* that would, with certainty, fail if the bug is present, or succeed if it is not (the *perfect test*).
3. Identify, from the past history of the code, the first change for which the *perfect test* fails (First Failing Change, FFC).

Rodríguez-Pérez et al. (2020) distinguish between intrinsic and extrinsic bugs. *Intrinsic bugs* are bugs that have been introduced by a change in the code. In the case of intrinsic bugs, there should be a BIC, and that will correspond to the FFC: before the BIC, the bug was not present, and after it, it was present until fixed. *Extrinsic bugs* are not introduced by a change to the source code but by an external factor, e.g., a change in an external API. In the case of extrinsic bugs, Rodríguez-Pérez et al. indicate that there is a First-Failing Moment (FFM), not present in the version control system and the FFC is the first change to the version control system after the FFM. Rodríguez-Pérez et al. analyzed how 116 bugs were introduced in the Nova and ElasticSearch projects, and created manually curated datasets for both projects. From those 116 bugs, 60%-64% were intrinsic bugs caused by changes or omissions in the source code of the project, and 9%-21% were extrinsic bugs caused by changes that are not recorded in the source code (Rodríguez-Pérez et al. 2020). In another study using the McIntosh and Kamei's OpenStack dataset (McIntosh and Kamei 2018), out of 1,880 bugs Rodríguez-Pérez et al. identified 1,120 intrinsic bugs (59.6%), 212 (11.3%) extrinsic bugs, and 548 (29.1%) mislabeled bugs (Rodríguez-Pérez et al. 2020); only considering bugs, this means 15.9% of the bugs were extrinsic.

In the current work we focus on intrinsic bugs, and therefore for us finding the FFC will mean we found the BIC. We will not address the detection of FFC in extrinsic bugs since regression tests cannot help us find that change and the bug dataset chosen to test our proposal (to be described in the next section) only contains intrinsic bugs.

For describing the bug in terms of a *perfect test* (2), we use regression tests. Regression tests are designed to detect if bugs are introduced in future changes, and we postulate that they are also useful to detect them in past changes. Therefore, for (3) we run those tests in snapshots of the source code after changes that are previous to the BFC in the history of changes to the source code. Figure 1 illustrates the transplantation process, by showing a

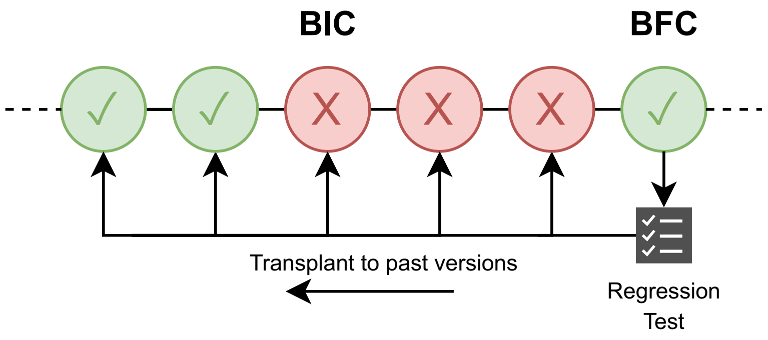


Fig. 1 Simplified process leading to finding the First Failing Change (FFC), by running tests in past snapshots of the project

simplified version of it: we will later discuss on this section how the BIC should be searched considering that the git history is a graph rather than a line.

The paper presenting the *perfect test* method (Rodríguez-Pérez et al. 2020) states that one of the main limitations of the *perfect test* is that being able to construct a *perfect test* requires a deep knowledge of the bug, how it was fixed, and the project in which it was found. In our case, we assume that developers writing regression tests have all this knowledge, and therefore their tests will be close to the theoretical *perfect test* for the bugs they fix.

3.2 The bugs dataset

Defects4J (Just et al. 2014) is a well-known dataset with 835 bugs from 17 Java open source software projects, including only bugs located in the source code, excluding those related to the build system, configuration files, documentation or tests. It has been used as ground truth in the evaluation of several implementations of SZZ-derived algorithms (Neto et al. 2019; Pokropiński et al. 2022; Wen et al. 2019; An and Yoo 2021). Previous studies have identified several issues that might threaten application of SZZ-derived algorithms: e.g., links to the repositories might have changed (Lawrence et al. 2001), repositories might have been deleted, moved, made private, or their history might have been altered (Bird et al. 2009). However, Defects4J includes the whole source code management repositories of each project, avoiding the problems mentioned above. All but one of the repositories in Defects4J are git repositories, which is the source code management we will target in our study.

Every bug included in Defects4J identifies the change (commit) fixing it (its BFC), and refers to a publicly available bug report which details the nature of the bug. From here on we refer to changes as commits, since we will focus on git repositories. This bug report will be required when manually evaluating the detected BICs. Therefore, the dataset complies with step (1) of the *perfect test* method. Associated with each bug there is also a regression test, included in the BFC, that exposes the bug. The dataset also provides its own commands to compile the code and execute the test in this commit. We will use this regression test as a *perfect test* for the bug, therefore complying with step (2) of the *perfect test* method. The authors of the Defects4J dataset have carefully reviewed the regression tests and explicitly mentioned in their work that they have excluded any flaky test. However, we have run each test 3 times in the BFC to verify that their results do not differ, avoiding the inclusion of non-deterministic tests in our experiment.

Table 1 Description of the projects used from Defects4J

Project	# of bugs	# of commits	First Commit	Last Commit
Cli	39	914	2002-06-10	2019-03-25
Closure	174	2,898	2009-11-03	2013-12-13
Codec	18	1,795	2003-04-25	2019-04-23
Collections	4	3,091	2001-04-14	2019-03-25
Compress	47	2,682	2003-11-23	2019-03-25
Csv	16	1,290	2005-12-17	2019-04-14
Gson	18	1,476	2008-09-01	2019-11-05
JacksonCore	26	1,724	2011-12-22	2019-04-24
JacksonDatabind	112	5,241	2011-12-22	2019-05-15
JacksonXml	6	949	2010-12-30	2019-05-05
Jsoup	93	1,261	2010-01-17	2019-07-04
JxPath	22	598	2001-08-23	2018-05-15
Lang	64	3,596	2002-07-19	2013-10-10
Math	106	4,913	2003-05-12	2013-10-16
Mockito	38	3,262	2007-11-15	2016-08-02
Time	26	1,718	2003-12-16	2013-12-04

The dataset does not include information about the commit that introduced the bug (if any), which we would need to verify that the result of step (3) of the *perfect test* method found the right change. However, the dataset provides a synthetic snapshot of the code (i.e., created by the authors of the dataset) which contains the same code as the fix snapshot, but without the changes of the fix (i.e., it only contains the regression test added along with the fix). In some of these synthetic snapshots it also includes additional changes such as, for example, the removal of flaky tests. The purpose of this synthetic snapshot is to provide a version where you can run the regression test without the fix and check its result. This supports our assumption that the regression tests identified in the Defects4J dataset can be used as *perfect tests*.

From the whole Defects4J collection of projects, we excluded project Chart because it uses Subversion [6] as version control system, as we focus only on projects that use Git. As a result, we included in our experiment 16 projects out of the 17 found in Defects4J, with a total of 809 bugs of the initial 835 bugs. Table 1 shows a brief description of the selected projects: the number of bugs it contains reported by the dataset, the number of stored commits, the dates for the first commit of the project and the last one (the last commit does not correspond to the last one in the official repository, since the projects in the dataset are stored as a copy of the git repository at a specific point in time). It should be noted that this dataset was extended in 2020, based on the original 2014 dataset (Just et al. 2014).

3.3 Transplanting the test to the past

Following the process shown in Fig. 1, we have designed and implemented a Python tool that automates all the necessary tasks to transplant the regression test to the commits corre-

[6] https://subversion.apache.org/

sponding to past changes, and run it to determine if it fails or succeeds. For each bug in our subset of the dataset, it takes the following steps:

1. **Extract information.** By using the command-line tool provided by Defects4J, extract the fix commit, bug report and regression test for the bug.
2. **Set up the repository.** By using the Defects4J command-line tool, obtain a copy of the git repository of the project corresponding to the bug.
3. **Execute the regression test on the fix commit.** This will ensure that the test actually succeeds for this snapshot, which means that it succeeds when the bug is fixed (if not, the test would not check properly, as the *perfect test* method states). For this, the tool checks out the snapshot corresponding to the bug fixing commit (BFC), and runs the regression test on it. In this step, the file containing the test is stored, in order to be transplanted into the previous snapshot (the test method, as well as the file name and the path where it is located is provided by the dataset).
4. **Execute regression test on all previous commits.** This will allow us to find the first failing change (FFC), which as we discussed, in the case of intrinsic bugs, will be the BIC that we are looking for. For this, the tool checks out each of those past commits, transplants the test to each of them, and runs it in each of them. The FFC will be the first one that fails after the last one that succeeds.

In the steps that execute regression tests, the tool follows this procedure: (1) *check out the corresponding commit*, (2) *transplant the regression test*, (3) *compile the source code*, (4) *compile the regression test*, and (5) *execute the regression test*. For compiling and executing we decided to use standard Maven and Ant commands, since the command provided by Defects4J only works with some specific commits (in the BFC and in a synthetic commit where the bug is present), and we needed to build and run the test on any of them (see previous discussion on finding the FFC at Section 3.1).

When the tool finishes all actions for a given bug, it reports the results of executing the regression test (if that was possible) for all commits prior to the BFC. These results include the success or failure of the different phases: building the source code, building the transplanted test code, running the regression test, and the result (fail or success) of running the test. In order to know the reasons for failures, the tool also keeps a log of the execution of each step.

In order to measure how far a test can be transplanted to the past and answering $\mathbf{RQ_{1A}}$, it is necessary to define a metric that allows us to measure how far we can transplant a regression test. We propose the *Transplantability* metric. For a given bug for which we have a test that detects it, we consider all commits that are ancestors of the BFC, in chronological order, and from this ordering we will find a commit n which is the oldest at which the test can be transplanted and executed. We define **Transplantability (in days)**, T_{days} as the number of days between commit n and the BFC. In the same way, we define **Transplantability (in commits)**, $T_{commmits}$ as the number of commits between commit n and the BFC. The two different ways of quantifying, together, are intended to give a more comprehensive view of how far we can transplant a test into past commits. Commits and days can be very different metrics for measuring distance of transplants. Both capture different information, depending on how the project behaves, since the number of commits per unit of time can be very different from project to project. The number of commits gives an idea of how many changes suffered the code base, while the number of days gives an idea of how much “real time” passed by. Since both changes to the source code and environmental changes due to the pass of time are relevant, we need both metrics. According to previous work (Maes-Bermejo et al. 2022; Tufano et al. 2017), as we go further in the past from the BFC, dependency errors are more likely to appear. Similarly, the more commits we go into the past, the higher chance that the code might have changed in ways that make compilation impossible (due to refactoring, for instance). The idea of using two different metrics to measure the same phenomenon has also been explored by Zerouali et al. (2018). In their research, to calculate the distance between two versions of a package (the one used by a project and the last one released), the authors propose two metrics: *time lag* (time elapsed between the date of the used version of a dependency and the date of the latest available version of this dependency) and *version lag* (how many major, minor or patch versions the release of a required dependency is behind).

When designing our experiment, we envisioned three different challenges to transplantability: source code compilability, transplanted regression test compilability, and transplanted regression test execution. Indeed, previous literature (Tufano et al. 2017; Maes-Bermejo et al. 2022) warns us that at least in the compilation of the source code we may face problems that will prevent us from continuing with the experiment. We also consider the likelihood that errors may also occur in the compilation of the regression test and in the execution of the test itself (i.e., that the test does not generate a report indicating whether the test passes or fails).

Problems related to compilability of source code, the compilability of the regression test code (the transplanted test), and the runnability of the regression test will be studied in order to answer **RQ$_{\mathbf{1B}}$**. To further understand these problems, we introduce three definitions: source code compilability, transplanted test compilability, and transplanted test runnability.

Source code compilability is defined as the percentage of snapshots in the past history of the BFC that could be successfully compiled. This metric was originally defined by Tufano et al. (2017). **Transplanted test compilability** is similar, but for the transplanted test: the percentage of snapshots in the past history of the BFC in which we could compile the transplanted test. These parameters let us know how often the reason for not being able of running the transplanted test was not being able to compile the test itself or compiling the source code in each of the snapshots.

Transplanted test runnability is defined as the percentage of snapshots in the past history of the BFC in which we could run (execute) the transplanted test without errors, producing a “success” or “fail” result. The larger the test runnability, the more commits in which we could run the test. If it is 100%, the *perfect test* method can be run for the whole history before the BFC. When test runnability is not 100%, for some commits we cannot assess if the test fails or succeeds (because it doesn’t run). Depending on the success or failure status in other commits, we might have to add the commit to the list of candidate BICs without being able to be more conclusive about it. Therefore, the transplanted test runnability shows how successful is the transplantation of regression tests to past commits.

3.4 Identifying the Bug-Introducing Change

From the results obtained after transplanting the regression test on the commits prior to the BFC we can seek to locate the BIC. At this point, we can revisit what “all previous commits” from Step 4 mean. In Fig. 1 we present the history of commits as linear. However, in a real git repository the commit history might not be linear: there are development models (as GitFlow [7]) with git in which development is done in parallel branches, which are merged when convenient. Therefore, the history of a project can be considered as a directed graph. The simplified model presented in Fig. 1 assumed a linear history of commits. We consider the history of the source code as a graph, which allows us to better determine the BIC. The consideration of the history as a graph to further refine the BIC search is a contribution of

[7] https://nvie.com/posts/a-successful-git-branching-model/

this work. Each node represents a commit that will point to the commits that precede it (its parents), which can be one or more.

With this in mind, we developed Algorithm 1 to detect the BIC in this graph. This algorithm receives as parameters the graph, annotated with the result of running the tests (its status, which can be success, fail or error), and the identification of the BFC in that graph, returning the list of candidates to be the BIC. This algorithm traverses the graph using a depth-first search (Cormen et al. 2001), starting with an empty list of candidates, and trying to find the list of candidates to be the BIC, according to the *perfect test* method. For that, the algorithm finds the node n fulfilling the condition that the test fails for it, but succeeds for all its preceding nodes, which would make n a candidate to be the BIC.

Since for some nodes we could not run the test (because it could not be compiled, it did not run, or the source code could not be compiled), we must consider that in these nodes the tests may or may not be executed successfully.

The algorithm operates under the following assumptions: (1) all the ancestors of the BIC are success commits, until the beginning of the history of the project, or until the features on which the test is built are introduced; (2) in the case of a commit with two or more parents, which generate branches, the BIC can only be in one of these branches. The first assumption is based on the definition of the BIC: the first commit where the bug manifests, which following the *perfect test* method, means the first commit where the test fails. In other words, in snapshots previous to the BIC the bug is not present, and therefore the test, if it can be run, should succeed. The test may not run if it is designed on top of some feature (e.g., some function) that does not exist for some snapshot, because it was not yet implemented. The second assumption is based on the unlikeness that the same bug (which is fixed by a single fix commit) is introduced in two different branches.

Our algorithm has as a precondition that the regression test can be run and gives a successful result in the BFC. If this precondition is fulfilled, we can run the algorithm, and come with two different outcomes:

- **The test succeeds in some preceding commit.** We should find at least one parent commit of the BFC for which the test fails (we should have at least one snapshot where the bug is present). Following the ancestors of those commit that fail, we eventually find one that succeeds. If we find a commit n where the test does not succeed, but it succeeds for all its parents, then n should be considered the commit that introduced the bug. In this case, the algorithm found a single candidate and returns it. If we find a commit n where the test cannot be compiled, and it succeeds in all its parents, then all commits between n and m, being m the first descendant of n where the test can be compiled and fails, are considered candidates. We found several candidates, and we cannot decide which one is exactly the BIC because we cannot run the test in them, but we know the BIC should be one of them.
- **The test does not succeed in any precedent commit.** In this case, we should assume that the bug was always in the code, since the functionality tested by the test was introduced or that we cannot run the test for some reason (i.e., we are not able to compile the source code). In any case, we cannot find the BIC: either it doesn't exist, or it is hidden because we cannot run the test, so the algorithm returns an empty list of candidates.

As an example of how the algorithm works, Fig. 2 represents the commit history for Bug 41 of the JacksonDatabind project from Defects4J dataset. The figure includes only commits relevant to understanding how the algorithm finds the BIC, and consecutive commits (without forks or merges) of the same color have been reduced to one. Each commit is identified with a number, whose value simply indicates its chronological position with respect to the fix

Algorithm 1 Algorithm to detect bug-introducing commits

```
Input        : A graph (commit history) and the bfc
Output       : A list of candidates for BIC
Precondition: The $bfc$ status (the output of the regression test) must be success
$candidates \leftarrow []$;
$queue \leftarrow [bfc]$;
$visited \leftarrow [bfc]$;
$temp\_candidates \leftarrow []$;
while $NotEmpty(queue)$ do
    $n \leftarrow GetFirst(queue)$;
    if $hasFailStatus(n)$ then
        $candidates \leftarrow []$;
    end
    $parents \leftarrow GetParents(graph, n)$;
    $allParentsSuccess \leftarrow True$ ;      /* Control if all parents are success */
    $allParentsError \leftarrow True$ ;        /* Control if all parents are errors */
    if $isEmpty(parents)$;                            /* Reach first commit */
    then
        if $size(queue) == 0$ then
            $break$;
        else
            $continue$;
        end
    end
    for $p \in parents$ do
        $allParentsSuccess \leftarrow allParentsSuccess$ and $hasSuccessStatus(p)$;
        if $\neg hasSuccessStatus(p)$ then
            if $size(queue) == 0$ then
                if $hasErrorStatus(p)$ then
                    $allParentsError \leftarrow False$;
                end
                if $p \notin visited$ then
                    $AddItem(queue, p)$;
                    $AddItem(visited, p)$;
                end
            else
                $allParentsError \leftarrow False$;
            end
        end
    end
    if $allParentsError$ then
        $AddItem(candidates, n)$;
    end
    if $allParentsSuccess$ and $\neg hasSuccessStatus(n)$ then
        if $hasFailStatus(n)$ then
            return $[n]$;
        else
            $AddItem(candidates, n)$;
            if $IsEmpty(queue)$ then
                return $candidates$;
            else
                $temp\_candidates \leftarrow candidates$;
            end
        end
    end
end
return $temp\_candidates$
```

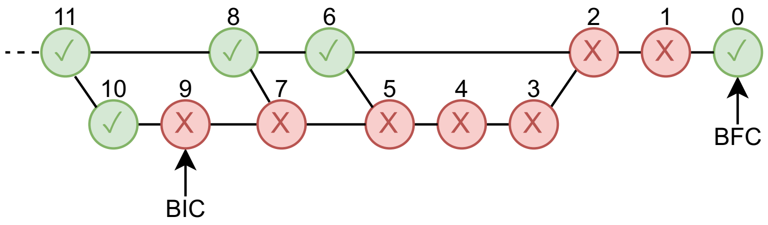


Fig. 2 Visual representation of the results of the experiment for Bug 41 of JacksonDatabind project

commit (BFC) in order to be able to refer to them. Colors in the figure show if the regression test succeeds (green ✓) or fails (red X).

In this figure we can see how Commit 2, although it has a green parent (Commit 6), cannot be considered as BIC, since it has another parent (Commit 3) where the bug is present. Following the ancestors chain, the algorithm will find at some point Commit 9, which is red, but for which all parents are green (in this case, only Commit 10). Therefore, the candidate list in this case will include only Commit 9.

3.5 Manual validation

To ensure that the results of our study can be considered as ground truth, we verified them by performing a manual validation of the BICs detected for each analyzed bug. For this purpose, one author performed the following steps for each BIC detected:

– Check and understand the bug report.
– Check and understand the fix in the BFC.
– Check and understand the changes to the code in the candidates to be the BIC.
– Check the output of the test run.

Following these steps, we categorized the BICs found in our study as true positives or false positives, using only true positives as the ground truth in order to generate a validated dataset of BICs.

4 Experimental Results

In this section we show the results of our study, answering the research questions presented in the introduction. The following results are intended to determine the extent to which we can operationalize the theoretical model proposed by Rodríguez-Pérez et al. We have considered a total of 809 bugs in the Defects4J dataset, after filtering out bugs for the project we do not consider as explained in Section 3.2. The results we found for each of those bugs are summarized in Fig. 3. This figure differentiates the cases in which, out of the total number of bugs, the regression test was found to pass again in some commit prior to the BFC from those that did not. In turn, from this first group, we differentiate the bugs from which we have been able to obtain a single candidate to be the BIC or several of them.

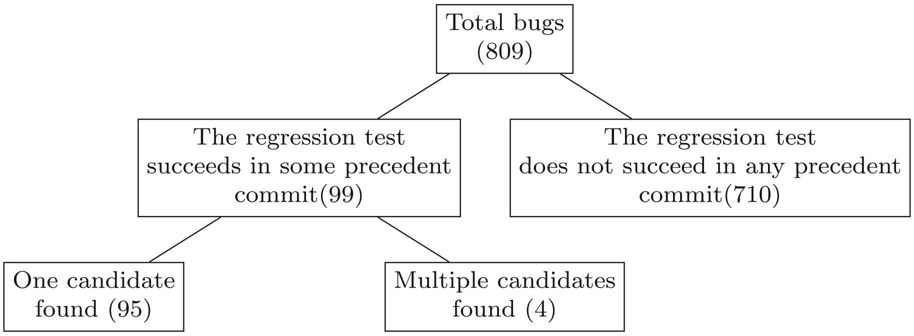

Total bugs (809)
The regression test succeeds in some precedent commit(99)
The regression test does not succeed in any precedent commit(710)
One candidate found (95)
Multiple candidates found (4)

Fig. 3 Summary of results for each of the bugs considered in the study

4.1 RQ$_{1A}$: "How far can a test be transplanted into the past?"

In Table 2 we show the values for each of the interpretations of Transplantability for each bug. The values are aggregated by project, showing the mean and median for all bugs in each project. Additionally, we add in the table the relative position (%) of commit n (1) with respect to the total number of days elapsed between the BFC and the first commit of the project and (2) with respect to the total number of commits between the BFC and the first commit of the project. A value close to 0% in this relative position indicates that we have barely been able to transplant the regression test, while values close to 100% indicate that the test has been transplanted in most of the past commits (with respect to the BFC).

Table 2 Transplantability (in days and in number of commits) for each bug, aggregated by mean ($\bar{x}$), by median ($\tilde{x}$) and by the relative position of the oldest commit where the test could be transplanted (%)

Project	# bugs	T_{days} $\bar{x}$	$\tilde{x}$	%	$T_{commmits}$ $\bar{x}$	$\tilde{x}$	%
Cli	39	913	1168	34.87	134	115	33.44
Closure	174	235	108	36.74	453	192	39.79
Codec	18	703	427	22.17	195	83	21.58
Collections	4	599	703	11.05	178	213	6.30
Compress	47	1,914	2,051	47.88	1,242	1,331	83.24
Csv	16	106	41	3.11	41	27	5.17
Gson	18	1,283	1,212	47.60	481	368	41.70
JacksonCore	26	444	450	32.98	262	258	34.00
JacksonDatabind	112	726	691	44.50	1,200	1,181	41.41
JacksonXml	6	890	939	40.78	263	239	41.58
Jsoup	93	437	240	26.95	142	76	18.33
JxPath	22	607	532	24.44	80	79	21.65
Lang	64	355	246	14.88	283	206	13.05
Math	106	197	120	8.14	295	194	10.75
Mockito	38	1,664	1,552	96.61	1,781	1,540	95.93
Time	26	502	483	15.44	109	94	6.68
All bugs	**809**	**591**	**313**	**32.58**	**536**	**216**	**33.54**

Table 3 Distribution of Transplantability results for all projects

	# bugs	mean	std	min	25%	50%	75%	max
T_{days}	809	591	708	0	87	313	832	3,475
$T_{commits}$	809	536	699	1	76	216	727	3,709

For a more comprehensive view of the Transplantability results, Table 3 provides in detail the distribution of T_{days} and $T_{commits}$ results. First, we found that both metrics offer very similar results, showing that the average frequency with which a commit is added to these repositories is approximately 1 day.

To check if the distances between the BIC and the BFC we found are consistent with other studies, we have used the datasets of Rosa et al. (2021) and Petrulio et al. (2022), computing $T_{commits}$ and T_{days} for the bugs on them.

Rosa et al. dataset contains 1,115 bugs from 887 different projects written in different programming languages (C, C++, Python, JavaScript, Java, PHP, Ruby ...) and it offers the BFC and BIC for each bug. We calculated the distance in days and in number of commits between the BIC and the BFC for 1,040 bugs (for the rest of the bugs, it was not possible to retrieve the code repository). The results can be seen in Table 4. It should be noted that there are projects with a huge number of commits that disturb the mean.

Petrulio et al. dataset contains 5,348 bugs from Mozilla project, written in C++ and JavaScript, and it also offers the BFC and BIC for each bug validated by the developers. In this dataset, the links can contain N BFCs and M BICs, being $N > 0$ and $M \geq 0$. We have discarded 45 links that did not have a BIC associated, and 1,157 links where there was more than one BFC (we consider that a bug can only be resolved in a single commit, given our definition of BFC: the commit where the bug is no longer present due to a change in the code). For the remaining 4,146 links we have calculated the distance in commits and days. For those cases where there was more than one BIC, we calculated the average distance between the BFC and each BIC. Finally, we discarded 90 links whose average distance in days was negative (i.e., the BIC was later than the BFC), thus exposing some limitations of this dataset. This results in a total of 4,056 links, whose distribution can be seen in Table 5.

Comparing Tables 4 with Table 3, we can see that, for example for the 75% percentile, our transplantability distances are much larger (by about an order of magnitude) than the distance between the BIC and the BFC in Rosa et al. That means that, if our projects behave similar to those in the Rosa et al. dataset, at least for 75% of bugs we are very likely having the capacity of transplanting the test well beyond the BIC.

Comparing Tables 5 with Table 3, if we consider the transplantability metric in days and the distance in days, the results are similar to the comparison with the Rosa et al. dataset (i.e., one order of magnitude higher). It should be noted that the Mozilla project has many modules that are developed at the same time, so the distance in number of commits is significantly greater and is not directly comparable to other BIC datasets such as ours or that of Rosa et al. in this metric.

Table 4 Distance between BFC and BIC in days and commits for the dataset of Rosa et al. (2021)

	count	mean	std	min	25%	50%	75%	max
In # of days	1,040	106	367	0	1	9	51	6,778
In # commits	1,040	1,678	13,129	1	3	11	81	262,977

Table 5 Distance between BFC and BIC in days and commits for the dataset of Petrulio et al. (2022)

	count	mean	std	min	25%	50%	75%	max
In # of days	4,056	100	319	0	5	16.5	70	9,1954
In # commits	4,056	14,555	40,382	4	697	2,442	10,450	514,394

RQ_{1A}: "How far can a test be transplanted into the past?"
For the dataset used, we have managed to transplant the regression test for a bug up to 591 days (536 commits) in the past on average. For 50% of the bugs, the regression test could be transplanted up to at least 313 days (216 commits). On average, regression tests can be transplanted to a 32.58% of the days (33.54% of the commits) between the BFC and the initial commit of the project.

4.2 EQ_{1B} "How compilability and runnability problems impact the transplantation of the regression tests to the past?"

In Table 6 we show average and mean data for the three metrics defined in Section 3: source compilability, transplanted test compilability and transplanted test runnability.

Table 7 provides in detail the distribution these metrics for a more comprehensive view of the results.

Table 6 Source code compilability, transplanted test compilability, and transplanted test runnability for each bug, aggregated by mean ($\bar{x}$) and by median ($\tilde{x}$) per project

Project	# bugs	Source Compilability		T.Test Compilability		T.Test Runnability	
		$\bar{x}$	$\tilde{x}$	$\bar{x}$	$\tilde{x}$	$\bar{x}$	$\tilde{x}$
Cli	39	55.66	62.61	28.19	30.88	28.19	30.88
Closure	174	59.40	54.36	34.47	17.82	34.47	17.82
Codec	18	23.74	14.29	20.44	8.84	20.44	8.84
Collections	4	97.94	98.98	5.73	7.46	5.61	7.37
Compress	47	32.29	23.60	22.90	17.02	21.07	10.92
Csv	16	19.07	17.38	5.26	3.51	5.21	3.51
Gson	18	42.00	35.98	40.73	34.68	40.73	34.68
JacksonCore	26	35.08	32.54	31.52	30.12	31.52	30.12
JacksonDatabind	112	85.18	85.59	21.97	16.02	21.97	16.02
JacksonXml	6	89.26	88.04	28.83	24.59	28.83	24.59
Jsoup	93	21.08	12.52	17.56	9.74	17.56	9.74
JxPath	22	92.25	100.00	21.93	23.57	21.93	23.57
Lang	64	74.08	66.08	11.61	8.50	11.61	8.50
Math	106	40.00	36.19	8.61	6.28	8.61	6.28
Mockito	38	30.69	30.06	23.83	25.22	23.83	25.22
Time	26	69.58	100.00	6.64	5.80	6.35	5.80
All	**809**	**52.95**	**49.83**	**21.86**	**12.75**	**21.74**	**12.47**

Table 7 Distribution of source compilability, transplanted test compilability and transplanted test runnability results for all projects

	# bugs	mean	std	min	25%	50%	75%	max
Src compilability	809.0	52.95	31.87	1.58	24.88	49.83	82.5	100.00
T.Test compilability	809.0	21.86	24.37	0.10	4.36	12.75	30.5	99.66
T.Test runnability	809.0	21.74	24.39	0.10	4.36	12.47	30.4	99.66

The reasons that prevent transplanting the regression test are directly related to these metrics and can be classified as follows:

- **Compilability of the source code**. If the source code cannot be built for the snapshot, there is no way to build the regression test. Compilability of past snapshots has been studied in detail by Tufano et al. (2017), whose experiment in 2014 showed an average compilability of 100 Java projects of 37.74%. Maes-Bermejo et al. (2022) replicated this experiment in 2020 and a decrease in compilability was observed due to the lapse of time, obtaining a value of 25.09%. The compilability may vary largely from project to project, and it will depend among other factors on the availability of third party modules needed to build the code, on the availability of the building tools in the right version, and on the complete automation of the building process. These factors are usually affected by time, and a degradation of compilability as time passes has been observed in these two previous studies. In our case, the mean compilability of the snapshots previous of each bug is 52.95% (with a median of 49.83%). The value obtained is higher than that obtained in previous studies due to a combination of good practices by the authors of the Defects4J dataset; storing project dependencies and adapting their configuration files to ensure high reproducibility in the experiments (although only in the BFCs), together with additional adaptations made by the authors of this work, completing the adaptation of the configuration of each project to each commit of its history.
- **Compilability of the regression test**. If the test cannot be built, it cannot be run. The test is built on top of the snapshot, and it may not build because the code it expects in the snapshot is not present. This may happen because that code was still not implemented for a given snapshot. For example, this is the case if the test tests a certain function: in some past snapshot the function may not be implemented yet. This may also happen because the code was in the snapshot, but not in the way the test expects it. For example, this may happen in case of refactoring between the snapshot we are trying to build and the snapshot for which the test was designed. In general, these problems will become more and more frequent for older commits with respect to the BFC, since more artifacts (code files, libraries or configuration files) could change since those snapshots to the one corresponding to the BFC, for which the regression test was designed. They will also be frequent in branches which for some reason lack some code needed by the test. The aggregate results for the transplanted test compilability (21.86% mean and 12.75% median for all bugs) are strictly lower than source code compilability, since compiling the source code is a necessary step to be able to compile and run the tests. To the best of our knowledge, there is no large-scale study on the compilability of a test that is transplanted to past commits, so we do not have a baseline on which to compare the results obtained on this metric.
- **Runnability of the regression test**. Even if the regression test can be built, maybe its run does not produce a result, but fails earlier due to some code not behaving as expected.

When aggregating all bugs together, the mean for transplanted test runnability is 21.74%, and the median is 12.47%. This means that in half of the bugs the test could be transplanted successfully to about 12.5% of the commits (although the difference with the mean shows how in some bugs, the transplant was successful in a much higher fraction of the cases). These values are again slightly lower than the previous metric (the compilability of the regression test) since we need to compile the test code in order to be able to run it. In this case, the values are very close (or even equal) to those of the compilability of the regression tests, so we can affirm that if the transplanted test is successfully compiled, it can be executed. Again, to the best of our knowledge, the runnability of a test transplanted to the past has not been addressed on a large scale.

We will discuss the problems of transplanting the tests in the past in more detail in Section 5.4.

When examining the numbers for each project, we can see how, even when it varies from project to project, transplanted test runnability is always very similar to transplanted test compilability, which means that if the test compiles, usually it runs. In other words: the main blocker for running a test is that the test does not compile. For compiling the test, we should compile the source code. Again, by looking at this table, we can see how in some cases, the blocker for compiling the transplanted test is that the snapshot does not compile (for example for Gson, in almost all commits for which the snapshot compiled, the test also compiled). But for most of them, even if the snapshot compiles, the test does not.

These results are relevant because they show that by improving the compilability of the source code, and of the regression test, we could improve transplanted test runnability, and therefore, the applicability of the *perfect test* method. They also show that some projects have a very low compilability. This may be due to age (See Table 1) of those projects (maybe the developers of those projects were using old practices that do not interact well with modern tools), or to specific characteristics of those projects, that maybe could be fixed with more knowledge about their building configuration.

Despite all these factors, the fact that for some projects transplanted test runnability is high shows that at least in some cases the conditions to use the *perfect test* method with regression tests hold.

RQ$_{1B}$: "How compilability and runnability problems impact the transplantation of the regression tests to the past?" The effectiveness of the transplant is limited by compilability issues unrelated to the transplanted test (the snapshot does not compile 47.05% of the time) or by compilability of the transplanted test (the test does not compile 78.14% of the time, essentially because of the limitation in compiling the source code or because it relies on missing code). However, we found that when the test compiles, in general, the test can be run. Therefore, compilability of the source code is a blocker that, when improved, could improve the transplanted test runnability.

4.3 RQ$_2$:"Can the BIC for a given bug be found using its regression test?"

Following on with the chart in Fig. 3, to answer RQ_2 we will study in which cases the test succeeded in some precedent commit, and in those cases, how many candidate BICs were found. We consider the following scenarios as defined in our methodology.

The test does not succeed in any previous commit.

From the total 809 bugs, in 710 the regression test did not succeed in any commit previous to it. In these cases, the *perfect test* method (using regression test as *perfect tests*) does not allow us to identify candidates for being the BIC. Since the test never succeeds in commits previous to the BFC, we cannot find the commit in which the bug was introduced. In the snapshots corresponding to some of those commits this is because the test cannot be run (and therefore we don't know if the bug is present or not in them), and in some others because it can be run, but fails (and then we know that the bug is present).

The reasons for the test not running (the test was not successfully transplanted) were already explored in the previous section, and in many cases are related to compilability problems. The fact that the tests are executed (successfully transplanted) but fail in all commits prior to the BFC is a case contemplated and addressed by the *perfect test* method: it means that the bug is present in those snapshots since the feature tested was introduced. However, if we cannot find a previous commit for which the test succeeds, we have no evidence of where the BIC is.

The test succeeds again in some past commit.

For the remaining 99 bugs, the test succeeds in at least one commit previous to the BFC, and therefore we can provide more conclusive results by running the *perfect test* method.

For 95 bugs out of these 99, our algorithm produces a single candidate to be the BIC. The regression test succeeds in snapshots previous to this BIC candidate, which means it is the FFC (First-Failing Change), and following the *perfect test* method, the commit that introduce the bug (BIC). In 8 of these cases, the BIC was found in a direct parent of the BFC, which means the bug was detected and fixed very quickly after it was introduced. This is due to a practice known as *Backtracking* (Yoon and Myers 2012, 2014), where the developer reverts part or all of the change made when an issue is reported.

For the 4 remaining bugs, the algorithm finds multiple candidates. If n is the first commit (going backwards from the BFC) in which the test succeeds, we have several candidates if for one or more commits that are right before n (again, going backwards from the BFC) the test could not be executed (we are not able to compile the code or the test). In those cases, since the test could not be run, we do not know if the bug is present or not, and, according to the definition provided by Rodríguez-Pérez et al., any of them could be the FFC (the first failing change). So, all of them, including the last one that failed (again, in the same order), are possible BICs.

RQ$_2$: "Can the BIC for a given bug be found using its regression test?" Yes, at least for those cases where a functionality no longer behaves as the regression test expects it. The regression test can be transplanted to past commits, and using the *perfect test* method with the regression test as *perfect test*, the BIC can be found. In our case, for the 809 bugs for which we could assess the regression test detected the bug, we could use the method to identify precisely the BIC in 95 bugs, and to provide a list of candidates for other 4 bugs. The bugs for which the method did not work were mainly due to not being able of running the test, because of compilability or runnability problems, in addition to the contemplated case in which the tested feature has always contained the bug. In general, when regression tests could be run, the method worked.

4.4 Validation of results

Once we have detected the BIC for 95 bugs through the *perfect test* method (as well as detecting BIC candidates for 4 bugs), we want to ensure that the BIC found is actually the BIC, the commit that introduced that bug. Following the steps already described in Section 3, we checked manually all these cases, finding that all of them are true positives.

In addition, we also evaluated manually the 4 remaining bugs for which more than one BIC candidate. For three out of the four bugs (JacksonDatabind 14, Math 2 and Gson 7), there are only two candidates, while in the remaining one (Closure 20) there are 17 candidates. In the JacksonDatabind 14 bug, the first candidate (chronologically) only adds a test that does not compile, while the second one adds new functionality so this second one is the candidate to be the BIC. In the Math 2 bug, the first candidate (chronologically), adds a new feature, but performing an import incorrectly and preventing it from compiling. The second candidate fixes this import and does not add more code, so the first one is the candidate to be the BIC. In the Gson 7 bug, the first candidate (chronologically), completely deletes the JsonReader class (preventing the code from compiling). In the second candidate, the JsonReader2 class is renamed to JsonReader, so the second candidate is the candidate BIC. Performing the same manual validation as for the cases with only one candidate identified, we have determined that the only candidates identified (after discarding the others), are the BIC, adding 3 BICs to the total detected. Finally, for the Closure 20 bug, we found 17 candidates. In the first candidate (chronologically) a completely different internal module (still under development) is started to be used. The development takes 16 commits, during which the project cannot compile, as classes required will be added in later commits. After finalizing the functionality, the regression test starts to fail until the BFC. In this case, further analysis would be necessary, with more domain knowledge, to find the BIC among the candidates.

Two conclusions can be drawn from the results of this manual validation:

- The *perfect test* method can be used to identify the bug-introducing change, with regression tests as *perfect tests*, at least for those cases where a functionality no longer behaves as the regression test expects it.
- We have a ground truth dataset of 98 bugs for which we know the BFC and the BIC, which can be used to evaluate methods for finding the BIC.

This manual analysis also allowed us to explore in some detail how those commits introduced the bug. We found the following two cases:

(1) **The bug is introduced in a commit marked as FIX**. This means that the commit was trying to fix another bug, but while doing that, it introduced a new one. For example, Bug 24 of the JacksonDatabind project has a BIC which is the BFC of another bug. The bug fixed in that BIC deals with date serialization, changing several classes for that. One of those commits, which is reverted in the BFC for Bug 24, was the one introducing that new bug. Previous studies (Guo et al. 2010; Purushothaman and Perry 2005; Yin et al. 2011) also state that bug-fixing commits are more likely to introduce a new bug in the software.

(2) **The bug is introduced as a refactoring or reimplementation**. This case will be illustrated with Bug 23 of the Jsoup project (an HTML parser). The bug is that the special entities that include numbers to display fractions in HTML do not recognize the numbers, so they do not work as expected (i.e., the string "&frac12" should generate "$\frac{1}{2}$"). In the commit marked as BIC by our tool, there is a refactoring of the parse functionality, the main method of the library. This function goes from internally using the Parser

class to using TreeBuilder (a new class). When using the Parser class, the bug did not manifest itself and the regression test passes. This refactoring brings with it the creation of new classes (that TreeBuilder needs to operate), among which we will highlight Tokeniser and CharacterReader. Tokeniser uses CharacterReader to read characters from the stream, one at a time, using the method consumeLetterSequence. In the BFC the method consumeLetterThenDigitSequence, considering both letters and numbers for an entity, is added to the CharacterReader class. Then the Tokeniser class is changed to call these new method instead of the old one (consumeLetterSequence), that only considered letters, hence solving the bug by considering numbers as well as part of an entity.

This manual validation shows one of the most important results of our study: a collection of bugs with an identification of the BIC that introduced them can be produced automatically from a collection of BFCs using the *perfect test* method, with regression tests as *perfect tests*. These collections could be used to produce ground truth datasets for evaluation analysis, as we do in this study, but also for any other study where a collection of BICs is needed, such as those on how bugs are introduced. In this paper we provide BIC-RT as one of these collections obtained from the Defects4J bug dataset, with a total of 98 identified BICs.

5 Discussion

In this section we will discuss in detail the implications for methods for finding bug-introducing changes (Section 5.1) and the contribution of the generated dataset through (1) an evaluation of SZZ-based tools on it (Section 5.2) and (2) a comparison of our BIC dataset (BIC-RT) with a previous BICs dataset using Defects4J as bug dataset (Section 5.3). We will also discuss the limitations of transplanting a test into the past (Section 5.4) and the implications of our work for practitioners (Section 5.5) and researchers (Section 5.6). Finally, we discuss the threats to validity of this paper (Section 5.7).

5.1 Implications for methods for finding bug-introducing changes

The work by Rodríguez-Pérez et al. (2020) already showed that BICs could be defined and identified in terms of the "perfect test" method. We have made this approach practical, by using regression tests as proxies for perfect tests. In our study, we have shown how a sensible proportion of regression tests can be transplanted to past commits, and used to identify when the bug fixed in a BFC was introduced. It is important to notice that, if we agree on the definition of a BIC based on a perfect test, and regression tests can work as a proxy for that perfect test, this way of identifying the BIC is "perfect", in the sense that it should really be the BIC.

Our study showed that the process can be automated (at least to some extent), and that it works in a sensible fraction of BFCs, in a real project, with no specific considerations for designing regression tests that can be used as perfect tests (for example, not specifically designed to work in the past), and with little adaption to the history of the project (for example, by building more complex tools that can adapt the tests to past states of the system, so they compile and run in more cases).

The study also showed that, when BICs can be identified this way, we have a very interesting baseline to measure effectiveness of BIC-finding algorithms, such as SZZ and derivatives (see Section 5.2 for details). Using this baseline, we have found BICs that are not found by SZZ and their derivatives. It is interesting to notice that the reason why they are not finding them is because of their assumptions on how to find the BIC. Since they rely on relationships between commits in the VCS graph, they cannot found those BICs that are not related to the BFC that way, or they cannot decide which one is the real BIC when there are several candidates with similar graph relationships.

Therefore, the regression test approach we propose is, more than a practical approach to find BICs (which for now we didn't prove in the general case, and we know could have practical problems in the general case), a practical construct that, for the cases where it can be applied (regression tests are available, transplantable to past commits, and runnable for them), identifies the BIC with great certainty. The only limitation is to which extent the regression test actually triggers the bug. However, these cases should be few, because regression tests are precisely designed to trigger the bug.

Therefore, as we explain later in this section, it is a valuable tool for determining when a BIC-finding algorithm works, and to which extent it can find the BIC for specific BFCs. Since it can be automated at least in some cases, those algorithms could be run on those specific BFCs to check for their accuracy. In fact, the dataset we provide as a result of our study can be a resource for doing exactly that, helping to assess advances in finding commits that introduced bugs.

5.2 Evaluation of SZZ derivatives

Thanks to our subset of the Defects4J dataset with 98 bugs with verified BICs, we can evaluate the performance of the implementations of SZZ derivatives. We will use this dataset as the ground truth, and will run the SZZ derivatives implementations to check to which extent they correctly identify the right BIC for each bug. We will analyze in detail those bugs where the implementations of SZZ derivatives are not able to find the BIC.

Several implementations of the SZZ algorithm and derivatives of it have been presented in the literature. However, for many of them their implementation has not been published, which has made it difficult to reproduce their results (Rodríguez-Pérez et al. 2018). Fortunately, several recent studies (Borg et al. 2019; Lenarduzzi et al. 2020; Pokropiński et al. 2022; Rosa et al. 2021) provide public implementations of their algorithms. We evaluate their performance in finding the BIC, considering the manually validated results of our study as the ground truth.

At the moment of writing there are seven publicly available implementations of SZZ variants:

– OpenSZZ (Lenarduzzi et al. 2020). It is based on the original version of the SZZ (Śliwerski et al. 2005).
– PySZZ (Rosa et al. 2021). Includes five implementations of SZZ-derived algorithms: ag, l, r, ma and ra. SZZ-ag was proposed by Kim et al. (2006) and is based on the original SZZ algorithm (Śliwerski et al. 2005), solving some limitations related to cosmetic changes in the code, such as moving a bracket to another line. SZZ-l and SZZ-r were proposed by Davies et al. (2014) and is based on SZZ-ag, using two different criteria to select the BIC among the candidates: SZZ-l uses the largest candidate (the commit with the highest number of changes), while SZZ-r uses the most recent candidate. SZZ-ma was proposed by Da Costa et al. (2016) and is based on SZZ-ag, excluding from the BIC candidates all commits that do not include changes to the source code, including merges between branches. SZZ-ra was proposed by Neto et al. (2018) and is based on SZZ-ma, excluding from the BIC candidates those commits that include refactoring operations.

Table 8 Results of SZZ algorithms on our BIC dataset

SZZ Implementation	Correct BICs	Hit rate	Candidates (avg)
OPENSZZ	17	17.35	1.05
SZZ UNLEASHED	6	6.12	17.37
PYSZZ-ag	39	39.80	1.21
PYSZZ-l	15	15.31	0.68
PYSZZ-r	22	22.45	0.68
PYSZZ-ma	52	53.06	2.45
PYSZZ-ra	39	39.80	1.44

– SZZ Unleashed (Borg et al. 2019). This variant partially implements an algorithm proposed by Williams and Spacco (2008) based on SZZ-ag, improving it by using a line-number mapping approach (Williams and Spacco 2008) and DiffJ [8] (a java syntax-aware diff tool). We emphasize that it only partially implements it since it does not use DiffJ.

For our work, we have selected these seven implementations of the SZZ to examine their results on the BICs detected by our tool. These implementations will identify, for each bug, a list of commits that are candidates to be the BIC, starting from the BFC for that bug. For evaluating each implementation, we have computed the number of commits that they included in the list of candidates for each bug, and the number of bugs for which the correct BIC is in the list of candidates. We have then aggregated the numbers for each implementation, computing the total number of bugs for which it correctly included the BIC within the list of candidates, its percentage over the total number of bugs (98), or *hit rate*, and the average number of BIC candidates per bug (including those bugs for which the implementation produced zero candidates). These results are shown in Table 8.

The results show a great variability in the ability of the implementations we have evaluated to identify the correct BIC, for the bugs in BIC-RT. It is remarkable that the hit rate is relatively low, except for the most advanced implementations of PySZZ (ag, ma and ra), which obtain an acceptable hit rate despite not having the information on whether the bug is present or not that the *perfect test* method provides.

There is also great variability in the number of BIC candidates produced per bug, from 0.68 to 17.37. However, for most implementations (all of them but one) the number of candidates per bug is relatively low (less than 2, in average). This means that they are reasonably precise, given that they only use limited information.

In addition, for 40 bugs none of the SZZ implementations included the right BIC in the list of candidates. These 40 bugs have in common that the BFC does not fix the same lines that were introduced in the BIC, so the main premise of the SZZ fails, with the result that it cannot find the BIC. This evidences a known limitation of SZZ-based approaches. Our tool contributes to the state of the art to correctly identify BICs for this kind of bugs.

With this evaluation we have also shown that BIC-RT can provide ground truth for evaluating SZZ derivatives, and other algorithms for automatically finding the BIC that introduced a certain bug. Since BIC-RT was produced following an automatic procedure (we validated it manually, but the BICs were first identified in a completely automated way), we expect that

[8] https://github.com/jpace/diffj

larger ground truths can be obtained in the future to better evaluate any proposed algorithm to solve the problem of finding the BIC.

5.3 Comparing BIC-RT with InduceBenchmark

An important result of our study is a dataset of BICs (BIC-RT), automatically found from their BFC, and manually validated, extracted from the Defects4J dataset. Based also on Defects4J, InduceBenchmark (Wen et al. 2019), a dataset with 91 BICs, was created to evaluate SZZ implementations. We compare the results of our technique on 82 of the bugs in InduceBenchmark (the remaining 9 bugs correspond to the project we filtered out in Section 3.2). Our technique found automatically the BIC for 30 of those 82 bugs. In 25 of these cases, we found exactly the same BIC identified in InduceBenchmark. For the 5 bugs where we do not get the same BICs as in InduceBenchmark (all of them belonging to the Closure project), we have analyzed the results of both datasets. For bugs 90 and 114, we found that in the commits reported by InduceBenchmark as BICs, the regression test provided by Defects4J that reveals the bug gives a success result. In fact, in these commits reported as BICs there is no real change in the application code, just changes in the comments (90) and file permissions (114). For bugs 12, 82 and 131, the test fails on the commit marked as BIC. Reviewing these commits we find that there are no real changes in functionality that could cause the bugs to be introduced. These commits only change author names (12), modify toString() methods (82) and delete whitespace (131).

Summarizing this discussion, BIC-RT adds 68 new BICs to those offered by InduceBenchmark, our tool offers a method to obtain new project BICs automatically and can also be used as an automatic method to validate BIC datasets. In addition, we detected 5 BICs that were misidentified by InduceBenchmark.

5.4 Transplanting tests to the past

One of the main reasons for proposing the operationalization of the *perfect test* method using regression tests is that these (regression tests) are present in many modern projects. This means that the technique could be used in many of them, if those tests can be transplanted successfully to past commits. However, in Section 4.2 we showed how in many cases this was not possible, and how much success we had in the different phases of the process (compilation of the snapshot, compilation of the test, and execution of the test). These problems clearly limit the effectiveness of the technique.

5.4.1 Improving the transplantability of the regression tests

In fact, studies on the compilability of past commits (Tufano et al. 2017; Maes-Bermejo et al. 2022) show that it is relatively usual that a considerable fraction of past commits in a project are not automatically compilable as such. However, those studies also show that some reasons for those problems (such as the availability of dependencies or the suitability of build configurations) can be mitigated. To mitigate these problems, our tool allows to include a script in which the user can define fixes to be applied to each commit in which the regression test is executed. The authors of the Defects4J dataset follow a similar approach (from which we draw our inspiration). They provide manually generated configuration files so that the tests can be run on the BFC and on a synthetic version of the BFC without the fix code (with the aim of providing a commit where the bug is revealed). These configuration files resolve some dependency issues by providing these dependencies as part of the Defects4J framework. For 502 bugs out of 809, we have taken advantage of these configuration files, making some modifications to them, in order to transplant them together with the regression test and ensure high compilability of past commits. The resolution of dependencies from external repositories is one of the main causes of failure in the build of Java projects (Tufano et al. 2017; Maes-Bermejo et al. 2022). One of the main advantages offered by these configuration files is that the project dependencies are obtained as local files (which are part of Defects4J) instead of downloading them from a remote repository, solving the above-mentioned problem.

In the following we discuss other relevant modifications and fixes included, based on the suggestions proposed in the literature (Maes-Bermejo et al. 2022). We found references to dependencies (not included in Defects4J) that include the suffix “-SNAPSHOT”, which indicates that this is a volatile development version and is sometimes removed from the dependency repositories (causing the impossibility to compile a project that depends on them). For 15 bugs we found that this dependency was included in the BFC, and the compilation of the BFC failed due to it, thus preventing our method from being able to work (the compilation and test execution at the BFC, with a success result, is a precondition for the method). The removal of this suffix, forcing it to use the stable version of that dependency, has allowed us to compile the BFC of these 15 bugs, allowing our method to start finding the BIC for them. This fix on dependency issues has been used on a total of 44 bugs (including the 15 mentioned above). In 144 bugs from 6 projects we faced problems with source code parsing (due to the inclusion of unrecognized characters in strings or comments). Two different types of fixes have been used to solve this problem: (1) modify the encoding in the configuration file that is transplanted to the past along with the regression test or (2) modify the snapshot configuration file to include the new encoding. We have also had to consider, in one project (Joda Time), that the code directories may change location (be placed in subfolders) in older commits, so it has been necessary to automate their re-structuring so that it can be compiled.

Building transplanted tests was also a problem. First, the standard way of building tests in Java requires building all of them together. This means that if there is a problem compiling just a single test, the test compilation fails, and therefore we cannot run the transplanted test (even if it compiled successfully). This effect could be mitigated by ensuring that only the transplanted test is compiled, with the risk, maybe, of having dependencies on some test classes that are not run (e.g., inheritance of a parent class). Fortunately, we also observed that once the test was compiled, it almost always runs successfully. In any case, for 17 bugs, we have automatically removed in the past commits some problematic tests (that did not compile) and were not related to the regression test. For 25 bugs, it has been necessary not only to take the regression test to past commits, but also to take a file on which the test depends (auxiliary classes created specifically for that test or modified parent classes in the BFC that include code required by the regression test).

5.4.2 Limitations on regression test transplantability

Despite all our efforts to transplant as many regression tests as possible, we still found severe limitations in transplanting them. For the 710 bugs for which their corresponding test did not succeed in any previous commit snapshot, only in 55 cases we did not encounter any limitations to the transplant: the test could be transplanted, including execution, up to the initial project commit. This is because the first commit of the project includes code developed in another repository. In the remaining 655 cases, we detected, by checking the execution logs, several errors that limit transplantability. In Table 9 we show the errors that limit the transplantability found along with occurrence, divided in two categories: when the source

Table 9 Errors that limit of the transplantability of regression tests. Capitalized errors are a group of very similar errors, while non-capitalized errors correspond to the exact message returned in the log

Category	Error	Count
Source code build error	No pom.xml file	35
	File or directory not exist	25
	Java version error	19
	Other source build error	3
	Subtotal	**82**
Test code not compatible with source code	cannot find symbol	352
	cannot be applied to given types	51
	package X does not exist	40
	Java version error	33
	duplicate class	27
	no suitable method found	21
	incompatible types	11
	no suitable constructor found	10
	has private access	5
	unreported exception	5
	class X is not abstract and does not override abstract method Y	4
	non-static method cannot be referenced from a static context	3
	method X in class Y cannot be applied to given types	2
	method does not override or implement a method from a supertype	2
	type StringEncoderAbstractTest does not take parameters	2
	clone() has protected access in Object	1
	annotation type not applicable to this kind of declaration	1
	class or method has private access	1
	reference to X is ambiguous	1
	try-with-resources not applicable to variable type	1
	'void' type not allowed here	1
	Subtotal	**573**
	Total	**655**

code is not compilable and when the test code is not compatible with the source code. This categorization is based on whether the error limiting the transplantability of the regression test is in the source code building phase or in the regression test building phase.

In the *Source code build error* category, we found that the number of errors (82) barely represents 12% of the total number of errors. This is mainly due to our efforts in fixing the most common bugs in the build, as described in Section 5.4.1. The errors are mainly due to a change in the build system in older commits (affecting only a single project), due a directory restructuring (preventing the configuration file from finding the right directories or files) and our use of Java 8, while the code was expecting an earlier version of Java (very old versions of the code use “enum” as a variable name: from version 5 onwards it is a reserved word of the language).

In the *Test code not compatible with source code* category, we find a wide variety of errors. The main error (352, 53% of the total), *cannot find symbol*, is due to the fact that some method or class used by the test does not exist as of a certain commit in the source code (the one prior to its implementation). The error *package X does not exist* (40) is similar, but applied to a package that no longer exists. The errors *cannot be applied to given types* (51), *no suitable method found* (21) and *no suitable constructor found* (10) are due to method or constructor headers that have changed and are no longer compatible. The *incompatible types* error is due to the fact that in the test code, a method does not return an object of the expected class. As in the source code problems, we found errors due to the Java version used (Java 8), for example, that the use of generics in the regression test is not supported in versions of the code prior to Java 5. The remaining errors involved less frequent cases but also related to the compatibility of the regression test with the rest of the source code of the project in previous versions.

These limitations of transplanting regression tests to past snapshots can be mitigated by rewriting them, focusing on the functionality tested, and having into account that they may be transplanted to past snapshots. We will discuss later how developers can improve transplantability when writing these regression tests. However, in many cases these tests are limited by the natural development of the project, and it will not be possible to make changes to ensure compatibility with all previous versions.

5.5 Implications for practitioners

In open-source projects, such as those of the Apache Foundation (Iida and Matsumoto 2016), is a common practice to include a test that reveals the bug in the bug report (available before the bug is fixed) and then include it in the bug-fixing change. In some others, practitioners start by building that test before trying to fix the bug. In both cases, these regression tests are available before starting to fix the bug. The operationalization of the *perfect test* method with these tests allows to automatically find the BIC for a bug, assuming that the project took care of facilitating transplanting tests to the past (something that they can do by maintaining some rules on how to compile the source code as the project evolves). Despite the fact that the *perfect test* method defines the BFC as a starting condition, in fact only the regression test is required. Therefore, when starting the process to fix a bug, a developer would have a hint about how the bug was introduced, which may be invaluable for speeding up the fixing process. This is an advantage over methods based on the SZZ algorithm (Śliwerski et al. 2005) that require the BFC to operate.

Although it is a recommended practice to add a test that detects the bug in a fix change, some studies (Levin and Yehudai 2017) reveal that developers usually fix bugs without performing complementary test maintenance in the same commit. Nevertheless, the study conducted by Pinto et al. (2012) on a collection of open source software projects shows that 14% of the tests added to the code were included in a fix change.

In addition, to have more evidence about the use of regression testing, we have studied the frequency of regression tests in a dataset other than Defect4J. Again, we will use the dataset of Rosa et al. (2021), which contains 1,115 bugs from 887 different projects written in different languages where each bug has its BFC identified. Through an automatic search, we have detected that 108 bugs have modified test code in their BFC. We have manually evaluated these tests, obtaining that in 55 cases the test added (or modified) is a regression test. In the remaining cases, the change in the test did not detect the fixed bug (50) or the original repository was not available (3). Therefore, for this dataset, we find that only 5% of the BFCs have test code added to prevent future regressions. This is a low number for a practice that is usually recommended in modern software development, which came as a surprise to us. From this, we can only conclude that more studies are needed to know to which extent the practice of writing regression tests is common in software projects nowadays. In any case, our method would work only in projects in which regression tests are common when fixing bugs.

Assuming that regression tests in a certain project are common, a practitioner aiming to use our method would have to deal with two challenges: i) compiling the source code into previous snapshots of the code and ii) compiling the regression test code into previous snapshots of the code. The first challenge has already been addressed in previous studies (Tufano et al. 2017; Maes-Bermejo et al. 2022) and involves dealing mainly with the resolution of dependencies (always trying to use dependencies in stable versions). The second problem is more complex. The compilability of the regression test is limited by how the project is developed: the further back we transplant the test, the more likely it is that changes in the code will be reverted that prevent the test from being compiled (for example, a function called in the test has been refactored over time). To mitigate this problem, practitioners are encouraged to make their regression tests as black box as possible. Tests should also be, as far as possible, self-contained: do not rely on utilities of other classes or use inheritance. This would not only make it easier to transplant to old snapshots of the project, but also make it more maintainable as a test that detects that the bug is not reintroduced in the future.

5.6 Implications for researchers

We have found an automatic method for producing a reliable collection of BICs, given their BFCs and their regression tests. This may be quite important for producing much larger datasets with BFC and their BICs, which could be used by researchers not only to evaluate algorithms for finding BICs, but also for other research purposes, such as training models of analyzing how bugs are introduced. Maybe those datasets could be biased, because they would only include bugs for which our method worked. But by improving compilability of past commits and transplanted tests, we think that the bias can be severely reduced, at least for some projects.

5.7 Threats to validity

Construct Validity. Our work is the first attempt to operationalize the *perfect test* method for identifying the change that introduced a bug (BIC), using regression tests. The method is based on the tests signaling the moment at which the bug was introduced, that is, we rely on regression tests as perfect detectors of the bug. Therefore, our proposal is subject to construct validity threats because it depends on the quality of the regression test, so its results when transplanted to the past could be less or more conclusive. To mitigate this, a bug dataset has been chosen where the regression tests have been previously checked and validated, ensuring that they are tests able to detect the existence of the reported bugs. Also, since there are bugs that are extrinsic (according to the definition by Rodríguez-Pérez et al. (2020)), these bugs do not have a BIC, so the perfect test method cannot find it. Therefore, our method, based on it, also can't.

Internal Validity. For our results, it is crucial that the reproduction of the execution of each snapshot is accurate, and exact as it would have executed at the moment the snapshot was produced. The Defects4J dataset tries to provide the libraries, commands and configurations needed to compile and execute the snapshot, but the environment provided by the dataset is not exactly the original one, which may produce differences in behavior.

External Validity. To conduct our study, we are limited to a dataset that provides all the prerequisites needed by the method: for each bug, the BFC, a regression test, the Git repository, etc. Thus, we only experimented with 809 bugs from 16 projects, all written in Java. It could happen that any conclusion is not directly translatable to other projects, to other languages, or to projects with different characteristics.

6 Conclusions and future work

In this paper we operationalize the theoretical method, called *perfect test*, to detect the change that introduced a bug (BIC), by using a regression test as *perfect test*. We show, using a well-known bugs dataset, that the method works for those bugs where we are able to transplant the regression test in the past and find a commit where this test passes again, by using our tool to automatically detect the BIC and then validating the results. However, we also find that our method is limited by the transplantability of regression tests to past snapshots, and in particular by the compilability of past snapshots.

As a result of applying our method, we produce, by a completely automated procedure, a dataset of BICs (BIC-RT), that can be used as ground truth for evaluating methods for detecting BICs. We apply it to some SZZ derivatives, proposing a method for evaluating their relative performance, and verifying a well-known limitation of them. This method could be exploited for producing, automatically, much larger collections of BICs. We also propose our method for automatically providing developers fixing a bug with detailed information about the BIC that introduced it.

Future lines of work can extend this study by exploring the application of the method on datasets of projects in other programming languages than Java (Python, JavaScript, C, C++ ...), of other types of projects (not only libraries), and in general to projects with different testing practices. Another line of research of major interest would be to study those bugs introduced along with the functionality. These bugs cannot be detected directly by our tool because it cannot automatically distinguish whether the regression test fails because the change that introduced the functionality that introduced the bug was found or because of a refactoring.

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Michel Maes-Bermejo. The first draft of the manuscript was written by Michel Maes-Bermejo and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. The research presented in this paper has been supported in part by the Government of Spain, through project Dependentium (PID2022-139551NB-I00).

Data Availability A documented reproduction package is public available in GitHub (https://github.com/codeurjc/BugHunter). It includes a link to an extra package in Zenodo (https://zenodo.org/record/8274835) (due to size limitations), with raw and processed results.

Declarations

Compliance with Ethical Standards This research has not involved human participants and/or animals.

Conflict of Interest The authors declared that they have no conflict of interest.

References

Ali NB, Engström E, Taromirad M, Mousavi MR, Minhas NM, Helgesson D, Kunze S, Varshosaz M (2019) On the search for industry-relevant regression testing research. Empir Softw Eng 24(4):2020–2055

An G, Yoo S (2021) Reducing the search space of bug inducing commits using failure coverage. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1459–1462

Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1):11–33. https://doi.org/10.1109/TDSC.2004.2

Barr ET, Brun Y, Devanbu P, Harman M, Sarro F (2014) The plastic surgery hypothesis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 306–317

Barr ET, Harman M, Jia Y, Marginean A, Petke J (2015) Automated software transplantation. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 257–269

Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009) The promises and perils of mining git. In: 2009 6th IEEE International Working Conference on Mining Software Repositories, pp. 1–10. IEEE

Bludau P, Pretschner A (2022) PR-SZZ: How pull requests can support the tracing of defects in software repositories. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1–12. IEEE

Borg M, Svensson O, Berg K, Hansson D (2019) SZZ Unleashed: An open implementation of the SZZ algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, pp. 7–12

Castelluccio M, An L, Khomh F (2019) An empirical study of patch uplift in rapid release development pipelines. Empir Softw Eng 24(5):3008–3044

Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms. p. 540-549. MIT press

Da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2016) A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans Softw Eng 43(7):641–657

Davies S, Roper M, 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

Desikan S, Ramesh G (2006) Software testing: principles and practice. pp. 193–208. Pearson Education India

Engström E, Runeson P (2010) A qualitative survey of regression testing practices. In: International Conference on Product Focused Software Process Improvement, pp. 3–16. Springer

Gousios G, Pinzger M, Deursen Av (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering, pp. 345–355

Guo PJ, Zimmermann T, Nagappan N, Murphy B (2010) Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1, pp. 495–504

Harman M (2010) Automated patching techniques: the fix is in: technical perspective. Communications of the ACM **53**(5), 108–108

IEEE Standard Classification for Software Anomalies (2010) IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) pp. 1–23. https://doi.org/10.1109/IEEESTD.2010.5399061
Iida H, Matsumoto K (2016) Improving the High-Impact Bug Reports: A Case Study of Apache Projects
Jang J, Agrawal A, Brumley D (2012) ReDeBug: finding unpatched code clones in entire OS distributions. In: 2012 IEEE Symposium on Security and Privacy, pp. 48–62. IEEE
Jiang Y, Liu H, Niu N, Zhang L, Hu Y (2021) Extracting concise bug-fixing patches from human-written patches in version control systems. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 686–698. IEEE
Just R, Jalali D, Ernst MD (2014) Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 437–440
Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2012) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773
Khattar M, Lamba Y, Sureka A (2015) Sarathi: Characterization study on regression bugs and identification of regression bug inducing changes: A case-study on Google Chromium project. In: Proceedings of the 8th India Software Engineering Conference, pp. 50–59
Kim S, Whitehead Jr EJ (2006) How long did it take to fix bugs? In: Proceedings of the 2006 international workshop on Mining software repositories, pp. 173–174
Kim S, Zimmermann T, Pan K, James Jr E et al (2006) Automatic identification of bug-introducing changes. In: 21st IEEE/ACM international conference on automated software engineering (ASE'06), pp. 81–90. IEEE
Lawrence S, Pennock DM, Flake GW, Krovetz R, Coetzee FM, Glover E, Nielsen FA, Kruger A, Giles CL (2001) Persistence of web references in scientific research. Computer 34(2):26–31
Lenarduzzi V, Palomba F, Taibi D, Tamburri DA (2020) Openszz: A free, open-source, web-accessible implementation of the SZZ algorithm. In: Proceedings of the 28th international conference on program comprehension, pp. 446–450
Levin S, Yehudai A (2017) The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 35–46. IEEE
Maes-Bermejo M, Gallego M, Gortázar F, Robles G, Gonzalez-Barahona JM (2022) Revisiting the building of past snapshots — a replication and reproduction study. Empir Softw Eng 27(3)
McIntosh S, Kamei Y (2018) Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Trans Softw Eng 44(5):412–428
Neto EC, Da Costa DA, 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), pp. 380–390. IEEE
Neto EC, Da Costa DA, Kulesza U (2019) Revisiting and improving SZZ implementations. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–12. IEEE
Onoma AK, Tsai WT, Poonawala M, Suganuma H (1998) Regression testing in an industrial environment. Commun ACM 41(5):81–86
Perscheid M, Siegmund B, Taeumel M, Hirschfeld R (2017) Studying the advancement in debugging practice of professional software developers. Software Quality Journal 25(1):83–110
Petrulio F, Ackermann D, Fregnan E, Çalikli G, Castelluccio M, Ledru S, Denizet C, Humphries E, Bacchelli A (2022) SZZ in the time of Pull Requests. IEEE Trans Softw Eng
Pinto LS, Sinha S, Orso A (2012) Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, pp. 1–11
Pokropiński, J., Gasiorek, J., Kramarczyk, P., Madeyski, L.: SZZ Unleashed-RA-C: An Improved Implementation of the SZZ Algorithm and Empirical Comparison with Existing Open Source Solutions. In: Developments in Information & Knowledge Management for Business Applications, pp. 181–199. Springer (2022)
Purushothaman R, Perry DE (2005) Toward understanding the rhetoric of small source code changes. IEEE Transactions on Software Engineering 31(6):511–526
Rodríguez-Pérez G, Nagappan M, Robles G (2020) Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the OpenStack project. IEEE Trans Softw Eng
Rodríguez-Pérez G, Robles G, González-Barahona JM (2018) Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. Inf Softw Technol 99:164–176
Rodríguez-Pérez G, Robles G, Serebrenik A, Zaidman A, Germán DM, Gonzalez-Barahona JM (2020) How bugs are born: a model to identify how bugs are introduced in software components. Empir Softw Eng 25(2):1294–1340

Rosa G, Pascarella L, Scalabrino S, Tufano R, Bavota G, Lanza M, Oliveto R (2021) Evaluating SZZ implementations through a developer-informed oracle. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 436–447. IEEE

Saha R, Gligoric M (2017) Selective bisection debugging. In: International Conference on Fundamental Approaches to Software Engineering, pp. 60–77. Springer

Schmidt DC, Porter A (2001) Leveraging open-source communities to improve the quality & performance of open-source software. In: Proceedings of the 1st Workshop on Open Source Software Engineering, vol. 1. Citeseer

Sidiroglou-Douskos S, Lahtinen E, Eden A, Long F, Rinard M (2017) Codecarboncopy. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 95–105

Sinha VS, Sinha S, Rao S (2010) Buginnings: identifying the origins of a bug. In: Proceedings of the 3rd India software engineering conference, pp. 3–12

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? ACM sigsoft software engineering notes 30(4):1–5

Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Empirical Softw Eng 19(6):1665–1705

Tantithamthavorn C, Teekavanich R, Ihara A, Matsumoto Ki (2013) Mining a change history to quickly identify bug locations: A case study of the eclipse project. In: 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 108–113. IEEE

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2017) There and back again: Can you compile that snapshot? Journal of Software: Evolution and Process 29(4):e1838

Wahl NJ (1999) An overview of regression testing. ACM SIGSOFT Software Engineering Notes 24(1):69–73

Weiss C, Premraj R, Zimmermann T, Zeller A (2007) How long will it take to fix this bug? In: Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007), pp. 1–1. IEEE

Wen M, Wu R, Liu Y, Tian Y, Xie X, Cheung SC, Su Z (2019) Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM

Williams C, Spacco J (2008) Branching and merging in the repository. In: Proceedings of the 2008 international working conference on Mining software repositories, pp. 19–22

Williams C, Spacco J (2008) SZZ revisited: verifying when changes induce fixes. In: Proceedings of the 2008 workshop on Defects in large software systems, pp. 32–36

Yang D, Mao X, Chen L, Xu X, Lei Y, Lo D, He J (2022) Transplantfix: Graph differencing-based code transplantation for automated program repair. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–13

Yin Z, Yuan D, Zhou Y, Pasupathy S, Bairavasundaram L (2011) How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 26–36

Yoon YS, Myers,BA (2012) An exploratory study of backtracking strategies used by developers. In: 2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE), pp. 138–144. IEEE

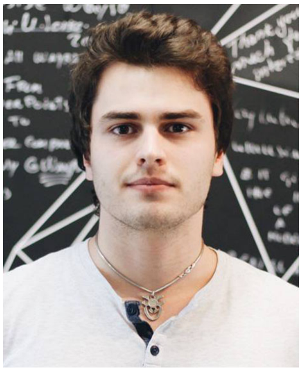
Yoon YS, Myers BA (2014) A longitudinal study of programmers' backtracking. In: 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 101–108. IEEE

Zeller A (2002) Isolating cause-effect chains from computer programs. ACM SIGSOFT Software Engineering Notes 27(6):1–10

Zeller A, Hildebrandt R (2002) Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng 28(2):183–200

Zerouali A, Constantinou E, Mens T, Robles G, González-Barahona J (2018) An empirical analysis of technical lag in npm package dependencies. In: International Conference on Software Reuse, pp. 95–110. Springer

Zhang T, Kim M (2017) Automated transplantation and differential testing for clones. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 665–676. IEEE



Michel Maes-Bermejo is a post-doc researcher in the Computing Science Department at University Rey Juan Carlos, Móstoles, Madrid, Spain. His research interests include bug introducing changes, empirical software engineering, software maintenance and evolution and mining software repositories

Alexander Serebrenik is a Full Professor of software evolution at the Eindhoven University of Technology, The Netherlands. His research interests include a wide range of software maintenance and evolution topics, from source-code analysis to collaborative and human aspects of software engineering.

Micael Gallego is an Associated Professor in the Computing Science Department at University Rey Juan Carlos, Móstoles, Madrid, Spain. His main research activities are related to software engineering and software testing.

Francisco Gortázar is an Associated Professor in the Computing Science Department at University Rey Juan Carlos, Móstoles, Madrid, Spain. His main research activities are related to software engineering and software testing.

Gregorio Robles is a Full Professor at the Universidad Rey Juan Carlos, Madrid, Spain. He mainly does research in the following two fields: a) mining software repositories (socio-technical issues such as community metrics, software evolution, and development effort estimation of F/OSS); and b) computational thinking (with evaluation tools such as DrScratch).

Jesús María González Barahona is a Full Professor in Telematics Engineering at Universidad Rey Juan Carlos (Móstoles, Madrid). With over 20 years of teaching experience, he specializes in computer networks, data transmission, and telematic services and protocols. His research interests revolve around the empirical study of software development, quantitative methods for activity and process analysis, and data visualization in extended reality (VR and AR).

Authors and Affiliations

Michel Maes-Bermejo[1] · Alexander Serebrenik[2] · Micael Gallego[1] · Francisco Gortázar[1] · Gregorio Robles[3] · Jesús María González Barahona[3]

Alexander Serebrenik
a.serebrenik@tue.nl

Micael Gallego
micael.gallego@urjc.es

Francisco Gortázar
francisco.gortazar@urjc.es

Gregorio Robles
gregorio.robles@urjc.es

Jesús María González Barahona
jesus.gonzalez.barahona@urjc.es

1 Department of Computer Science, Universidad Rey Juan Carlos, Madrid, Spain

2 Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, Netherlands

3 Department of Telematic and Computational Systems Engineering, Universidad Rey Juan Carlos, Madrid, Spain