European Research Council
Established by the European Commission

Self-assessment Oracles for Anticipatory Testing

TECHNICAL REPORT: TR-Precrime-2019-06

Antonia Bertolino[a]*, Guglielmo De Angelis*[b]*, Breno Miranda*[c]*, Paolo Tonella*[d]
[a]*CNR–ISTI,* [b]*CNR–IASI,* [c]*Federal University of Pernambuco,* [d]*USI*

Run Java Applications and Test Them In-Vivo Meantime

Project no.:	787703
Funding scheme:	ERC-2017-ADG
Start date of the project:	January 1, 2019
Duration:	60 months

Technical report num.:	TR-Precrime-2019-06
Date:	September 2019
Organization:	Università della Svizzera italiana
Authors:	Antonia Bertolino[a], Guglielmo De Angelis[b], Breno Miranda[c], Paolo Tonella[d] [a]CNR–ISTI, [b]CNR–IASI, [c]Federal University of Pernambuco, [d]USI
Dissemination level:	Public
Revision:	1.0

Università della Svizzera Italiana (USI)

Principal investigator:	Prof. Paolo Tonella
E-mail:	paolo.tonella@usi.ch
Address:	Via Buffi, 13 – 6900 Lugano – Switzerland
Tel:	+41 58 666 4848
Project website:	https://www.pre-crime.eu/

Abstract

The outcome of test case execution depends on the state of the object under test. While testers can carefully choose meaningful and representative object states for test execution, it is unaffordable to cover the combinatorial space of possible object states exhaustively. An appealing option is to delegate part of the testing activities to the runtime and to execute test cases in the field whenever a new or uncommon state is observed. We have designed and developed Groucho, a framework for in-vivo testing of Java applications. Among the challenges that we faced, the most important ones are isolation of the test session from the user session and minimal performance overhead. Experimental results show that if the activation probability is kept reasonably small (e.g., 10^{-4}), the impact of the framework is imperceptible (i.e., either statistically insignificant or with a negligible effect size).

Contents

1 Introduction

The effectiveness of in-lab, pre-release testing is intrinsically limited when the number of possible states of the applications under test is very large. Actually, this is often the case in object-oriented programming, since the space of the object states is given by the (recursive) combination of all possible attribute values – a combinatorial number. Testers select representative object states, trying to cover all their equivalence classes, under the assumption that a fault is either exposed by any member of a state equivalence class or is not there at all. However, defining such equivalence classes is a difficult task and correspondingly faults may escape in-lab testing and manifest themselves in production, giving raise to *field failures*. Gazzola et al. [5] have performed a wide empirical investigation of field failures. In their study, they observed a substantial number of faults that are reported only by end users, who experienced them in the field. According to the authors' analysis, the dominant cause for field failures is *combinatorial explosion*, i.e., the combinatorial growth of the states to be tested in-house, which inevitably brings to sampling just a (hopefully) representative subset.

In-vivo testing consists of test case execution in the production environment. The term "in-vivo" is borrowed from biological studies, where it refers to tests conducted on a living organism, as opposed to "in-vitro" tests conducted in the laboratory. Introduced in the late 2000s [3, 4], the idea behind adopting in-vivo testing for software applications is that to tackle the combinatorial explosion problem we can take advantage of the enormous number of executions and object states that occur in the field. While a time-limited, in-lab testing session cannot go very far in covering all many possible execution states, a distributed testing session where every user becomes potentially a tester provides a unique opportunity to cover the various use cases at large and to test the *relevant* object states (i.e., those occurring in production and not observed before [8]).

However, from a technical point of view, implementation of in-vivo testing is challenging for various reasons, among which: (1) isolation and (2) overhead. By *isolation* we mean the need to sandbox the execution of test cases, such that the end user session is not corrupted due to test execution. In other words, the testing session should produce no side effect on the user session. This is particularly challenging because we also want to exploit information from the user session, so to run the test cases on states that are possibly different from the ones considered during in-lab testing. The other major challenge is limiting the performance *overhead* introduced by any framework for in-vivo testing. In fact, normal executions must be monitored and in some cases even suspended when new or uncommon object states are observed, but this has to be achieved with minimum impact on the end user's activities.

In this paper, we introduce Groucho, a framework for in-vivo testing of Java applications. To ensure isolation, Groucho adopts a checkpoint/rollback strategy [2], and supports selective threads suspension and resuming.

We have measured 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. Results show that the overhead grows linearly with the number of threads, but it is possible to make it imperceptible (i.e., either statistically insignificant or with a negligible effect size) by reducing the activation probability of in-vivo testing. When the user base is large, even adopting a very low activation probability (10^{-4} or lower) still allows for performing several in-vivo testing sessions that complement in-lab testing, but without interfering with the response time of the application under test.

The paper is organized as follows: Section 2 briefly describes the framework Groucho; Sections 3 and 4 present the experimental methodology and the results collected while measuring the performance impact of the framework; while Section 5 concludes the paper.

2 An In-vivo Testing Framework

The current section presents Groucho, a Java-based framework enabling the activation of a testing session in-vivo, i.e., while a Java application is running in its operational environment. By combining Aspect-Oriented Paradigm (AOP) with lightweight isolation approaches Groucho allows a tester to easily inject tests that, if needed, can be safely executed at run-time.

The high-level architecture of the framework is depicted in Figure 1. Specifically the core part of Groucho is structured around four main building blocks. The `Annotation` part defines the meta-information that can enable or regulate the in-vivo testing activities. `Instrumentation` exploits AOP principles and technology that are leveraged to inject the testing aspects within the production environment. Each injected testing aspect refers to a dedicated layer (i.e., `Callback`) that is responsible for both the orchestration of the

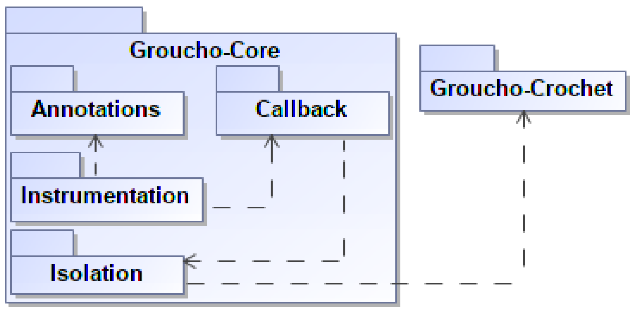

Figure 1: Groucho: High Level Architecture

isolation policies, and the execution of the test cases actually codified as test programs. Groucho structures the common API for the isolation mechanisms within `Isolation`. Specifically, this part also includes the default behaviours foreseen for suspending and resuming threads, while the implementation of the checkpoint/rollback strategy is provided by the module `Groucho-Crochet`.

Groucho is developed and distributed as an open-source project at: `http://saks.iasi.cnr.it/tools/groucho`.

In the rest of this paper the term "test case" refers to a a Java test method that can be activated at run-time.

2.1 Scenario: Roles and Responsibilities

A typical reference scenario for in-vivo testing is depicted in Figure 2. Basically this scenario can be partitioned in two main subsets of activities: one subset (shown in the left thread of Figure 2) includes the activities that take place off-line and are performed by the Test Engineers; the other subset (in the right thread) concerns those activities performed at run-time by the execution environment (i.e., the Java Virtual Machine – JVM).

Allowing for in-vivo testing clearly requires that some assumptions are met in both threads. However, aiming at a broad applicability, Groucho has been designed to keep such assumptions as limited as possible. Concerning the execution environment, even though the presented framework relies on standard off-the-shelf JVMs (e.g., Oracle HotSpot and OpenJDK), it is important to clarify that the solution relies on the Java Agents technology. Agents in Java are implemented by means of the JVM Tool Interface [11] (JVM-TI) as part of the Java Platform Debugger Architecture [9] (JPDA). As a consequence, the system under test (SUT) has to be deployed and launched on a JVM with such instrumentation capabilities enabled.

On their side, the Test Engineers are responsible for the selection of the portions of the SUT that could be subject to in-vivo testing. In addition, they are also requested to properly customise the activation policies for the test cases that will be enabled at run-time. Similarly to [7], the granularity for both the test cases selection, and their activation policies are defined at method level. In this sense Groucho foresees that the Test Engineers are knowledgeable about the responsibilities of the main classes in the SUT and possibly about some details on their implementation. In other words, Test Engineers are partially aware of some internal workings of the SUT.

The first task for the Test Engineers is to annotate the source code of the SUT (① in Figure 2). The objective is to specify which methods will be subject to in-vivo testing and precisely to which test case they are bound. As detailed in Section 2.2, the Groucho framework provides them with a declarative support for such activity. As introduced above, the so annotated version of the SUT must be built (i.e., ②) and launched (i.e., ③) on a JVM properly configured with the Groucho agents.

Without loss of generality, the above scenario assumes that the Test Engineers can access and modify the source code of SUT. For closed-source applications it is still possible to add the Groucho statements (which would correspond to the manual annotations) by directly processing the bytecode before its execution. Such a solution can be achieved by means of off-the-shelf injection technologies [6] and it usually requires a minimal effort in the implementation.

According to the Java Agents technology, each time the JVM loads a new class, it processes on-the-fly the bytecode of the class. The Groucho agents dynamically instrument all the methods previously marked for in-vivo testing (i.e., ④). The instrumentation is responsible for the injection of code that dynamically links the SUT with the Groucho instance running on the JVM.

At run-time, when the JVM fetches any method that is subject to in-vivo testing and just before its execution, Groucho checks whether an in-vivo testing session should be activated (i.e., ⑤, and ⑥). Groucho can bind

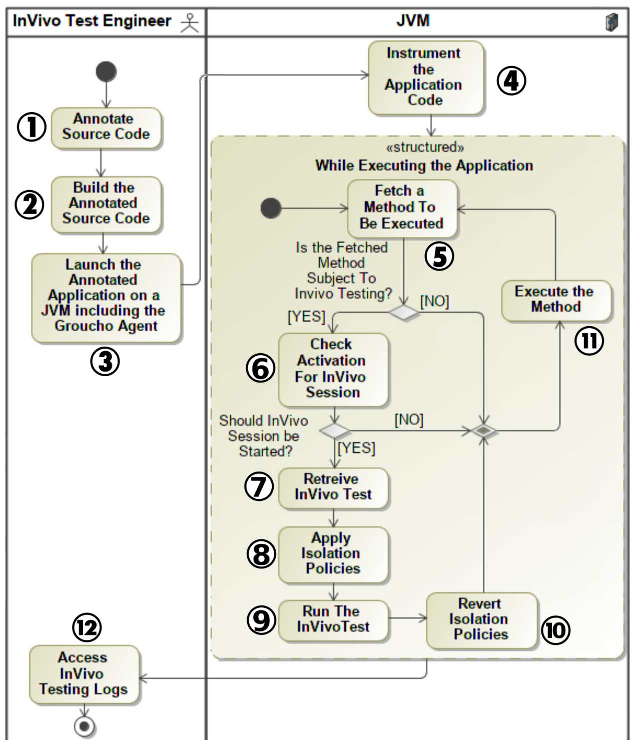
InVivo Test Engineer
JVM
① Annotate Source Code
② Build the Annotated Source Code
③ Launch the Annotated Application on a JVM including the Groucho Agent
④ Instrument the Application Code
«structured» While Executing the Application
⑤ Fetch a Method To Be Executed
Is the Fetched Method Subject To Invivo Testing?
[NO]
[YES]
⑥ Check Activation For InVivo Session
Should InVivo Session be Started?
[NO]
[YES]
⑦ Retreive InVivo Test
⑧ Apply Isolation Policies
⑨ Run The InVivoTest
⑩ Revert Isolation Policies
⑪ Execute the Method
⑫ Access InVivo Testing Logs

Figure 2: In-vivo Testing Scenario

to different kind of strategies governing the activation conditions. A strategy is implemented by means of a dedicated manager. The framework enables any manager implementation to refer the specific execution state reached by the SUT. Test Engineers are responsible for selecting or implementing the most suitable strategy fitting with the considered context. In case of a positive evaluation, Groucho retrieves the right test case (i.e., ⑦) and enforces a set of isolation policies before the actual invocation (i.e., ⑧). The objective of such last step is to avoid that in-vivo testing activities will impact on the results of the other computations not involved in the testing sessions. Section 2.3 discusses the isolation strategies adopted by Groucho.

Once the JVM environment has been isolated, the actual test case can be launched starting from the operational context available at run-time: this consists of the whole pool of Java objects in memory in their current state and with their actual cross-references (i.e., ⑨).

After testing is completed, Groucho reverts the JVM environment to the status available before the validation activities run in-vivo (i.e., ⑩), and reports the resulting verdicts to the Test Engineers (e.g., on some log system). Only at the end of these steps Groucho lets the JVM execute the fetched method (i.e., ⑪). The run-time steps in the scenario at Figure 2 keeps looping for all the time the SUT is active on the JVM.

2.2 Annotations

As described, Test Engineers are responsible to establish the relations between the methods of the classes in the SUT that are subject to in-vivo testing and the test cases that should be executed. Groucho supports such an activity by means of Java Annotations.

```
@TestableInVivo(invivoTestClass = "foo.test.TestClass", invivoTest = "testMethod")
private void methodSubjectToInVivoSession() {
    ...
}
```

Listing 1: Enabling in-vivo Testing with Groucho

In general, annotations in Java are a mechanism enabling the definition of meta-information within a Java class. How such meta-information is applied depends on the elements of the Java language that the annotation targets (e.g., methods, fields, whole classes, etc.); their availability depends on the annotation scope that actually defines when meta-information can be accessed and exploited (i.e., at build time by the compiler, at run-time by the JVM, etc.).

Within Groucho, the annotations target methods. Specifically, any method that has to be considered for the in-vivo testing activities (e.g., `methodSubjectToInVivoSession`) has to be marked with an annotation like the one reported in Listing 1. Specifically, the meta-information required includes the name of the test case to be executed in-vivo (i.e., `invivoTest`) codified as (public) method of a given Java class (i.e., `invivoTestClass`).

2.3 Isolation

The execution of in-vivo testing activities in production environment imposes to carefully avoid the corruption of any user session. This goal is achieved by adopting suitable isolation approaches enforcing the execution of the in-vivo test cases in a sandbox environment. More specifically, it is possible to distinguish them in two major categories: isolation approaches that preserve the state of the users' sessions; and isolation approaches that preserve the consistency of the execution flows in the sessions.

Groucho addresses the former category by adopting a checkpoint/rollback approach [2], which represents a substantial improvement over existing isolation frameworks, such as Invite [7]. In fact, Invite deals with isolation by forking a new, separate process where the testing session is activated. While the memory image of the forked process is a copy of the original memory with no possibility of interference, the operation of process forking is quite expensive, both in terms of the computational resources required to perform it and of the memory it requires. On the contrary, the checkpointing solution in Groucho performs a lazy deep copy of the objects involved in the testing session, leaving all other objects unaffected. The copy follows a lazy strategy as it is postponed until the first new access to an instance that have been previously subject to checkpoint (either directly, or because referred by another checkpointed instance). A similar approach is adopted when restoring of the state of the instances: attributes are reset to the value before the checkpoint at the first new access to the objects after the rollback.

The checkpoint/rollback strategy only ensures that a set of (in-memory) states are safely saved before the execution of an in-vivo test case, and then restored when the test is over. However, in multi-thread applications and while a thread is hosting the in-vivo testing session, the others concurrent threads must not rely on the state of those objects that have been copied at a checkpoint and that will be going to be restored. In fact, the checkpoint/rollback strategy does not force the execution of an application to backtrack, and it may lead concurrent applications to take inconsistent decisions. For this reason, Groucho includes an isolation layer that targets to ensure data consistency in multi-thread applications. Specifically it supports the definition of policies that can enforce the selective suspension and resuming of runnable threads.

Preserving data consistency by acting on the status of the threads is usually a non trivial task. Furthermore, as from its earliest releases, the Java concurrency model deprecated those platform primitives enabling the possibility to force thread suspension and resuming. The reason is that in the general case such possibility lead programmers to adopt inherently deadlock-prone solutions. Currently multi-threading in Java has been designed on top of a simplified concurrency model. In this sense, the recommended approach [10] invites software engineers to design reliable solutions that are explicitly tailored for each considered scenario. Among the others, the objects signalling API is the most minimal harness provided by the Java platform in order to architect synchronisation among threads. The consequence for robust solutions is that suspending a thread is not an atomic activity, and in some cases it is not always possible (e.g., in order to prevent scenarios leading to deadlock).

Groucho considers the selective suspension and resuming of threads a specific aspect to be injected in the SUT. As a consequence, during the code instrumentation phase (see ④ in Figure 2) each class of the application is equipped with dedicated features enabling the threads signalling mechanism that is exploited during the execution of the in-vivo testing sessions.

3 Validation Methodology

An important concern about Groucho is acceptability for both the engineering team and the final users of the SUT. Any platform that enables in-vivo testing, as Groucho, should impact in-field execution as little as possible, in terms of introduced overhead. In a real, operative scenario, the impact of Groucho originates from two 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 impact 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) . In general, it is test engineers who are in charge of properly mitigating all the risks of an in-vivo testing campaign. Hence, in our empirical validation of Groucho we have focused on the first concern and we have answered the following research questions (RQs):

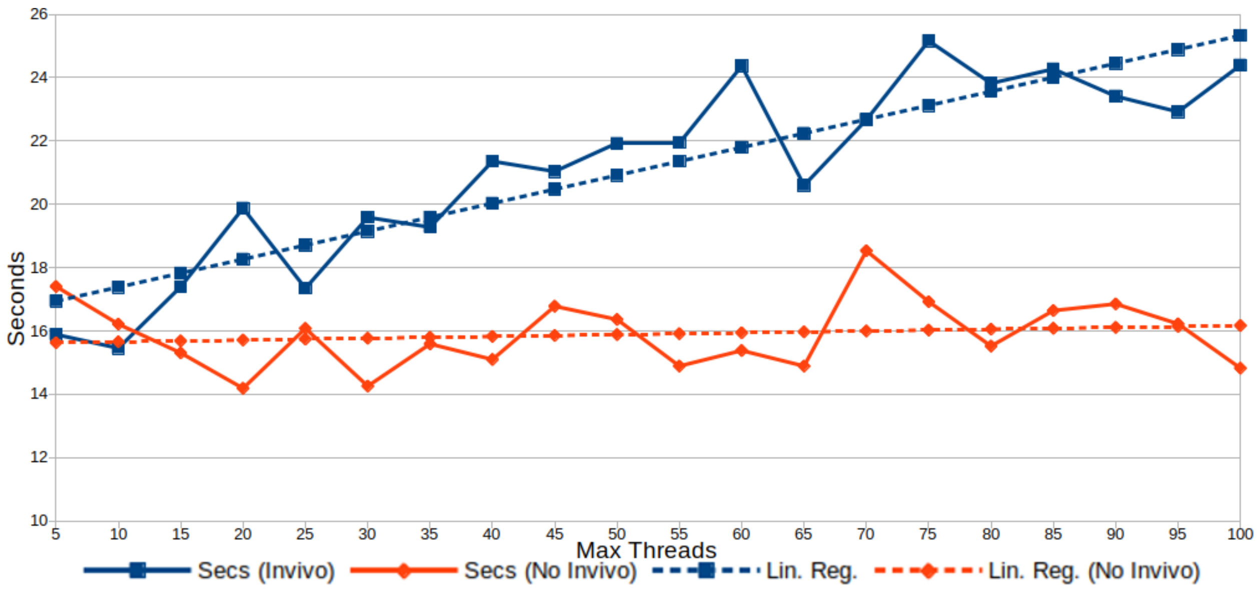

Figure 3: Variable Number of Threads and Fixed Activation Probability

RQ1: What amount of overhead does Groucho introduce, when varying the frequency of in-vivo test execution and the number of parallel threads involved?

RQ1: What are the configurations of Groucho under which its overhead can be considered small or negligible?

For the validation, we have identified a benchmark application to be exercised under the following conditions: first as a plain application running on a JVM, and then as a SUT instrumented for Groucho, so that it can be potentially subjected to in-vivo testing activities. Specifically, in the former setup no Java Agents were attached to the JVM, while in the latter the same JVM has been enabled with instrumentation capabilities. The objective is to measure and compare the execution time of the benchmark in both scenarios: with and without Groucho.

In this study, we defined a custom Java application to be used as benchmark. We have decided to create a custom application rather than reusing an existing one to have full control on the threads it creates. In fact, multi-threading is a major issue for Groucho and the degree of multi-threading is known to be a major source of overhead.

Specifically, the benchmark application has been 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).[1].

Given the design of the benchmark application, the two independent variables in our empirical validation are: number of active threads and activation probability of an in-vivo testing session.

We planned for two experiments. In the first experiment we fixed the activation probability and varied the number of active threads in the benchmark application. In the second experiment 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 the benchmark application implements a randomised behaviour, the estimated execution time for a given assignment of both independent variables has been measured as the average over multiple runs in the same setting.

4 Empirical Results

In this section, we analyse the results collected in the experiments and answer the RQs.

4.1 Answering RQ1

Figure 3 reports a comparison of the measured running time when Groucho is either activated (`in-vivo`) or not (`no in-vivo`) within the JVM hosting the execution of the benchmark. The x-axis displays the

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

number of concurrent threads under execution, while the y-axis reports the running time in seconds. The square-marked lines (blue) depict the benchmark performance 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 after 100 executions of the benchmark. For the scenario with in-vivo testing enabled, the activation probability was fixed to 1%, which means that the annotated method is expected to trigger Groucho only once in 100 invocations. For the scenario with in-vivo testing disabled the benchmark was running as a plain application on a JVM (this is our baseline for comparison). To observe the impact associated with different degrees of parallelism, we varied the number of concurrent threads from 5 to 100 (x-axis). In order to compare the trends of running time when varying the number of thread activated, the linear regression for both scenarios is also displayed in the plot in the form of dashed lines.

In the `no in-vivo` scenario the running time does not seem to be strongly influenced by the number of concurrent threads. This result is in line with our expectation since we did not expect the JVM to present a degradation in performance under the investigated conditions. In the `in-vivo` scenario, on the other hand, the regression analysis revealed a linear relation between the number of threads used and the execution time of the benchmark (the coefficient of determination is quite high: $RQ = 0.79899891$ for Lin. Reg. in Figure 3). This means that the lower bound overhead introduced by in-vivo testing with Groucho is linear with respect to the number of active threads in the JVM. This result is in line with our expectations. In fact, the isolation policy on threads, adopted in the experiment, imposes that Groucho pauses all the running threads except the one that is undergoing the in-vivo session. Such a policy is the worst case scenario in terms of added overhead. In fact, a real application may not need to pause *all* running threads, since only some of them may be possibly interfering with the one subjected to in-vivo testing.

Figure 4 reports a comparison of the running time measured on the benchmark when the number of concurrent threads is fixed to 30 and the activation probability is variable. The various activation probabilities explored (from 0.1% to slightly less than 10%) are displayed in the x-axis; all the other elements in the plot are analogous to those of Figure 3. Also in this case, a regression analysis was performed on the execution time over the frequency of in-vivo testing. As expected, the `no in-vivo` case results in a flat trend. In the `in-vivo` case, on the other hand, there is a linear relation between the activation probability and the execution time (with high coefficient of determination $RQ = 0.78963026$ for Lin. Reg. in Figure 4). Linearity of the relation between in-vivo testing probability and overhead provides test engineers a powerful and fine grained mechanism to fine tune the expected overhead such that it is acceptable for the end users. In fact, test engineers can reduce the activation probability of in-vivo testing until the associated overhead becomes negligible. 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.

4.2 Answering RQ2

To further investigate the linearity of the overhead with respect to both number of threads and in-vivo activation probability, we conducted additional experiments 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. Specifically, we want to understand: (i) the impact of the number of active threads in the benchmark (from 5 to 100) when the activation probability for an in-vivo testing session is rare (i.e. 10^{-4}); (ii) the impact of variations of low activation probabilities (i.e. from 10^{-5} to 10^{-4}) when the maximum numbers of threads is fixed to 30. Both for (i) and (ii), the results in each configuration have been computed as the average over 200 executions of the benchmark.

A summary of the empirical results collected for both (i) and (ii) is reported in Table 1 and Table 2, respectively. The p-value of the Wilcoxon test comparing the two distributions of execution times is always large (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) [1], 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.

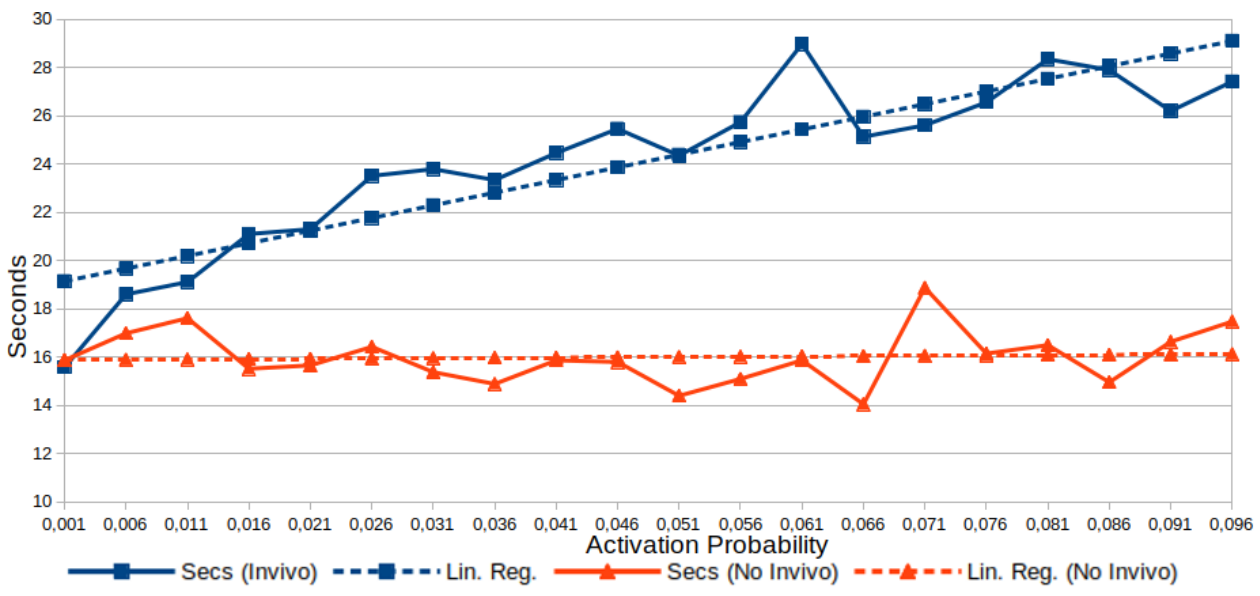

Figure 4: Fixed Number of Threads and Variable Activation Probability

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 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}
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	

Table 2: Impact of In-vivo Activation Probability ($10^{-5} \ldots 10^{-4}$) – Max Numbers of Threads: 30

4.3 Summary

We answer as follows to the RQs that motivated the study: **RQ1:** The overhead introduced by Groucho grows linearly with the number of threads and with the probability of in-vivo test case execution; **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.

The first answer 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. The second answer indicates that applications with a large user base can correspondingly adopt a very low activation probability, making the impact of the framework imperceptible.

5 Conclusions and Future Work

In-vivo testing is an appealing approach to software testing that consists of launching testing sessions in the production environment during end-user sessions. Within the context of object-oriented applications, it represents an opportunity for tackling the combinatorial explosion of the cases to be tested by leveraging the enormous number of executions and the actual object states that occur in the field. Such a possibility could be very effective in order to reveal unknown corner cases, or bugs that are very unlike to manifest themselves in the lab testing sessions. However, the application of in-vivo testing approaches comes with challenging obstacles from a technical point of view, such as isolation and overhead.

This work presents Groucho, an open-source framework that enables in-vivo testing for Java application. Specifically, the paper introduces the high-level architecture of Groucho; it describes the roles foreseen in the in-vivo testing scenario with their responsibilities; and it discusses the main technical features provided by the framework.

A first empirical validation of Groucho has been planned and performed. The objectives of the study were to understand the overhead that test engineers have to consider when enabling in-vivo testing with Groucho and under which conditions such overhead can be considered irrelevant. The collected results show that Groucho supports a fine, linear overhead tuning. In addition, if the activation probability is kept reasonably small (e.g., 10^{-4}), the impact of the framework is imperceptible.

The current version of Groucho targets in-vivo testing of an in-memory collection of object states. Also the current isolation policies avoid suspending a thread if it is performing some blocking activities. Future work will investigate how to extend the checkpoint/rollback strategy so as to cover broader scenarios that are not limited to activities within a single JVM. Also, as future work we would like to support smarter isolation policies that would allow for full recovering of a thread even if it is interrupted during a blocking operation.

References

[1] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[2] Jonathan Bell and Luis Pina. Crochet: Checkpoint and rollback via lightweight heap traversal on stock JVMs. In *Proc. of the ECOOP*, 2018.

[3] Antonia Bertolino and Andrea Polini. The audition framework for testing web services interoperability. In *Proc. of SEAA*, pages 134–142. IEEE, 2005.

[4] Matt Chu, Christian Murphy, and Gail Kaiser. Distributed in vivo testing of software applications. In *Proc. of ICST*, pages 509–512. IEEE, 2008.

[5] Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. An exploratory study of field failures. In *Proc. of the 28th ISSRE*, pages 67–77, 2017.

[6] Anjana Gosain and Ganga Sharma. A survey of dynamic program analysis techniques and tools. In *Proc of the 3rd FICTA, International Conference*, pages 113–122, Cham, 2015. Springer.

[7] Christian Murphy, Gail E. Kaiser, Ian Vo, and Matt Chu. Quality assurance of software applications using the in vivo testing approach. In *Proc. of 2^nd ICST*, pages 111–120. IEEE-CS, April 2009.

[8] Christian Murphy, Moses Vaughan, Waseem Ilahi, and Gail Kaiser. Automatic detection of previously-unseen application states for deployment environment testing and analysis. In *Proc. of the 5th AST Workshop*, pages 16–23. ACM, 2010.

[9] Oracle. *Java Platform Debugger Architecture (JPDA)*. `https://docs.oracle.com/javase/8/docs/technotes/guides/jpda` accessed on 2019-07.

[10] Oracle. *Java Thread Primitive Deprecation*. `https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html` accessed on 2019-07.

[11] Oracle. *Java Virtual Machine Tool Interface (JVM-TI)*. `https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti` accessed on 2019-07.