Received: 25 July 2022 | Revised: 24 March 2023 | Accepted: 27 May 2023

DOI: 10.1002/stvr.1857

RESEARCH ARTICLE

WILEY

In vivo test and rollback of Java applications as they are

Antonia Bertolino[1] | **Guglielmo De Angelis**[2] | **Breno Miranda**[3] | **Paolo Tonella**[4]

[1]ISTI–CNR, Pisa, Italy

[2]IASI–CNR, Rome, Italy

[3]Federal University of Pernambuco, Recife, Brazil

[4]Università della Svizzera italiana, Lugano, Switzerland

Correspondence
Guglielmo De Angelis, IASI–CNR, Rome, Italy.
Email: guglielmo.deangelis@iasi.cnr.it

Funding information
Italian MIUR PRIN 2017 Project, Grant/Award Number: SISMA 201752ENYB; European Research Council Project, Grant/Award Number: 787703; Precrime; Italian Research Group: INdAM-GNCS

Summary

Modern software systems accommodate complex configurations and execution conditions that depend on the environment where the software is run. While in house testing can exercise only a fraction of such execution contexts, in vivo testing can take advantage of the execution state observed in the field to conduct further testing activities. In this paper, we present the Groucho approach to in vivo testing. Groucho can suspend the execution, run some in vivo tests, rollback the side effects introduced by such tests, and eventually resume normal execution. The approach can be transparently applied to the original application, even if only available as compiled code, and it is fully automated. Our empirical studies of the performance overhead introduced by Groucho under various configurations showed that this may be kept to a negligible level by activating in vivo testing with low probability. Our empirical studies about the effectiveness of the approach confirm previous findings on the existence of faults that are unlikely exposed in house and become easy to expose in the field. Moreover, we include the first study to quantify the coverage increase gained when in vivo testing is added to complement in house testing.

KEYWORDS

coverage, empirical results, field failures, in vivo testing, isolation, Java platform

1 | INTRODUCTION

In recent years the practice of continuing software testing in production has passed from a bad reputation, symptomatic of a poor quality assurance process, to its acceptance as an indispensable strategy for mastering the growing complexity and dynamism of configurations and environmental conditions. Both in the white and in the grey literature, this practice is referred to using different terms, which include "field testing," "live testing," "post-release testing," and "in vivo testing." For the purposes of this paper, we consider all such terms as synonyms.

Despite the common denominator of analysing the behaviour of an application under test (AUT) while it is in operation and accessed by actual users, in reality the practice of field testing can comprise quite different approaches and strategies. Some approaches, such as canary releases, dark launches, or A/B testing, consist of controlled online experiments [1] in which an organization aims to evaluate some change by measuring the users' reactions. This kind of live testing is especially useful in continuous deployment processes for supporting data-driven decisions [2,3], and mostly consists of passively monitoring a system configuration in the field. We do not consider controlled online experiments in this work.

Different is the case of field testing approaches that actively intervene on the AUT, either by launching test invocations mimicking users' invocations, or by altering the system state. Such proactive techniques aim at detecting field failures before the actual users experience them, trying out the AUT resilience to (intentional or unintended) breakages. Indeed, despite all the efforts, faults that escaped in house testing will eventually manifest themselves after the software is released for production use. An early report on the impacts of inadequate infrastructure for software testing estimated

that 10% of the faults introduced during the coding/unit testing phase are found only after the product is released [4]. Those faults are missed not necessarily because the testing process is poor; some faults are genuinely hard to detect in house, mainly because it is hard to predict all the interactions that will occur in the production environment. More recent studies [5,6] highlight that a large percentage of failures occurring in the field yield characteristics that would be difficult and costly to reproduce in the lab. Hence, there are signals indicating that this trend of *right-shifting* testing [7] towards the production stage is here to stay. In the latest World Quality Report [8], organizations were asked to rate the importance of mechanisms for improving test efficiency, and 40% of the respondents qualified "shift test right" as essential or very important. Correspondingly, we are experiencing a growing interest in this topic in academic research, as reported in a systematic study of the literature [9].

Specifically the empirical study by Bertolino et al. [9] collected 80 scientific works proposed until 2017, and they found that the most substantial activities on field testing started only after 2002. In addition, the survey reports that most of its primary studies have been developed between 2007 and 2017 (with an average of almost 5 works on the topic per year). More recently Silva et al. [10] performed a similar study, limited to papers published between 2012 and 2021, and focusing only on self-adaptive testing approaches in the field. From this last empirical study, it appears that 6 additional works have been developed between 2018 and 2021 addressing a specific context for field testing. In summary, although field testing is a specific research area within software testing, nevertheless it receives constant interest by both researchers and practitioners. Thus, despite the many challenges it poses, field testing is perceived as an important complement to in house testing, especially when the configuration space the AUT can operate on becomes huge.

While it would be desirable that tests launched in production do not interfere with actual users' invocations, if executed without any protection mechanism, they usually would impact the state of both the AUT and the environment, producing visible side effects. By and large these effects can be contrasted taking three different approaches: (i) just accepting them; (ii) isolating the test execution from live operations; (iii) rolling-back the AUT to its original state after testing terminates.

The first approach is taken, for instance, in Chaos Monkey testing [11], introduced by Netflix, in which outages are intentionally and regularly caused in production to test the network resilience. However, not all systems are apt for tolerating breakages without serious consequences, and moreover functional testing aiming at finding field failures may require more sophisticated interventions, for example, launching end-to-end tests in production, than just introducing a disconnection.

The Invite framework by Murphy et al. [12] is an example of the second approach to in vivo testing. Its solution consists of isolating the test execution context from the operation environment, by duplicating the system state within a separate sandbox. Such type of solution mitigates the risk that any impact leaks from testing to production. However, it can be demanding in terms of resources required, and hence may not scale up to large AUTs).

This paper presents the Groucho framework, which is a solution pertaining to category (iii). Groucho supports checkpointing, testing and reverting of the AUT, while other threads not involved in the online testing session are suspended, to be resumed when the test is over.

A preliminary version of Groucho has been early introduced in a prior work [13]. Both Invite and that prior version of Groucho require, as a prerequisite for application, that the source code of the AUT undergoes some substantial manipulation. This requirement assumes that either the AUT has been prepared *ad hoc* by its developers who anticipate the subsequent needs of vivo testing, or its source code can be accessed and modified by the testers. This underlying prerequisite can be an issue because the source code is not always accessible or, even if accessible, the version modified for testing purposes cannot be released in production.

In this paper, we introduce an extended version of the Groucho framework that can support in vivo testing of Java applications *as they are*, that is, without requiring any code modification and without even requiring the availability of AUT source code. To the best of our knowledge Groucho provides the first unobtrusive field testing framework.

We have evaluated Groucho along several directions: We have compared it against Invite in a qualitative evaluation (we could not conduct a quantitative study because Invite is not made available); we have evaluated empirically the overhead implied by the introduction of in vivo testing; and finally we have evaluated empirically the functionality of Groucho, showing that indeed Groucho can detect failures that would escape in house testing (as previously also shown with regard to Invite). We also conducted a study comparing the coverage achieved by Groucho against that reached in the lab. While several works claim that in vivo testing can provide a more extensive coverage of stateful applications, this is the first study actually providing such an evidence.

This paper builds on the results from our previous work [13] and extends them in several ways. Overall, the main novel contributions specific of this work are:

- an extended version of the Groucho framework that supports unobtrusive in vivo testing (see Section 4.2, Section 4.4);
- a qualitative comparison between Groucho and Invite (see Section 5);

- a comparative study on the effectiveness of testing activities performed in vivo or in the lab, considering both the coverage they reach and the faults they detect (see Section 7).

The paper is structured as follows: In the next section, we overview relevant related work; then in Section 3 we briefly introduce as an illustrative example a small AUT. The details of the solutions implemented in Groucho are given in Section 4, while the qualitative comparison between Invite and Groucho is reported in Section 5. The results of the conducted empirical studies are reported and discussed in Section 6 for what concerns the overhead evaluation, and in Section 7 with regard to Groucho's effectiveness. Finally we discuss the threats to validity affecting all three studies in Section 8 and draw conclusions in Section 9.

2 | RELATED WORK

Recent literature provides evidence that field failures are often difficult to reproduce during pre-release, in house testing. We report about such evidence, and then continue with a summary of field testing approaches proposed to address the problem.

2.1 | Field Failures

Testing in the field is motivated by several studies that analysed the prevalence and the characteristics of failures arising in production. Gazzola et al. [5] performed a wide empirical investigation of field failures in three open source systems (Eclipse, OpenOffice and Nuxeo). The authors inspected 119 bug reports and concluded that 70% of the faults could be classified as *field-intrinsic faults*, that is, faults that would be hard to reveal during in house testing. The main cause to miss field-intrinsic failures was found to be the combinatorial growth of the states to be tested.

In order to improve their field quality predictions, Li et al. [14] examined usage information and failure reports from millions of Windows 7 OS users. Among their findings, the authors identified the most important usage characteristics that differentiate pre-release and post-release versions, including the number of applications executed in the user machine and the type of installation.

Another study over industrial software (37 projects by BHL BNP Paribas) was conducted by Rwemalika et al. [6] who analysed the differences between pre-release and post-release bug fixes. Their conclusions support the intuition that post-release patches are larger than pre-release ones, and are also more dispersed among several source code files and configuration files.

2.2 | Approaches to Field Testing

All the above studies motivate to continue testing beyond the deployment boundary [7]. Indeed, as we report in a recent systematic study [9] to which we refer for a more extensive survey of literature, many different approaches have been proposed.

Some authors [15,16] observe the system in the field to collect data and information on its actual usage, but then use such data and information to test the system in the lab. This approach is sometime referred to as ex vivo testing. A notable example of ex vivo testing is Pankti [17], an approach that monitors in-field executions and collects serialized data to automatically construct off-line unit test cases. The tests generated by Pankti perform differential testing and make use of a differential test oracle based on the execution results observed in production. Another approach that does not interfere with field execution is called *passive testing* [18]. In the passive testing approach, no test input is provided: The system is observed (i.e., monitored) while in operation, and the traces produced are collected and analysed. Originally proposed for failure detection in network management [19], different passive testing techniques have been investigated with the goal of improving trace analysis. They have been applied in several domains, such as, among others, service-oriented systems [20] and timed systems [21].

Other authors proposed instead to launch test invocations on the software directly in the deployment/production environment. Actually, this has been advocated as a viable alternative for testing of service-oriented architectures [22,23], for which testing in house may be impossible. In other cases, field testing is motivated by the opportunity of leveraging the participation of a wide number of machines using different configurations, as in the early *Skoll* approach to distributed quality assurance [24]. At Netflix, engineers inject disruptions into the production system in order to improve resilience, because some failures cannot be easily exercised in house or in a simulated environment. This approach is known as *Chaos Engineering* [25].

In between passive observation and chaos, some frameworks have been proposed that try to limit the impact of proactively launching test cases in production. The RTF4ADS framework [26] supports run-time testing by using several strategies. In particular, it can apply different policies for isolating the on-line tests from operation, depending on the system to be tested. Such isolation policies may include cloning the system under test, blocking user-requests, using a dedicated built-in-test interface, collaborating with a tagged test-aware system, or adopting aspects. While proposing an extensive and articulated framework, built on the standard TTCN-3, this approach specifically targets reconfiguration actions at system architecture level. In comparison, our approach addresses unit testing of Java applications. The ATLAS architecture [27] for runtime testing leverages built-in-testing and assumes that the component under test is aware of being field-tested. Thus it is the component that must ensure that field testing does not interfere with the normal operation, while in our approach the AUT is transparently isolated by our framework.

The approaches closer to this work are those that test a system in production in order to observe its behaviour under a state that is difficult or impossible to reproduce in house, and among them the most notable one is Invite [12]. The Invite framework supports in vivo testing of Java applications that have been previously instrumented. Then, when the AUT is running in production, its in vivo tests are launched with some pre-defined probability. Test execution happens in a separate *sandbox*, which replicates the production environment. The main disadvantage of this approach is that it can introduce a large resource overhead, associated with the creation of a sandboxed replication of the execution environment. With the aim of reducing such overhead, we developed the Groucho framework, which supports in vivo testing of Java applications using an opportunistic strategy that isolates only the objects tested in the field, and selectively suspends/resumes the existing threads. Our experimental results show that Groucho's performance overhead is negligible when a reasonably low activation probability of in vivo test execution is set (see Section 6). Moreover, while Invite requires that the AUT undergoes some non-trivial preparation by the software engineering teams, Groucho does not require any previous preparation of the AUT nor even its source code availability (see Section 5).

3 | ILLUSTRATIVE EXAMPLE

For illustrative purposes, this section introduces a simple example that will be used through the rest of the paper in order to discuss the in vivo testing process, the technical assumptions associated with the referred frameworks, the responsibilities assigned to the AUT software engineering teams (i.e., design, development, and deployment teams), and the associated implications on the life-cycle of the considered AUT.

```
1 package org.application.utils;
2 ...
3 public class ListOfElements {
4     ...
5     public Object setElementAt(int index, Object element) {
6         ...
7     }
8         ...
9 }
```

Listing 1 Class under in vivo testing

Specifically, let us assume that we are interested in launching an in vivo test campaign over the class `ListOfElements` reported in Listing 1. More precisely, each time the method `setElementAt` is invoked, before its execution an in vivo testing session is activated and the launched in vivo test execution will refer to the actual state reached by the AUT (i.e., the runtime state of `org.application.utils.ListOfElements`). Indeed, after that Groucho automatically configures both the current context and the referred formal parametes, existing (e.g., manually designed) test cases are executed on such in vivo states.

4 | IN VIVO TEST AND ROLLBACK APPROACH

Groucho provides an automation framework for in vivo testing thanks to the following features:

(i) declarative source-code annotation for enabling in vivo testing (see Section 4.1);
(ii) on-the-fly manipulation of plain AUT binaries (see Section 4.2);
(iii) thread-safe rollback capability during in vivo testing (see Section 4.3);
(iv) multiple isolation layering within one in vivo test session (see Section 4.4).

Features (i) and (ii) are somehow alternative, or better complementary: The former relies on the assumption that Test Engineers are involved in the development process of the AUT and can manually annotate its source-code; on the other hand, the case referred by item (ii) releases such assumption and does not require any previous preparation of the AUT, nor even its source code availability like in the case of 3rd party or legacy libraries.

In addition to the above features that are specific to the Groucho framework, by its same definition any in vivo testing approach requires that the test cases are launched from some state/configuration the AUT will actually meet at run-time. Therefore the potential of in vivo testing is best achieved by using parametric and configurable test cases (see Section 4.5).

4.1 | Declarative annotation of AUT source code

When planning the execution of in vivo testing, Test Engineers are in charge of selecting the units under test, and of defining and customizing the activation policies for the test cases to be executed at run-time. Hence, Groucho works under the assumption that Test Engineers are knowledgeable about organization and responsibilities of the main classes in the AUT, and are also aware of implementation details relevant for in vivo testing.

Similarly to other in vivo testing frameworks (e.g., Invite [12], as also detailed in Section 5), the basic use case in Groucho expects that tests are selected and enabled for in vivo execution in advance, before compiling the AUT source code [13]. The granularity for both test case selection, and activation policies is at method level. Test Engineers manually annotate the AUT producing the instrumented version AUT + in vivo (SRC) (see Figure 1a),

```
1  package org.application.utils;
2  ...
3  public class ListOfElements {
4      ...
5      @TestableInVivo(invivoTestClass = "org.invivoTestPackage.InVivoTestClass", invivoTest = "
6          invivoTestMethod")
7      public Object setElementAt(int index, Object element) {
8          ...
9      }
10         ...
11 }
```

Listing 2 Enabling in vivo testing with Groucho by declarative annotations

Specifically, the manual annotation of the AUT source code requested by Groucho is relatively lightweight and relies on the Java Annotation technology. Each annotation declares the methods subject to in vivo testing and, for each of them, the test program to be executed when an in vivo testing session is launched. With reference to

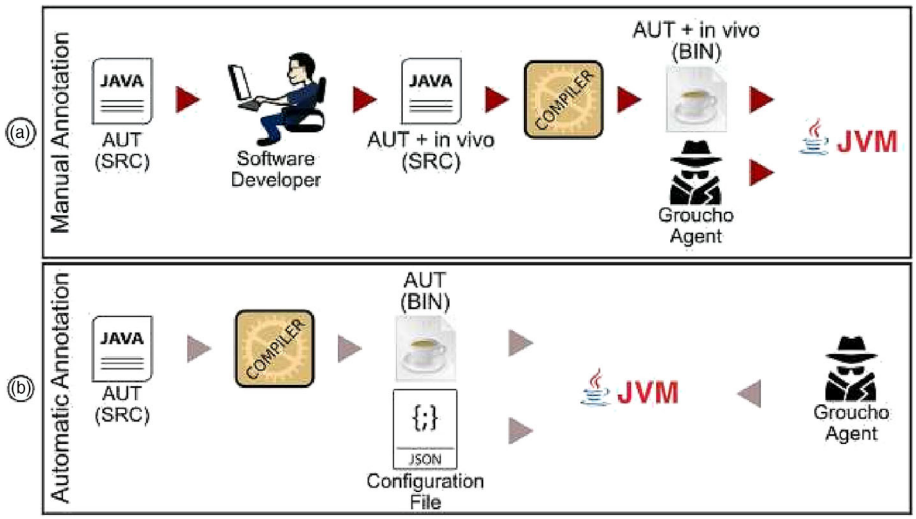

FIGURE 1 Steps required to apply Groucho: Manual source-code annotation (a) and automatic binary injection (b)

the example of Listing 1, the instrumented version enabling in vivo testing by means of Groucho is shown in Listing 2.

The in vivo test annotations to be added for Groucho target methods definitions [13]. Any method potentially subject to in vivo testing activities (e.g., `setElementAt`) is marked with a Java annotation (see line 5 in Listing 2), which declares meta-information that includes the name of the test case to be executed in vivo (i.e., `invivoTestMethod`), coded as a (public) method of a given Java class (i.e., `org.invivoTestPackage.InVivoTestClass`).

4.2 | On-the-fly manipulation of AUT binaries

Beyond the case described in the previous section, there can be development contexts in which the canonical iterations over software design and development might be completely agnostic of any kind of in vivo testing activities, either foreseen at production time or not. To account for such contexts, Groucho also enables in vivo testing without any assumption on the life-cycle of the considered AUTs). As we depict in Figure 1b, the original AUT (SRC) can be compiled into the AUT (BIN) and only afterwards Groucho is applied to the plain AUT binaries. Therefore, in order for Groucho to work, we do not necessarily need to modify Listing 1 with a manual annotation of the source-code, as discussed in Section 4.1.

In this scenario, Groucho applies Java bytecode analysis and manipulation techniques to the binary version of the considered AUT, deployed on any off-the-shelf JVM. The launch of an in vivo testing campaign over such JVM refers to the Groucho agent and to a JSON configuration file (see Figure 1b).

The latter allows Software Testers to specify the classes that will be subject to in vivo testing. In particular, for each class, it is sufficient to declare the methods acting as entry points for the in vivo sessions, the test program to be executed in vivo, plus other optional parameters that affect some isolation policies (e.g., data consistency in multi-thread applications). In Listing 3 we report such JSON configuration file for the illustrative example introduced in Section 5.

```
1  {
2    ...
3    "lst" : [
4    ..., {
5      "className" : "org.application.utils.ListOfElemenents",
6      "methodName" : "setElementAt",
7      "annotation" : {
8        "invivoTestClass" : "org.invivoTestPackage.InVivoTestClass",
9        "invivoTest" : "invivoTestMethod",
10       "carvingDepth" : 1,
11       "pauseOtherThreads" : true
12     }
13   }, ... ]
14   ...
15 }
```

Listing 3 Enabling in vivo testing with Groucho: Example of JSON configuration file

The Groucho agent is responsible for both the analysis of each class the JVM dynamically loads and the on-the-fly manipulation of the bytecode. Specifically, each time the JVM loads in memory the definition of an undeclared class, the Groucho agent checks if the name of such novel class is included in the JSON configuration file. In case the class is found, Groucho scans the class bytecode, looking for the declarations of its methods. For each method that is also listed in the JSON configuration file, Groucho processes the method declaration by plugging into it a jump-off to the in vivo testing session and by configuring it with the parameters reported in the JSON file. When the bytecode transformation is over, the manipulated declaration of the class is actually included within the JVM run-time data structures.

Hence, in vivo testing by Groucho does not force the AUT producers to *ad hoc* instrument their applications with dedicated features, neither to deal with different releases (production vs testing) of the same application. Also, solution/service providers have not to adapt their integration with 3rd party or legacy libraries in case they decide to take advantage of in vivo capabilities.

Once the modified bytecode of a class has been loaded into the JVM, any of its instances is transparently referred and used by the other class instances in the application.

4.3 | Thread-safe test and rollback

Whenever the JVM fetches an invocation to one of the class methods that are subject to in vivo testing, the run-time layer of Groucho evaluates the activation of a testing session[1]. When an in vivo testing session is activated, Groucho enforces a set of isolation policies following a "test & rollback" approach: to do this, Groucho leverages the framework Crochet [28], which provides a general purpose approach enabling fully automatic and fine-grained manipulation of runtime objects in the JVM, by altering the behaviour coded in some of their methods. Crochet uses such manipulation in order to support a lightweight copy of individual objects and an opportunistic, lazy copy-propagation over their references across the JVM heap. In addition, Groucho conservatively grants execution only to the JVM thread that enabled the in vivo session. More selective thread management policies can be enforced, by specifying them as parameters of the JSON configuration file (see `pauseOtherThreads` in Listing 3).

Thus, Groucho isolates the state of all in-memory objects affected by in vivo testing. The isolation procedure follows a lazy strategy: Only when an instance is explicitly invoked, its state is copied and saved. Once the in vivo testing session is over, before resuming the canonical execution of the AUT, Groucho rolls back the state of all saved instances to the state they had just before the AUT entered the in vivo testing session. Furthermore, all the other JVM threads are also rolled back either to runnable or to any other thread-state associated with their blocking conditions.

4.4 | Multiple isolation layering

Concerning isolation, Groucho distinguishes between in-memory data, and persisted/exported data. In the former case, Groucho performs a lazy deep copy of the objects involved in the testing session, leaving all other objects unaffected. The in-memory isolation of Groucho is applicable also to multi-thread applications [13]. In the latter case, similarly to other in vivo testing frameworks (e.g., Invite [12]), Groucho does not offer any general solution for dealing with data external to the memory of the AUT. It is under the responsibility of the AUT software engineers to foresee the needed isolation procedures. Specifically, such additional isolation procedures have to be invoked within the definition of the in vivo test method referred by the annotation `@TestableInVivo` (i.e., invivoTest = ∼`invivoTestMethod`∼, see Listing 2 at line 5).

It should be noted that the in-memory isolation policies offered in previous work [13] are limited to prevent any user data corruption due to the execution of a single in vivo testing session: Once an in vivo testing session is started, the isolation policies are enforced, while they are reverted once the in vivo test execution ends. Correspondingly, such solution allows for the execution of *just one test case per in vivo session*, as the AUT state is saved and then restored only once. To the best of our knowledge this limitation holds also for Invite [12]. In this work, we address such limitation by introducing the possibility to compose multiple isolation layers within a single in vivo testing session.

Specifically, developers of the test program to be executed in vivo can programmatically query Groucho asking for the activation of one or more additional isolation layers. The objective of each additional layer is to preserve the current state of the instances referred within the same in vivo testing session. At the same time, Groucho can be asked to restore such preserved states locally to the on-going in vivo session. In this way, Groucho enables the possibility to isolate the evolution of multiple computations, all of them starting from a known initial state, to which it is always possible to rollback. Thanks to this functionality, developers can plan the execution of multiple and independent tests within the same in vivo testing session, each of which can start from the desired application state (e.g., all from the same state, encountered at the beginning of the in vivo testing session).

Listing 4 reports an example of an in vivo test where multiple test cases are executed independently. All of them start from the same initial state (i.e. the state the AUT has before the activation of the in vivo test session). According to the configuration reported in Listing 3, whenever Groucho activates an in vivo testing session, the state of the AUT is saved and `invivoTestMethod` is invoked (see Listing 4, line 5). As a first step the in vivo test accesses the current instance under test (line 7). Then, it enables Groucho to host multiple isolation layers (line 9). When inside the in vivo testing session (i.e., inside `invivoTestMethod`), for each test to be executed in vivo and possibly obtained from an in house test suite (lines 12-26), line 15 shows how to query Groucho in order to apply an additional isolation layer on top of the considered instance under test (and its referred instances, if needed); thereafter, line 23 reverts all the modifications done since the latest isolation layer. The actual invocation of each (in house) test takes place between these two statements (see lines 18-19).

[1]The investigation of advanced, intelligent activation policies and their evaluation is part of our future research.

If the intent of the Tester Engineers is to explicitly combine the effects of two or more (in house) tests they can consider to remove the statements at line 15 and line 23 at Listing 4, or to surround them with appropriate decision logic. These modifications would create dependencies among the test cases, because they would be run from an AUT state that is affected by the execution of all the other tests launched after the creation of the latest isolation layer.

For the sake of clarity we remark that the isolation features of Groucho are limited to in-memory data. As said it is the responsibility of the in vivo testers to implement any needed strategies to isolate data external to the memory of the AUT (see line 16 and line 24).

```
1  package org.invivoTestPackage;
2  ...
3  public class InVivoTestClass {
4      ...
5      public boolean invivoTestMethod(Context c) throws InvocationTargetException{
6          ...
7          ListOfElements list = ((ListOfElements) c.getInstrumentedObject());
8          ...
9          RuntimeEnvironmentShield shield = new RuntimeEnvironmentShield();
10         ...
11         // Execute in vivo some in house test from ListOfElementsInHouseTestClass
12         for (Method method: getSomeInHouseTestMethods()){
13             ...
14             try {
15                 shield.applyCheckpoint(list);
16                 // Isolate data from persistency if needed
17
18                 ListOfElementsInHouseTestClass inhouseTest = new ListOfElementsInHouseTestClass(list)
19                     ;
20                 method.invoke(inhouseTest);
21             }
22             ...
23             finally {
24                 shield.applyRollback(list);
25                 // Restore data from persistency if needed
26             }
27         }
28         ...
29         return getInvivoTestExitStatus();
30     }
31     ...
32 }
```

Listing 4 Example of multiple isolation layering

4.5 | Parametric and configurable test cases

The benefits of in vivo testing are conditioned to the possibility to execute the tests from some state/configuration the AUT actually reaches at run-time. Consequently, it is important that any test code to be executed in vivo is implemented as a parametric test with parametric assertions, where parameters are introduced to let the test refer to the objects/variables being observed in the AUT at runtime, in the field, and to let the assertions compute the expected outcome of the test based on such objects/variables. Thus Groucho requires that testing engineers develop parametric and configurable test cases (this is also the case of other in vivo testing framework such as Invite [12]).

While this requirement may imply some additional effort for the software engineering team, which may have, for example, to re-engineer the existing testing code, on the other hand the practice of keeping the configuration part of the tests separate from their actual implementation is today widely used and is also promoted by the most common frameworks for writing and executing test programs [29]. In this respect, the assumption on parametric test cases can be considered viable and maybe even desirable in many realistic settings.

Listing 5 reports an example of parametric configuration that Groucho would require on top of an in house test program, in order to make it executable also in vivo, on the runtime AUT state. Specifically, any reference declaration to the instance under test is removed from all the test methods, and is included as an attribute of the test class (see line 6). While the in house test class may already include its own default constructor (line 10), test engineers are required to include either another parametric constructor (line 16), or another configuration mechanism that could be invoked for the initialization of the instance under test. Finally, the implementation of each test should avoid statements referring to hard coded constants that hold the values observed during in house testing, as those values might change during in vivo testing. Such constants should be replaced with expressions that compute the associated values from test parameters. The assertions should be also formulated in terms of parametric logical expressions (lines 23-31).

```
package org.application.utils.tests;
...
public class ListOfElementsInHouseTestClass {
   ...
// Subject of the test
   private ListOfElements listUnderTest;
   private int minSize;
   ...
// Default Configuration
   public ListOfElementsInHouseTestClass (){
      this.listUnderTest = new ListOfElements(10);
      this.minSize=0;
      ...
   }
// Configuration to be used while running an in vivo session
   public ListOfElementsInHouseTestClass (ListOfElements list) {
      this.listUnderTest = list;
      this.minSize=0;
      ...
   }
   ...
   @Test
   public void replaceElementTest(){
      int index = RandomGenerator.nextInt(this.minSize, this.listUnderTest.getSize());

      Object item = new Object();
      Object oldItem = this.listUnderTest.setElementAt(index, item);

      assertFalse(oldItem.equals(this.listUnderTest.getElement(index)));
      assertTrue(item.equals(this.listUnderTest.getElement(index)));
   }
   ...
   @Test
   public void clearListTest(){
      ...
   }
   ...
}
```

Listing 5 Example of parametric test

5 | QUALITATIVE COMPARISON WITH Invite

An interesting study in terms of in vivo testing frameworks would be the empirical comparison between Groucho and Invite, aiming at assessing the respective benefits and drawbacks of the two technologies. Unfortunately, to the best of our knowledge and also according to the feedback received from one of the former contributors to the Invite project, no version of Invite exists that could be actually run within a custom empirical study. Therefore, in this section, we discuss only qualitatively the differences between Invite and Groucho, highlighting their respective technical assumptions, the responsibilities they assign to the AUT software engineering teams (i.e., design, development, and deployment teams), and the associated implications on the life-cycle of the considered AUT. Along such comparison, we refer to the illustrative example introduced in Section 3.

As sketched in Figure 2, the application of Invite involves manual manipulation of the source code of the AUT, to produce an *ad hoc* instrumented version (in the figure, AUT + in vivo (SRC)). With reference to Listing 1, Section 4.1 already described how Groucho instruments the AUT in order to enable in vivo testing capabilities (i.e., see Listing 2). The corresponding capabilities can be achieved within Invite by modifying the AUT code as reported in Listing 6. Specifically, for each method on which in vivo testing will be applied, Invite assumes [30] that the AUT developers or the in vivo testers perform the following steps: (i) they add a method (e.g., `__INVtest_setElementAt`) implementing the decision logic for in vivo testing (see lines 9-12); (ii) they change the name of the method subject to in vivo testing (e.g., `setElementAt` into `__setElementAt` see line 6), and (iii) they replace the original method with an *ad hoc* wrapper that activates the in vivo session after sandboxing the current execution environment (see body of `setElementAt` at line 18). In addition, AUT developers must ensure that the in vivo test methods reside in the same class as the code to be tested or in any super-class along the inheritance hierarchy [12,31].

Clearly, the technical requirements listed above have been imposed in order to ease the access to the internal state of the class under test, but they might severely constrain the way developers can organize their code (e.g., the test code must belong to the class under test; computation of coverage metrics must filter out the contribution from

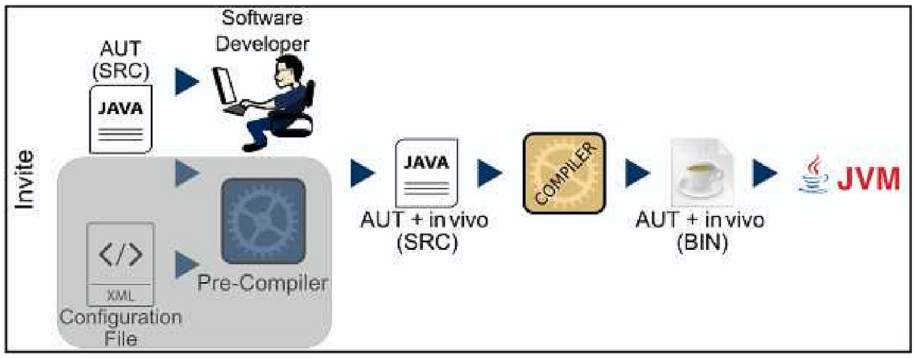

FIGURE 2 Steps required to apply Invite

instrumentation code). In contrast, as discussed in Section 4.2, Groucho includes native features for the automated binaries manipulation of both the AUT, and 3rd party/legacy libraries.

In earlier versions of Invite [31], its authors discussed the possibility to leverage the Singleton design pattern [32] in order to bring more flexibility to the in vivo testing campaigns, as usage of a Singleton would allow removing some of the constraints mentioned above, such as the coexistence of production and test code in the same class. However, even from their own conclusions, such a solution would require a strong design refactoring of the AUT and might not be generally applicable. Invite imposes also precise rules about the names of the tests and of the other methods the instrumentation process injects into the classes of the AUT [12]. Even though these naming rules do not imply any technical limitation, they however require that AUT designers and developers are aware of them and apply them consistently.

```
1  package org.application.utils;
2  ...
3  public class ListOfElements {
4      ...
5      /* Original method */
6      public Object __setElementAt(int index, Object element) {
7          ...
8      }
9      /* Invivo test method */
10     boolean __INVtest_setElementAt(int index, Object element) {
11         ...
12     }
13     /* Logic enabling Invivo test*/
14     boolean __should_run_INVtest_setElementAt(int index, Object element) {
15         ...
16     }
17     /* Wrapper function */
18     public Object setElementAt(int index, Object element) {
19         if (__should_run_INVtest_setElementAt(index, element)) {
20            create_sandbox_and_fork();
21            if (is_test_process()) {
22               if (__INVtest__setElementAt(index, element) == false)
23                  fail();
24               else
25                  succeed();
26               destroy_sandbox();
27               exit();
28            }
29         }
30         return __setElementAt(index, element);
31         }
32         ...
33 }
```

Listing 6 Class instrumented to enable in vivo testing with Invite

As noted by the authors of Invite [12], the instrumentation process and the compilation phase could be further engineered into a single processing step. Indeed, as an alternative solution to the above manual procedures, Invite hints at a

dedicated pre-compiler that could produce an instrumented version of the AUT along the above requirements, following some configuration rules reported in an XML file. This is represented in Figure 2 within a grey-shadowed box, though, because we are not aware of any release of Invite that includes this feature.

In any case, it is clear that with Invite two versions of the same AUT would exist: the plain, original one and the one equipped with in vivo testing features, denoted in Figure 2 as AUT + in vivo (BIN). Thus, both the provider of the AUT and those building solutions/services on it have to be aware that they are dealing with two different products. As noted by the authors [31], any source code modification made to the AUT should take into account the associated implications on the in vivo testability of the application and should be propagated to the in vivo enabled version. Moreover, users of the AUT must take care of installing the *right* version of the AUT in their environment if they want to activate/remove in vivo testing capabilities. Conversely, Groucho can enable in vivo testing capabilities by working directly on Java binary codes (see Section 4.2); thus just one version of the AUT is expected to be released.

For what concerns the sandboxing mechanism, Invite distinguishes between:

(i) the isolation of local data of the AUT stored in-memory
(ii) the preservation of the integrity of the data stored in some local/remote persistence layer or shared with other applications.

As for case (i), Invite offers facilities that can duplicate the whole OS process in which the AUT is deployed. In other words, the isolation of in-memory data is achieved by forking the OS process running the whole JVM. In the general case, this solution could be computationally demanding and could impact the resources available on the machine hosting the AUT execution. Indeed, within the demonstrative example in Listing 1, even though just one class of the AUT is subject to in vivo testing, the activation of each in vivo session requires the creation of a new OS process that will reserve the same memory as if a new instance of AUT were launched. In comparison, Groucho offers a lightweight isolation of individual in-memory objects as reported in Section 4.3. As for case (ii), and similarly to Groucho, Invite leaves to the software engineering teams the responsibility to deal with AUT-specific isolation strategies that could prevent actual changes of shared data.

In summary, both Invite and Groucho foresee use cases where the Test Engineers directly access the source-code of the AUT and modify it to enable in vivo testing. However, the modifications requested by Invite appear more invasive than those required by Groucho. Moreover, Groucho includes a well defined automated solution that does not require access to the source-code and can be applied even when using 3rd party or legacy libraries. The Invite in-memory sandboxing mechanism operates at the level of the OS process, while Groucho adopts a "test & rollback" approach built upon efficient checkpointing technologies. On the other hand, both frameworks do not offer any general solution for isolating data external to the memory of the AUT, leaving such responsibility to the Test Engineers. Finally, Invite does not explicitly provide support for executing multiple test cases starting from the same field execution context, as Groucho does.

6 | OVERHEAD EVALUATION

A key aspect for any platform enabling in vivo testing is to keep its unavoidable overhead as low as possible so as to produce a reduced impact on the actual execution *in-the-field*. Indeed, acceptability of the proposed technology by both the engineering team and the final users of the AUT strongly depends on such a requirement.

The validation of such non functional requirement has to focus on the measurement of the performance overhead introduced by Groucho under various configurations, differing by number of threads executed in parallel and by probability of activating an in vivo testing session. In this section we report the experimental methodology, the results collected while measuring the performance penalties introduced by Groucho and a discussion about when those become imperceptible. This part of Groucho evaluation has been anticipated in our prior preliminary work [13], and is included here for the sake of self-containedness.

6.1 | Methodology

The performance impact of Groucho has two main sources: (1) the overhead introduced by the platform and (2) the cost associated with the execution of each in vivo test case. Evaluation of the latter is application-specific and remains among the responsibilities of the test engineers, who should design in vivo test cases that have minimum execution costs (e.g., comparable to small unit test execution time).

Our research investigates solutions to minimize the first source of performance impact (i.e., the one we can control); indeed such overhead is due to the machinery Groucho implements and it is independent of the specific subject referred as AUT. We then designed the empirical validation to answer the following research questions (RQs):

RQ1 *What amount of overhead does* Groucho *introduce, when varying the frequency of in vivo test execution and the number of parallel threads involved?*
RQ2 *What are the configurations of* Groucho *under which its overhead can be considered small or negligible?*

As subject of the study, we have identified a benchmark application to be executed in the following conditions: first as a plain application running on a JVM and then as an AUT instrumented for Groucho, so that it can be potentially subjected to in vivo testing activities. Specifically, in the former setup no Java Agent was attached to the JVM, while in the latter the same JVM was enabled with instrumentation capabilities, such that the Groucho Agent can be attached to it. Our objective was to measure the execution time of the benchmark in both scenarios, with and without Groucho.

For this performance evaluation we have decided to define a custom Java application as benchmark, instead of reusing an existing one, to have full control on the threads it creates. In fact, multi-threading affects Groucho's performance to a major extent, so it is important to control for this important experimental factor.

In particular, the benchmark application was designed to instantiate multiple threads, each configured randomly, but all configured with the possibility to perform both CPU-intensive tasks and time-consuming activities (e.g., simulating the hang out for IO or remote interactions).[2].

Hence, the two independent variables of our empirical study are (1) the number of active threads and (2) the activation probability of an in vivo testing session.

We conducted two experiments: In Experiment 1, we fixed the activation probability and varied the number of active threads in the benchmark application; in Experiment 2, the role of the independent variables has been exchanged: The number of active threads in the benchmark was fixed, and the activation probability of an in vivo session was gradually changed. As a consequence of the randomized behaviour implemented by the AUT, the estimated execution time for a given pair of independent variable values was measured as the average obtained across multiple runs, all in the same setting.

6.2 | Empirical estimation of the overhead

This section presents the results collected in our experiments and answers our RQs.

6.2.1 | Overhead analysis (answer to RQ1)

In Figure 3, we compare the running times measured when Groucho is either activated (`in vivo`) or not (`no in vivo`) within the JVM hosting the execution of the benchmark AUT. The x-axis displays the number of concurrent threads under execution, while the y-axis reports the running time in seconds. Square-marked lines (blue) depict the execution time in seconds when in vivo testing is enabled, whereas the diamond-marked lines (red) refer to the scenario where in vivo testing is disabled. Each mark (square or diamond) in the solid lines is the average running time observed in 100 executions of the benchmark. When in vivo testing was enabled, the activation probability was fixed to 1%, which means that the annotated method is expected to trigger Groucho only once in 100 invocations. When in vivo testing was disabled the benchmark was running as a plain application on a JVM (this is our baseline for comparison). We changed the degree of parallelism of each execution by varying the number of concurrent threads from 5 to 100 (x-axis). To better visualize the run time trend when the number of active thread is changed, we show also a linear regression line in both scenarios (see the dashed lines).

As expected, the `no in vivo` scenario has an execution time that is not strongly influenced by the number of concurrent threads. The `in vivo` scenario, on the other hand, is characterized by a linear increase of the execution time with the number of threads, and the regression line has a non negligible, positive slope (the coefficient of determination of the linear regression is quite high: 0.79). We conclude that, as expected, the lower bound overhead introduced by in vivo testing with Groucho is linear with respect to the number of active threads in the JVM. Indeed, this is a consequence of the isolation policy on threads adopted in this experiment, which pauses all running threads except the one that is undergoing the in vivo testing session. Such a policy represents the worst case scenario in terms of added overhead. Actually, a real application may not need to pause *all* running threads, since only some of them interfere with the one subjected to in vivo testing.

[2]The benchmark is distributed with the source-code repository of Groucho.

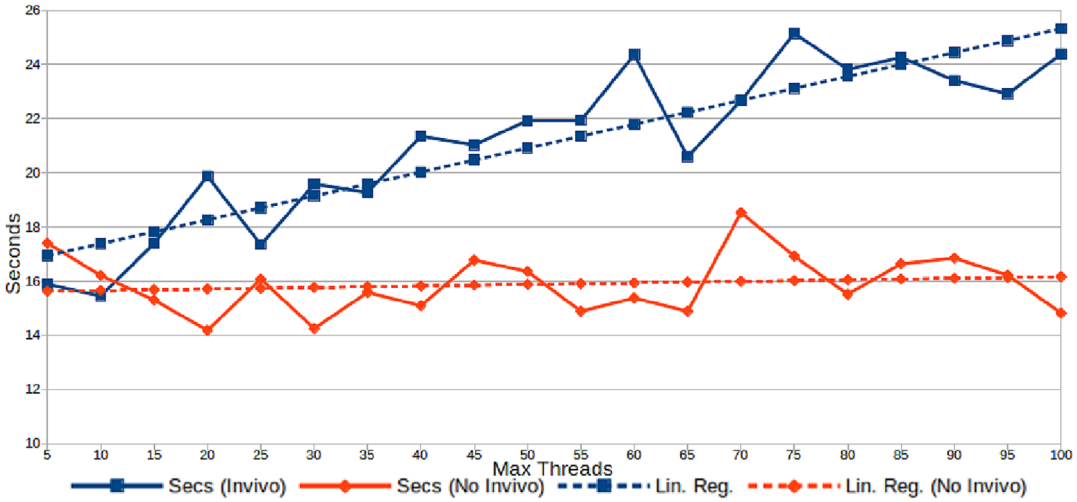

FIGURE 3 Variable Number of Threads and Fixed Activation Probability.

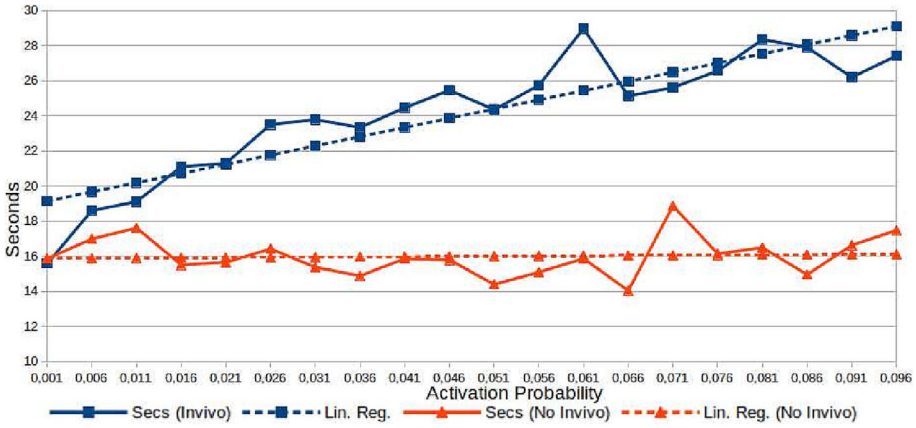

FIGURE 4 Fixed Number of Threads and Variable Activation Probability.

In Figure 4, we compare the execution time measured on the benchmark AUT with number of threads set to 30 and activation probability varied from 0.1% to slightly less than 10% (see x-axis). We also conducted a regression analysis of the execution time with respect to the frequency of in vivo testing. We observed a flat trend for the `no in vivo` scenario, while in the `in vivo` case, there is a positive, non negligible linear relation between the activation probability and the execution time (with high coefficient of determination: 0.78). Thanks to the linearity of the relation between in vivo test execution probability and overhead, test engineers can fine tune the expected overhead, making it acceptable for the end users, by adopting a test activation policy that ensures a test execution probability associated with a negligible overhead. The corresponding number of in vivo test executions in a given time window T will also vary linearly, being roughly equal to $N \times p \times E(T)$, where N is the number of users running the application in parallel; p is the in vivo activation probability; $E(T)$ is the average number of executions of the method under test in a time window of duration T.

Answer to RQ1: The overhead introduced by Groucho grows linearly with the number of threads and with the probability of in vivo test case execution.

This result supports fine tuning of the framework's impact by test engineers, who can reduce the in vivo activation probability until an acceptable overhead is achieved.

6.2.2 | Acceptable configurations (answer to RQ2)

To answer RQ2, we conducted a second experiment, under the assumption that the user base is large (large N in the formula $N \times p \times E(T)$), such that the activation probability (p in the same formula) can be kept extremely small. The goal is to assess the impact of the number of active threads (from 5 to 100) when the in vivo activation probability is low (i.e., 10^{-4}), as well as the impact of variable activation probabilities (from 10^{-5} to 10^{-4}) when the number of threads is set to 30. The results reported for each configuration have been computed as the average over 200 executions of the AUT.

Table 1 and Table 2 report the collected empirical results. The p-value of the Wilcoxon test comparing the distributions of execution times with/without Groucho is above the commonly adopted threshold $\alpha = 0.05$. This suggests that the impact of Groucho is statistically insignificant when the in vivo activation probability is 10^{-4} or lower. Since we cannot rule out the possibility of a Type II error (accepting a wrong null hypothesis), we also measured the Vargha-Delaney effect size (A_{12} measure) [33], which is always small (S) or negligible (N). This means that even if our analysis were subject to a Type II error, the corresponding effect size would be anyway small or negligible, indicating a practically small/negligible impact of the framework.

Answer to RQ2: When the probability of in vivo activation is 10^{-4} or lower, even with 30 threads executing in parallel, the overhead of Groucho is statistically insignificant and practically negligible or small.

This result indicates that applications with a large user base can correspondingly adopt a very low activation probability, making the impact of the framework imperceptible.

TABLE 1 Impact of the number of activated threads—In vivo activation probability: 10^{-4}.

Max threads	Secs (in vivo)	Secs (no in vivo)	Diff (Secs)	Diff (%)	A_{12}
5	16.52	15.93	0.59	3.74	S
10	17.15	14.60	2.55	17.50	N
15	16.31	15.79	0.52	3.32	N
20	16.14	15.79	0.34	2.21	N
25	15.39	16.67	−1.27	−7.62	S
30	15.29	16.63	−1.34	−8.06	S
35	15.44	15.16	0.27	1.84	N
40	17.39	16.56	0.82	5.00	N
45	15.51	14.67	0.83	5.71	S
50	14.58	16.07	−1.49	−9.28	S
55	15.66	15.09	0.56	3.73	N
60	16.58	15.30	1.27	8.32	N
65	16.83	15.76	1.07	6.81	N
70	16.39	16.42	−0.03	−0.22	N
75	15.54	15.97	−0.43	−2.70	S
80	15.83	16.08	−0.24	−1.53	N
85	16.39	17.62	−1.22	−6.93	S
90	15.56	17.27	−1.70	−9.85	S
95	15.45	15.13	0.31	2.11	N
100	15.48	16.39	−0.91	−5.56	N
Average	15.97	15.95	0.02	0.43	

TABLE 2 Impact of in vivo activation probability (10^{-5}…10^{-4})—Max number of threads: 30.

Activation probability	Secs (in vivo)	Secs (no in vivo)	Diff (Secs)	Diff (%)	A_{12}
1E-05	16.00	16.28	−0.28	−1.73	S
6E-05	14.78	16.00	−1.22	−7.66	S
0.00011	14.74	15.44	−0.70	−4.54	S
0.00016	14.43	15.86	−1.43	−9.05	S
0.00021	15.89	16.74	−0.84	−5.06	S
0.00026	16.49	15.62	0.87	5.59	N
0.00031	15.58	16.70	−1.12	−6.71	S
0.00036	15.33	17.33	−2.00	−11.57	S
0.00041	16.44	15.34	1.09	7.16	N
0.00046	16.61	14.711	1.90	12.94	N
0.00051	16.33	15.72	0.60	3.87	N
0.00056	16.42	15.34	1.08	7.07	N
0.00061	15.39	15.06	0.33	2.22	N
0.00066	16.18	15.55	0.63	4.08	N
0.00071	17.32	16.95	0.37	2.20	S
0.00076	16.92	17.65	−0.72	−4.11	S
0.00081	16.61	15.37	1.24	8.08	N
0.00086	16.17	16.70	−0.52	−3.17	S
0.00091	15.54	17.37	−1.82	−10.51	S
0.00096	16.11	16.14	−0.03	−0.19	S
Average	15.96	16.09	−0.13	−0.57	

7 | FUNCTIONAL EVALUATION

In this section we report the results of our study to evaluate the effectiveness of Groucho in terms of failure exposure and statement coverage in comparison to traditional testing in house.

7.1 | Experimental design

The goal of our empirical evaluation is to *assess the effectiveness of the proposed "test & rollback" approach, which we expect to be able to exposing more faults and achieving higher code coverage than traditional testing in house*. Correspondingly, our empirical study aims at answering the following two research questions:

RQ3 *Can in vivo test execution supported by* Groucho *expose failures that go undetected when running the same test suite during in house testing?*

Our first research question addresses the main motivation behind our approach for in vivo testing: detecting faults that are very difficult to expose in the lab, as they require peculiar execution conditions that are not reproduced by stand alone, in house tests, whereas they may occasionally occur in the field. We execute the same test suites in house and in the field on our subject systems, and we monitor the occurrence of failures in the two settings.

Metrics: We collect the number of failures (i.e., exceptions and runtime errors) occurring during in house and in vivo testing. We report the total number of failures observed during in house test suite execution and during in vivo sessions that simulate various field execution conditions of the AUTs). We also report the number of unique faults.

RQ4 *Can in vivo test execution supported by* Groucho *cover instructions or branches that remain uncovered when running the same test suite during in house testing?*

The ability to cover more code during in vivo testing is a key enabler toward fault exposure, since faults residing in uncovered statements go definitely undetected. So another important indicator of the advantages possibly brought

TABLE 3 Number of TCs of the two subject systems, followed by instructions and branches in the CUT.

System	TC	CUT instructions	CUT branches
OSCache	23	3622	442
JCS	3	364	58

by in vivo testing is its capability to cover statements that were not covered during in house testing, thanks to the runtime state of the application under test, which could make some parts of the code reachable upon in vivo test case execution.

Metrics: We measure statement coverage achieved by the test suites available with our subject systems when executed in house. Then we collect coverage information during the in vivo testing session and compare the statements covered in vivo with those covered in house. We report the number of additional statements covered by the available test suite in the in vivo setting.

7.1.1 | Subject systems

In general, planning empirical studies on in vivo testing leads to several constraints on the selection of candidate subjects. For example, the framework used for in vivo testing usually imposes specific constraints on the technological stack. Indeed, Groucho requires that the subjects are exclusively built on the Java technology. When the goal of the experimentation concerns faults exposure, a narrowing selection criterion refers to the availability of documented information about in vivo faults observed for the candidate subjects. Unfortunately, there is no repository of known in vivo failures that could be used for our empirical evaluation and creating such a repository is highly demanding as it involves executing and monitoring possibly many different software systems in their production environment. For this reason, we have run our experiments on two systems that have been used as a benchmark for the Invite approach [12]. Indeed, during in vivo testing of these two systems Invite was able to expose two in vivo faults requiring very specific runtime conditions, which are hardly met during in house testing. We want to observe if Groucho can also expose these two faults during in vivo testing (RQ3).

The characteristics of the test cases used to test the subject systems are shown in Table 3: number of Test Cases (TC) and number of instructions/branches in the Classes Under Test (CUT). It should be noticed that the classes under test are those implementing the caching functionality of both systems and represent a subset of the classes implementing the entire set of functionalities available in the two systems. We have chosen to focus on these classes as they contain the in vivo failures reported by Invite.

OSCache[3] is a system that supports multi-level caching for web pages and web content generated by means of servlets. It represents an interesting subject for in vivo testing because the state of the cache that is exercised during in house testing might differ substantially from the state of the cache that is observed in real execution.

OSCache does not come with any test suite defined by its developers. Hence, we have contacted a colleague of one of the authors, who was not involved in the present research in any way. The contacted person has more than 10 years of industrial experience as software developer, among which seven spent in several companies as a software tester and a senior software tester. She was given the task to develop an extensive set of JUnit test cases for the class LRUCache, covering all relevant corner cases. The result consists of a test suite containing 23 test cases. Notably, Groucho does not lead to any specific constraint on the definition of the test suite. Specifically we asked our colleague to design and develop the test programs for `LRUCache` as in a traditional in house testing context. The only requirement we expressed to her was to formulate parametric implementations for these tests, which is an acceptable and quite common request [29].

JCS (Java Caching System)[4] is an alternative caching system developed by Apache and also supposed to be used with Java servlets within Apache Tomcat. Similarly to OSCache, its runtime state is richer than the in house one, so this is also a good candidate for in vivo testing.

JCS comes with a test suite created by its developers for in house testing. We have reused such test suite after some syntactic adaptation according to the principles described in Section 4.5, to make it suitable for in vivo testing. Such adaptations did not affect in any way the logic of the individual test cases.

[3] https://web.archive.org/web/20101009184819/http://opensymphony.com/oscache/.
[4] https://commons.apache.org/proper/commons-jcs/.

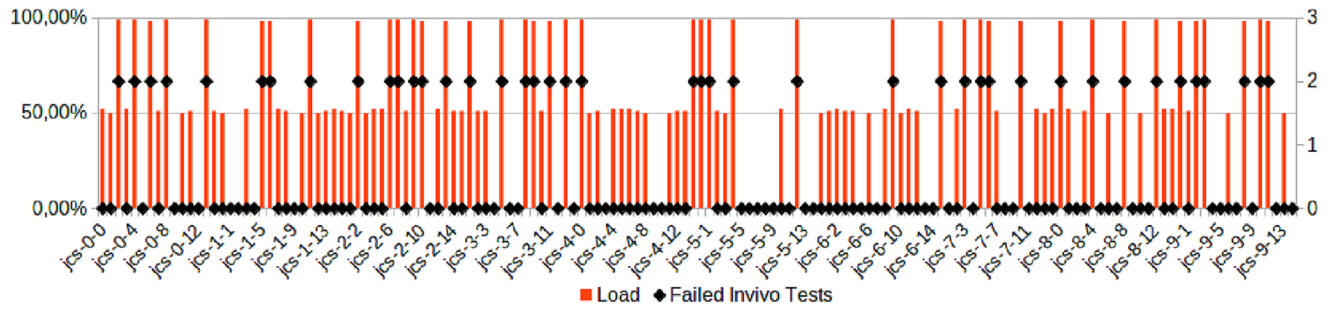

FIGURE 5 Experiment JCS: The *x*-axis displays the identifiers of the in vivo test sessions, the left-side of the *y*-axis shows the load of the cache instance before the in vivo testing session starts, while its right-side shows the number of in vivo tests failed.

7.1.2 | Experimental procedure

We have instrumented the two subject systems to collect coverage information using JaCoCo[5]. Then, we have executed in house the test suites available for the two subjects. In such executions we have collected failure and coverage information, useful for the comparison with in vivo execution.

The in vivo execution followed the instrumentation schema for Groucho discussed in Figure 1b. The two subjects have been executed on a JVM whose arguments were both the Groucho agent, and the JSON configuration file declaring the entry-points for the in vivo testing sessions (see Section 4). Specifically, the two methods annotated with `@TestableInVivo` in the JSON file (one per subject) have been chosen based on the detailed descriptions of the issue reports JCS-16[6] and CACHE-236[7] [12], because we want to test Groucho's capability to expose known in vivo faults.

Whenever the execution invokes any of these methods and the in vivo testing features are enabled, Groucho recognizes the annotation, it suspends the application control flow, and it starts an in vivo testing session. At the end of the session, the application state is rolled back and the main execution is resumed. During in vivo test execution, the failure and coverage data are collected as done for in house testing. To speed up the experimental evaluation, the activation probability of in vivo testing was set to 100%, but the same functional results would be obtained with a lower activation probability, except that the experiment would last substantially longer. In a real setting, where multiple users run the application in parallel, a low activation probability is definitely possible.

The field execution context of the two systems was simulated by means of similar drivers, each one: (i) performing randomized configurations of a fresh cache instance, (ii) causing the invocation of the entry-point methods, so as to activate an in vivo testing session, and (iii) collecting failure/coverage data.

The randomized configurations instantiated by the driver consist of a random uniform selection over three different scenarios for the cache instance: almost empty or empty cache, cache filled around half of its capacity, almost full or full cache. In all three cases, the test driver uses fresh randomic entries values in order to populate the cache instance.

Upon activation of each in vivo testing session, an in vivo test method is invoked according to the configuration reported in the JSON file. The in vivo tests have been coded so as to execute the whole set of test cases considered for each subject (see Section 7.1.1). Furthermore, their implementations leveraged the API provided by Groucho in order to grant for isolation among different test case executions: The execution of each test case starts in the state the cache had just before the activation of the in vivo session. This means that execution is rolled-back also between test cases, not only between in vivo test session and main execution.

The outcome of each in vivo session is expected to depend on the current configuration of the cache instance, which is chosen non-deterministically. Thus, the test driver stimulated the considered subjects in 15 different runs ($\texttt{run}_i$). In addition, each experiment aggregating these runs has been repeated on a fresh execution environment for 10 iterations ($\texttt{it}_j$). Hence, in total 150 in vivo test sessions have been launched against fresh and randomized cache instances for JCS ($\texttt{jcs-it}_j\texttt{-run}_i$), and 150 for OSCache ($\texttt{osc-it}_j\texttt{-run}_i$).

7.2 | Experimental results

In this section, we present the results obtained for RQ3 and RQ4.

[5]See at https://www.jacoco.org.
[6]See at https://issues.apache.org/jira/browse/JCS-16.
[7]See at https://web.archive.org/web/20090214001824/http://jira.opensymphony.com/browse/CACHE-236.

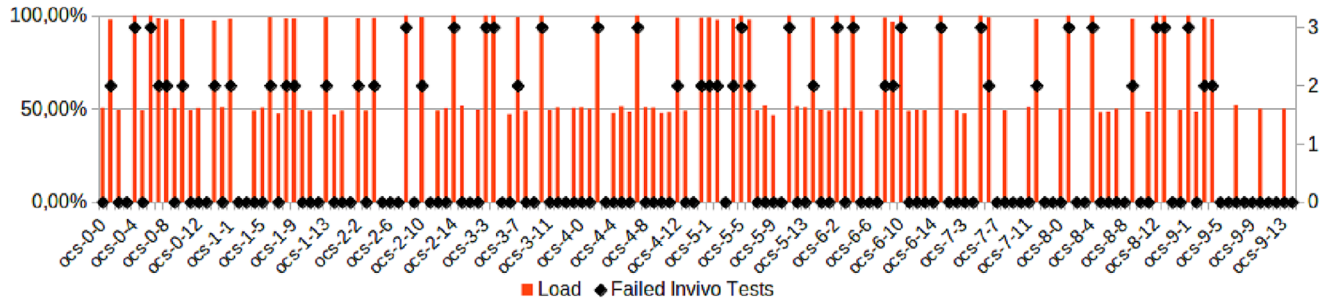

FIGURE 6 Experiment OSC: The *x*-axis displays the identifiers of the in vivo test sessions, the left-side of the *y*-axis shows the load of the cache instance before the in vivo testing session starts, while its right-side shows the number of in vivo tests failed.

7.2.1 | Failure detection (answer to RQ3)

No failure was observed during in house execution of the available test suites. Figure 5 displays the number of failed in vivo tests as well as the cache configurations for each one of the 150 sessions launched for the JCS subject. The *x*-axis displays the session identifiers, whereas the *y*-axis carries two pieces of information: The left vertical axis displays the cache load, varying from 0 to 100%, represented as a (red) vertical bar. The right vertical axis displays the number of failing in vivo tests, varying from 0 to 3 (the maximum number of test cases for JCS), represented as a (black) diamond.

Looking at the individual sessions, the number of failed in vivo tests was always either 0 or 2. The exposed fault when two tests fail is always the same (i.e., 1 unique fault was exposed). This indicates that either the (unique) fault was not triggered or it was triggered by two tests cases from the JCS test suite. As we can see, the JCS failure could be revealed only in sessions where the cache is almost full or full. This was the case for 42 out of the 150 sessions. For any other possible cache configuration, the in vivo tests were not able to trigger the fault.

Figure 6 is analogous to Figure 5 – axes description and interpretation are the same. It displays the number of failed in vivo tests as well as the cache configuration for each of the 150 sessions launched on the OSC subject. Similarly to the observation done for JCS, the OSC fault was also triggered only in sessions where the cache was full or almost full (49 out of the 150 sessions). The main difference observed was that the number of failed in vivo tests varied across the sessions between 2 and 3. This happened because one of the tests requires specific cache content and capacity to fail, which is not necessarily satisfied by all execution contexts with an almost full cache. Similarly to JSC, there is one unique fault exposed by the test cases executed in vivo. So, regardless of the number of failing tests (2 or 3), when the cache is almost full there is one unique fault affecting this system that is always exposed.

Despite the non-determinism associated with the randomized cache configurations, the results on failure exposure obtained on both subjects are completely deterministic: When the cache is full or almost full, the in vivo failure is exposed. Given the deterministic nature of such outcome, statistical analysis of the results was not necessary.

As documented in the issue report, the fault in JCS-16 concerns a check in the `update` method. Specifically, when the maximum capacity is reached the method should not insert the new element in the cache as the overflow of the cache is expected. However, as from the snippet in Listing 7 the guard expression at line 7 (corresponding to line 126 in the original file of the class `LRUMemoryCache`) does not cover the case the cache is actually full (i.e., `size = this.cattr.getMaxObjects()`).

```
1  package org.apache.jcs.engine.memory.lru;
2  ...
3  public class LRUMemoryCache extends AbstractMemoryCache {
4      ...
5      public void update( ICacheElement ce ) throws IOException {
6          ...
7          if ( size < this.cattr.getMaxObjects() ) {
8              return;
9          }
10         ...
11     }
12         ...
13 }
```

Listing 7 **JCS**, the fault detected by the report issue `JCS-16`.

Similarly, the fault reported in CACHE-236 can be triggered by method `remove` (see Listing 8) when the cache instance reaches its maximum capacity: In this state, the guard expression at line 7 (line 1734 in the file that defines `AbstractConcurrentReadCache`) prevents to actually flush all the data in the cache.

```
1  package com.opensymphony.oscache.base.algorithm;
2  ...
3  public abstract class AbstractConcurrentReadCache extends AbstractMap implements Map, Cloneable,
4      Serializable {
5      ...
6      private synchronised Object remove(Object key, boolean invokeAlgorithm){
7          ...
8              if (overflowPersistence && ((size() + 1) >= maxEntries)) {
9                  persistStore(key, oldValue);
10             }
11         ...
12     }
13         ...
14 }
```

Listing 8 **OSCache**: the fault detected by the report issue `CACHE-236`.

Both for **JCS** and **OSCache**, in house testing does not expose the failures as the tests always run against a fresh cache instance. In vivo testing allows to execute the same tests but starting from arbitrary (i.e., randomized) configurations of the cache instances giving them the possibility to explore more scenarios and to expose the failures.

Answer to RQ3: Groucho was able to reproduce the two in vivo faults known from the Invite's benchmark. Reproduction happened deterministically whenever the cache was full or almost full.

7.2.2 | Coverage achieved (answer to RQ4)

The coverage achieved by in house and in vivo testing is reported in Figure 7. Results are grouped by coverage granularity (branch or instruction) and by CUT. Numbers in the horizontal axis represent the classes under test. Group #1 is

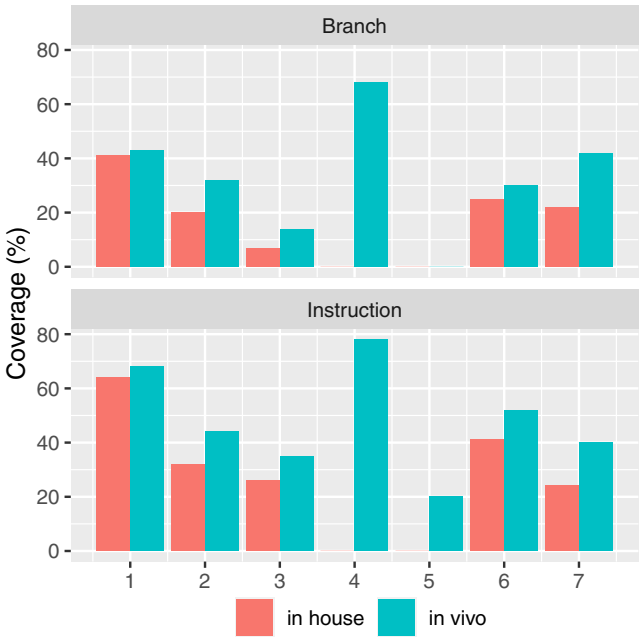

FIGURE 7 Coverage achieved by in house and in vivo testing.

the only CUT for the JCS subject, and groups #2 to #7 are classes from the OSCache subject. The vertical axis displays the coverage, in percentage, achieved by the two testing strategies, in house and in vivo.

While there is non-determinism in the randomized cache configurations, in our experimental setting in vivo test suites are executed always and entirely in each run. As a consequence, the reported coverage increases were observed with no variation in each of the 10 experimental iterations. In fact, the 15 runs executed in each iteration include always at least one instance of full or almost full cache, which is responsible for the coverage increase. Given the deterministic nature of the coverage increase observed across iterations, statistical analysis of the results was not necessary.

Overall, the in vivo testing sessions were able to increase both branch coverage and instruction coverage for both subjects. The in vivo sessions were able to cover two previously-uncovered CUT, namely `4:OSC-AbstractConcurrentReadCache.HashIterator` – 68% and 78% of branch and instruction coverage, respectively – and `5:OSC-AbstractConcurrentReadCache.HashIterator.new.AbstractSet()`. Specifically, both cases refer to local classes (i.e., inner or anonymous classes) whose visibility and usage strictly depend on the implementation of their declaration class. As reported in Section 7.1 the tests for **OSCache** have been developed covering relevant functional corner cases for the class `LRUCache`. As black-box testing strategies do not refer to information on the inner structure of the specific CUT, part of the implementation could remain unexplored. During in house testing, the tests designed for `LRUCache` never activated instances for the local classes labelled as `4:OSC-AbstractConcurrentReadCache.HashIterator` and `5:OSC-AbstractConcurrentReadCache.HashIterator.new.AbstractSet()`, while their in vivo executions reached these classes, extensively for the first class and mostly inside the constructor for the latter. For what concerns the other CUTs, the largest coverage increase was observed for branch coverage of `3:OSC-AbstractConcurrentReadCache.Entry` with a relative improvement of 100% (from 7% to 14% of branches covered). The smallest coverage improvement, on the other hand, was observed for branch coverage of `1:JCS->Shrinker>Thread`, with a relative improvement of $\approx 5\%$ (from 41% to 43%). Running in vivo sessions resulted in an overall average coverage improvement of $\approx 45\%$ when compared with the in house sessions.

Answer to RQ4: By running in vivo test sessions, Groucho was able to increase instruction and branch coverage by a significant amount. This is an important precondition to expose failures that may go undetected during in house testing.

7.3 | Qualitative analysis

We have analysed qualitatively the difference between the in-field test executions that expose the faults affecting our subjects and the offline test executions, which did not expose them. We also contrasted the in-field test executions responsible for coverage increase to the online ones. What we discovered is a confirmation of the usefulness of in-field testing: None of the offline test case was designed to produce the rare condition in which the cache is almost, but not completely, full. On the other hand, in-field execution of the same tests, but on the variety of different execution states observed in the field, was able to occasionally reproduce such rare condition.

8 | THREATS TO VALIDITY

In this section we discuss the threats to validity for the different evaluations we performed.

8.1 | Qualitative comparison with Invite (Section 5)

Threats to content validity concern the appropriateness of those aspects that have been taken in consideration while structuring contents for a validation campaign or a qualitative comparison. The assessment of the content validity is often a non-statistical procedure, which refers to area experts who evaluate the selection of the covered aspects. In this respect, the qualitative comparison discussed in Section 5 has been organised according to the authors' knowledge, which could have affected the comprehensiveness of the discussed aspects. We tried to mitigate such a threat by

collecting all the resources that have been published or discussed about Invite. Furthermore, we also interacted with former contributors of the Invite project, asking for feedback on any available release or technical support. Starting from all the collected information, we critically analysed both differences and similarities with Groucho, considering all relevant software engineering concerns that have to be to addressed when managing or executing the tests in the field [9].

8.2 | Overhead evaluation (Section 6)

With respect to the *external validity*, the more critical aspect refers to the choice of the case-study. Though the benchmark application has been conceived to simulate realistic activities of the SUT (i.e., both CPU-intensive and time-consuming tasks), the realization of such abstractions might not fully reflect the complexity of all the possible scenarios occurring with a real-world application. We tried to mitigate this threat by means of randomness. Specifically we randomly configured: the delays in the rump up during the thread activation, the number and the duration of the CPU-intensive tasks per thread, as well as the number and the duration of the time-consuming tasks per thread. In this respect, by leveraging the number of repetitions we could increase the possibility to explore a wide range of different combinations in the thread life-cycle so as to cover many different contexts while performing in vivo testing of a hypothetical multi-thread application. Yet, we are aware that the choice of the benchmark application might affect the validation of Groucho. As for the *construct validity*, the maximum level of parallelism in both experiments has been bounded to 100 threads. Our decision takes into account the default configurations of the popular multi-thread Java application Apache Tomcat (i.e., default max thread is 200). As it is not realistic to run in vivo testing under critical conditions, we identified the range 5–100 as a valuable *no-stress* scenario for our experimentation. Similar considerations were discussed during the definition of the maximum number of concurrent threads in the experiments with a variable activation probability (i.e., 30). Nevertheless, it is not easy to address these threats on a generic multi-thread application. Thus, possible future work could concern further studies, focusing on a single specific application. Regarding the activation probability in the in vivo scenario, we used small values that work well for applications with a large user base (see Section 6.2.1 and Section 6.2.2).

8.3 | Functional evaluation (Section 7)

The main threat that affect our findings is an *external validity* threat, as we only considered two case study systems. Considering additional subjects would require non negligible effort as we need also to simulate what happens in the production environment and this may require domain knowledge that only the actual software developers possess. The chosen case studies are however part of large size and complex systems, and they have been used as benchmarks in previous studies on in vivo testing techniques [12]. However, we are aware that the current experimentation somehow limits our capability of formulating conclusions with general validity. In Section 9 we refer to future activities we are planning in order to mitigate this threat. For what concerns the *construct validity*, we used consolidated and widely used metrics for fault detection and coverage, but of course different metrics may yield different results. The *conclusion validity* is possibly threatened by the impossibility to run statistical tests, due to the fully deterministic behaviours that we observed. However, the deterministic outcomes of our experiments fully support our answers to the two considered research questions.

9 | CONCLUSIONS AND FUTURE WORK

In vivo testing is a specific kind of field software testing [9] where testing activities are launched directly in the production environment during actual end-user sessions.

Among others, the main intuitive motivation for in vivo testing is that every testing process (even a good one) conducted in house achieves just partial exploration of the scenarios that could occur in the production environment. In other words, it is unfeasible to reproduce all the scenarios that may occur in production, and often the most difficult or costly corner cases may escape the in house testing sessions.

These arguments are especially valid for object-oriented applications where in vivo testing brings an opportunity for tackling the combinatorial explosion of the configurations a state of an object instance can assume in the production environment. However, the application of in vivo testing comes with challenging obstacles such as the isolation between the actual execution context and the in vivo test environment; the isolation across several execution of tests in the same in vivo session; the overhead; the impact the in vivo testing technology has against the software design/development process, its life-cycle, its stakeholders.

In this paper we introduced Groucho, a framework to conduct in vivo testing of Java applications transparently, which does not necessarily require any source code modification nor even source code availability. Groucho adopts a fully automated "test & rollback" strategy that makes it an unobtrusive field testing framework. Ultimately, our approach contributes to further advance in vivo testing towards a scalable and largely automated technique.

The evaluation of Groucho presented in this work covers several aspects, both qualitative and quantitative. For what concerns the qualitative evaluation, we provide a reasoned comparison between Groucho and Invite, a pioneer framework for in vivo testing of Java application. For what concerns the quantitative evaluation, we conceived two studies, in order to assess two major aspects of in vivo testing with Groucho. The former study focused on the overhead that test engineers have to consider when enabling in vivo testing with Groucho. The collected results show that Groucho supports a fine, linear overhead tuning and that under some conditions (i.e., activation probability lower than 10^{-4}), it can be considered irrelevant or imperceptible. The latter study investigates the effectiveness of Groucho in terms of failure exposure. This empirical study was performed against two open source systems (i.e., OSCache and JCS) that have been also used as a benchmark for Invite [12]. The collected results show that our framework can detect failures that would escape in house testing. Moreover, when executed in vivo by Groucho, the available test suites achieved a significant increase of instruction and branch coverage.

As part of our future work, we aim at extending the validation of Groucho on other applications, possibly with additional goals beyond overhead assessment or fault exposure. However, finding suitable subjects with documented in vivo failures is not an easy task. In addition, an immediate follow up will address an important question related to the activation policy, that is, *when should a new in vivo test session be run*? While existing approaches [30] run a new in vivo test session on every previously-unseen application state, we aim for an in vivo test execution policy that balances the testing load across all users of the AUT while at the same time recognizing the occurrence of novel states that deserve in vivo testing. Moreover, we would like to challenge Groucho in real continuous integration and continuous delivery environments, to consider other possible process factors that might impact the efficiency and scalability of our approach. We also plan to extend Groucho in order to collect in vivo execution states that are deemed interesting for developers (e.g., because they trigger a failure), so as to simplify in-house reproduction of test executions performed in the field. More generally, and in line with some other recent activities [34], an interesting future research concerns the study of practical implications entailed by in vivo testing on the software development process. Even though some of the potential adoption costs have been mitigated by our work (i.e., see Section 4), further research would be needed on the qualitative and quantitative analysis of the most prevalent practices followed by software projects that implement in vivo testing.

ACKNOWLEDGEMENTS

This paper has been supported by the Italian MIUR PRIN 2017 Project: SISMA (Contract 201752ENYB), partially by the European Research Council Project 787703: Precrime, and partially by the Italian Research Group: INdAM-GNCS.

DATA AVAILABILITY STATEMENT

The replication package that supports complete reproduction of the results reported in this paper is available from the following Github repository: https://github.com/IASI-SAKS/groucho.

ORCID

Antonia Bertolino https://orcid.org/0000-0001-8749-1356
Guglielmo De Angelis https://orcid.org/0000-0002-1076-0076
Breno Miranda https://orcid.org/0000-0001-9608-9393
Paolo Tonella https://orcid.org/0000-0003-3088-0339

REFERENCES

1. Kohavi R, Longbotham R. Online controlled experiments and a/b testing. Encycl Machine Learn Data Mining. 2017;7(8):922–9.
2. Kevic K, Murphy B, Williams L, Beckmann J. Characterizing experimentation in continuous deployment: A case study on bing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, IEEE, 2017;123–32.
3. Pietrantuono R, Bertolino A, De Angelis G, Miranda B, Russo S. Towards continuous software reliability testing in devops. In *Proc. of the 14th International Workshop on Automation of Software Test, AST@ICSE*. IEEE / ACM, 2019;21–7. https://doi.org/10.1109/AST.2019.00009
4. Planning S. The economic impacts of inadequate infrastructure for software testing. Nat Inst Stand Technol. 2002. https://lara.epfl.ch/w/_media/misc/rti02economicimpactsinadequateinfrastructuresoftwaretesting.pdf
5. Gazzola L, Mariani L, Pastore F, Pezzè M. An exploratory study of field failures. In *Proc. of the 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017;67–77.
6. Rwemalika R, Kintis M, Papadakis M, Le Traon Y, Lorrach P. An industrial study on the differences between pre-release and post-release bugs. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2019;92–102.

7. Kaulgud V, Saxena A, Podder S, Sharma VS, Dinakar C. Shifting testing beyond the deployment boundary. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, 2016;30–3.
8. Capgemini S. World quality report: 2020-2021, 12th edition, 2020. https://www.capgemini.com/research/world-quality-report-wqr-20-21/
9. Bertolino A, Braione P, De Angelis G, et al. A survey of field-based testing techniques. ACM Comput Surv. 2021;54(5):92:1–39.
10. Silva S, Bertolino A, Pelliccione P. Self-adaptive testing in the field: Are we there yet? In *Proc. of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '22*. ACM: New York, NY, USA, 2022;58–69. https://doi.org/10.1145/3524844.3528050
11. Blohowiak A, Basiri A, Hochstein L, Rosenthal C. A platform for automating chaos experiments. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 2016;5–8.
12. Murphy C, Kaiser GE, Vo I, Chu M. Quality assurance of software applications using the in vivo testing approach. In *Proc. of 2nd International Conference on Software Testing Verification and Validation (ICST)*. IEEE Computer Society: Denver, Colorado, USA, 2009;111–20.
13. Bertolino A, De Angelis G, Miranda B, Tonella P. Run Java applications and test them in-vivo meantime. In *Proc. of the International Conference on Software Testing, Verification and Validation – Testing Tools Track (ICST 2020)*. IEEE: Porto, Portugal, 2020;454–9. https://doi.org/10.1109/ICST46399.2020.00061
14. Li PL, Kivett R, Zhan Z, et al. Characterizing the differences between pre-and post-release versions of software. In *2011 33rd International Conference on Software Engineering (ICSE)*, IEEE, 2011;716–25.
15. Elbaum S, Diep M. Profiling deployed software: Assessing strategies and testing opportunities. IEEE Trans Softw Eng. 2005;31(4):312–27.
16. Morán J, Bertolino A, de la Riva C, Tuya J. Towards ex vivo testing of mapreduce applications. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2017;73–80.
17. Tiwari D, Zhang L, Monperrus M, Baudry B. Production monitoring to improve test suites. IEEE Trans Reliab. 2021;71(3):1381–97.
18. Cavalli T, Núñez M. A survey on formal active and passive testing with applications to the cloud. Ann Telecommun. 2015;70(3):85–93. https://doi.org/10.1007/s12243-015-0457-8
19. Lee D, Netravali AN, Sabnani KK, Sugla B, John A. Passive testing and applications to network management. In *Proc. of the International Conference on Network Protocols (ICNP). ICNP*. IEEE CS: Washington, DC, USA, 1997;113. http://dl.acm.org/citation.cfm?id=850935.852443
20. Andrés C, Cambronero ME, Núñez M. Formal passive testing of service-oriented systems. In *Proc. of the 2010 International Conference on Services Computing. SCC '10*. IEEE CS: Washington, DC, USA, 2010;610–13. https://doi.org/10.1109/SCC.2010.62
21. Andrés C, Merayo MG, Núñez M. Formal passive testing of timed systems: Theory and tools. Softw Test Verif Reliab. 2012;22(6):365–405. https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1464
22. Ali M, De Angelis F, Fani D, Bertolino A, De Angelis G, Polini A. An extensible framework for online testing of choreographed services. Computer. 2013;47(2):23–9.
23. Greiler M, Gross H-G, van Deursen A. Evaluation of online testing for services: A case study. In *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, 2010;36–42.
24. Porter A, Yilmaz C, Memon AM, Schmidt DC, Natarajan B. Skoll: A process and infrastructure for distributed continuous quality assurance. IEEE Trans Softw Eng. 2007;33(8):510–25.
25. Basiri A, Behnam N, De Rooij R, et al. Chaos engineering. IEEE Softw. 2016;33(3):35–41.
26. Lahami M, Krichen M, Jmaiel M. Safe and efficient runtime testing framework applied in dynamic and distributed systems. Sci Comput Program. 2016; 122(C):1–28. https://doi.org/10.1016/j.scico.2016.02.002
27. Gonzalez A, Piel E, Gross H. Architecture support for runtime integration and verification of component-based systems of systems. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, 2008;41–8.
28. Bell J, Pina L. CROCHET: Checkpoint and rollback via lightweight heap traversal on stock JVMs. In *Proc. of the European Conference on Object-Oriented Programming. ECOOP 2018*, 2018;17:1–31.
29. Massol V, Husted T. Junit in action. Manning, 2004.
30. Murphy C, Vaughan M, Ilahi W, Kaiser G. Automatic detection of previously-unseen application states for deployment environment testing and analysis. In *Proc. of the 5th Workshop on Automation of Software Test*, ACM, 2010;16–23.
31. Murphy C, Kaiser GE, Chu M. Towards in vivo testing of software applications. In CUCS-038-07, Department of Computer Science, Columbia University, 2007.
32. Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: Elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc.: USA, 1995.
33. Arcuri A, Briand L. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw Test Verif Reliab. 2014;24(3):219–50.
34. Barboni M, Bertolino A, De Angelis G. Insights from running flaky tests into the field: Extended version, 2022. ISTI-2022-TR/007, 2022.

How to cite this article: Bertolino A, De Angelis G, Miranda B, Tonella P. In vivo test and rollback of Java applications as they are. Softw Test Verif Reliab. 2023;e1857. https://doi.org/10.1002/stvr.1857