SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Department of

Informatica Sistemistica e Comunicazione

PhD program: Computer Science Cycle XXXI

FIELD TESTING OF SOFTWARE APPLICATIONS

Luca Gazzola

Registration number: 769189

Tutor: Alberto Leporati

Supervisor: Leonardo Mariani

Coordinator: Stefania Bandini

ACADEMIC YEAR 2018/2019

Abstract

When interacting with their software systems, users may have to deal with problems like crashes, failures, and program instability. Faulty software running in the field is not only the consequence of ineffective in-house verification and validation techniques, but it is also due to the complexity and diversity of the interactions between an application and its environment. Many of these interactions can be hardly predicted at testing time, and even when they could be predicted, often there are so many cases to be tested that they cannot be all feasibly addressed before the software is released.

Field testing aims to tackle the problem of applications failing in the field by moving the testing phase directly in the field environment. This makes it possible to exploit different scenarios that would otherwise be difficult to capture with in-house testing.

In this Ph.D. thesis we explore the area of software field testing, we present a study that characterizes the problem of applications failing in the field, a client-server architecture that can be exploited to organize and control the field testing process and a testing approach that exploits the field itself as testbed for running the test cases. The presented approach is empirically evaluated on a popular dataset of software faults demonstrating that 35% of the faults that were not discovered in-house could have been revealed with field testing.

Contents

1 Introduction **13**

2 The Field Testing Process **17**

2.1 Background 17

2.1.1 Failure Context 18

2.2 In-house Testing and its Limits 19

2.3 The Vision of a Field Testing Process 21

2.3.1 Challenges 21

3 Field Faults Studies and Field Testing Techniques **25**

3.1 Studies About Software Faults 25

3.1.1 Fault Types Distribution and Localization 25

3.1.2 Fault Distribution Across Code 26

3.1.3 Fault Root Cause 27

3.1.4 Fault-Failure Relationship 27

3.1.5 Estimation of remaining faults after testing 28

3.1.6 Discussion 28

3.2 Field Testing 28

3.2.1 In-vivo Testing 29

3.2.2 Ex-vivo Testing 33

4 A Study About Field Failures **35**

4.1 Subjects 36

4.2 Experimental Procedure 36

4.3 Results 40

4.4 Threats to Validity 51

4.5 Findings 51

4.6 Discussion 53

5 An Architecture for Field Testing **55**
5.1 Overview . . . 55
5.2 Client-side Components . . . 56
5.3 Server-side Components . . . 57
5.4 Components Interaction . . . 57
5.5 Implementation of the Architecture . . . 62
5.6 Discussion . . . 64

6 An Approach to Field Testing **67**
6.1 Aspect-Oriented Programming . . . 67
6.2 Moving Test Cases to the Field . . . 69
6.2.1 General Idea . . . 69
6.2.2 Field Testing Structure . . . 70
6.2.3 How to Write Field Tests . . . 71
6.3 Discussion . . . 75

7 Empirical Evaluation **77**
7.1 Subjects of the Study . . . 77
7.2 Experimental Procedure . . . 78
7.3 Results . . . 79
7.4 Qualitative Analysis . . . 81
7.4.1 Null Pointer Failure . . . 82
7.4.2 Value Failure . . . 85
7.5 Threats to Validity . . . 88

8 Conclusion **89**

List of Figures

2.1 General production environment 18

4.1 Reasons why software applications fail in the field 45
4.2 Field elements involved in Field Failures 48
4.3 Field Failure Types 48
4.4 Field Failure Detectability 50
4.5 Number of steps required to trigger a field failure 50

5.1 High-level overview of the framework 56
5.2 Sequence diagram illustrating how the framework components interact when discovering new configurations 58
5.3 Sequence diagram illustrating how the framework components interact when testing a new configuration 60
5.4 Sequence diagram illustrating how the framework components interact when testing a new configuration 61
5.5 Sequence diagram illustrating how the framework components interact when querying the server for changes in the model and configurations 63

6.1 Our field testing approach 70

7.1 failure types observed 79

List of Tables

3.1 In-vivo techniques attributes 30

4.1 Failure Types 37
4.2 Failure Detectability 39

7.1 Failure and Fault types revealed 80

Listings

6.1 Simple method that divides `a` by `b` 68
6.2 Join-point for the calls to `divide(double a, double b)` 68
6.3 Advice that checks the divisor value, and throws exception if it is zero 68
6.4 pointcut 72
6.5 advice that captures calls to `DatasetUtilities.iterateDomainBounds` and runs the corresponding test case 72
6.6 pseudo-code for a field test case 73
6.7 test case for DatasetUtilities.iterateDomainBounds using the captured object 73
6.8 `DatasetUtilitiesTest.testFindDomainBounds()` 74
6.9 `DatasetUtilitiesTest.createXYDataset1()` 74
6.10 `XYSeriesTests.testAddOrUpdate3()` 75
6.11 `XYSeriesTests.testAddOrUpdate3()` with relaxed assertions 75
7.1 `DatasetUtilitiesTests.testBug2849731_2()` 82
7.2 `DatasetUtilitiesTests.testFindDomainBounds()` 82
7.3 `DatasetUtilitiesTests.createXYDataset1()` 83
7.4 `IterateDomainBoundsTester` 84
7.5 `DatasetUtilitiesTestField.testFindDomainBounds()` 84
7.6 `UsageJFree2.trigger()` 85
7.7 `StringUtilsEqualsIndexOfTest.testEquals()` 85
7.8 `StringUtilsEqualsIndexOfTest.testEqualsOnStrings()` 86
7.9 `StringUtilsTestField.testCustomCharSequence()` 86
7.10 `UsageLang14.trigger()` 86
7.11 `StringUtilsTester` 87

Chapter 1

Introduction

Achieving high quality is mandatory in modern software applications, but, despite intensive in-house testing sessions, organizations still struggle with releasing dependable software. Faulty applications running in the field are the source of several problems, including higher maintenance costs, reduced customers satisfaction, and ultimately loss of reputation and profits.

Studying the reason why several faults are not detected with in-house testing is of crucial importance to mitigate their occurrence in the field. Although the faults present in the field could be the result of a poor in-house testing process, there are many faults that are objectively hard to detect in house, even using state of the art methodologies and techniques. This intuition has been confirmed by an analysis that we performed on multiple real software failures experienced in the field by the end-users. Our analysis shows that a significant proportion of the faults that cause field failures have specific characteristics that make them extremely hard to be detected in house. In particular, out of all the failures that we analyzed, approximately only 30% of them could be attributed to bad testing practices, while the remaining 70% should be attributed to an interaction between the software under test and the field that is objectively difficult or even impossible to test in-house.

The problem of software failing in the field can be tackled by trying to improve the in-house testing phase or by moving part of the testing infrastructure to the field, exploiting it as testbed for running the many test cases that cannot be executed in house, because of both the limited resources available for testing and the challenge of recreating in house the same environment that is available in the field. There are several important benefits related to the capability of exploiting the field as part of the testing process. If the application to be tested is popular enough to be installed in many devices and computers (e.g., hundreds or thousands), these many instances would offer unique opportunities in terms of range of situations and configurations that could be tested in parallel. Moreover, regardless of the number of installations of a same application that are available, testing an application in the field gives the unique opportunity of testing the software when used in its real environment, with real data. Finally, test cases could be executed perpetually, not only to test an application that has been just installed, but also to test the software while the environment, the users, and the data evolve. Successfully deploying the capability of executing test cases while the software is running in the field, namely the capability of doing field testing, even enables the possibility to identify potential failures before they are experienced by the end-users.

Testing applications in the field raises new issues that need to be addressed. Applications that support field testing need a properly designed test infrastructure that runs field tests systematically, according to some criteria; the tests must not cause any side effects, or degrade the performance of the software under test. The oracle problem, already present when testing software in-house, is exacerbated when testing software in the field due to lack of control over the test inputs and thus the impossibility to precisely assert the outcome of the test cases. These problems make the development of an effective and efficient approach extremely challenging.

In this Ph.D. thesis we present a client-server architecture that can be used to deploy and orchestrate testing capabilities in the field. The client-server approach allows for the distribution of field testing capabilities on multiple instances of the same application, and introduces the possibility to timely identify interesting field configurations that are worth testing. The client component, deployed with the application, recognizes when new field configurations are observed, and activates the field testing process accordingly. The server component keeps the global knowledge about the configurations that have been tested and provides traditional in-house testing capabilities when field testing cannot be performed.

We also present an approach to write test cases that can be executed in the field. Field test cases take inputs from the field instead of creating new ones, and exploit them to validate the software with stimuli not tested before. We empirically evaluated the proposed approach on 91 real faults that were revealed in the field, and assessed that field tests could have revealed 32 (35%) of these faults, indicating an interesting complementarity with respect to traditional in-house testing.

The contributions of this Ph.D. thesis are: (1) a study that characterizes field faults and consequent field failures, (2) a client-server architecture that allows to deploy and orchestrate testing capabilities in the field and (3) an approach to write test cases that can be executed in the field.

Chapter 2

The Field Testing Process

In this chapter we introduce and discuss the topic of field testing. In Section 2.1 we introduce the terminology that we use throughout the thesis, in Section 2.2 we explain the limits of an in-house testing process and Section 2.3 presents the challenges involved in designing a field testing approach.

2.1 Background

In this Section, we introduce the key concepts that are relevant in our work.

Software Fault: A *software fault* is a flaw in a computer application that may cause the application itself to behave unintendedly.

Software Failure: A *software failure* is a malfunction that makes an application behavior deviate from its specification.

Test Input: *Test input* is used to stimulate a software application to observe its behavior during testing.

Oracle: An *oracle* specifies how a software application must behave given a specific *test input*.

Field Failure: A *field failure* is a software failure experienced in a production environment.

Field Fault: A *field fault* is a fault that is present in a software program deployed and running in a production environment.

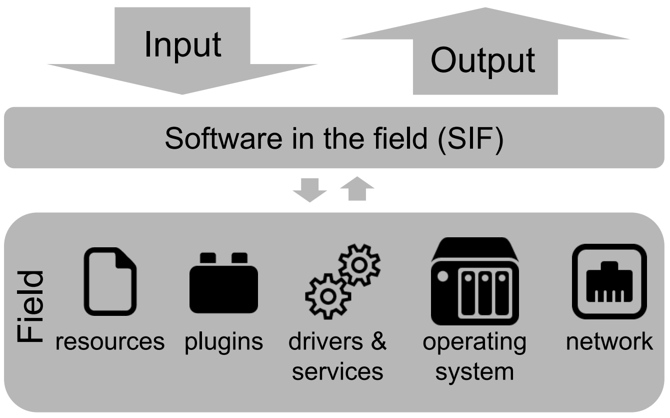

Figure 2.1: *General production environment*

Field faults may or may not cause field failures, depending on the execution conditions in the field.

In-house Faults and Failures: The term *in-house* refers to the development environment. *In-house failures* indicate failures that occur when testing the software system in the development environment and *in-house faults* indicate the causes of the failures exposed during testing.

Distinguishing between faults that survive the testing due to inadequate quality processes and faults that are inherently hard to detect is important to devise verification and validation strategies. We capture the different nature of faults with the new concept of field-intrinsic fault.

Field-intrinsic Fault: A *field-intrinsic fault* is a field fault that is inherently hard to detect in-house, either because impossible to activate in-house or because depending on *unknown* or *uncountable* many conditions.

2.1.1 Failure Context

Figure 2.1 shows the different elements that comprise a production environment and that play a key role in field-intrinsic faults. The *software in the field* (*SIF*) represents a software application or a software system running in the field. The SIF receives inputs and produces outputs. The SIF receives the *inputs* in the form of data and stimuli from both the SIF users and other systems interacting with the SIF. The SIF *outputs* might be either visualized for the users or dispatched to other systems. While executing, the SIF may interact with a *field* that includes several entities *that are not under the control of the SIF*: multiple types of *resources*, such as files and databases, which might be accessed by the SIF during computations, and third-party components that provide services to the SIF, such as *plugins* that extend the capabilities of the SIF with additional features, *drivers & services* that provide a range of services to the SIF, and the *operating system* that defines the basic runtime environment of the SIF. Finally, the SIF may communicate with other applications and services using the *network*.

The role of the environment in field failures leads to the concept of *failure context*:

Failure Context: A *failure context* is the execution context of a failure, that is, the specific state that the elements in the field must have to trigger the failure. For instance, being the operating system part of the SIF failure context, a failure might only trigger when the SIF runs on a specific operating system version.

2.2 In-house Testing and its Limits

Software testing [51] is the most common process used to assess the quality of software, it involves stimulating software applications with input data and observing how the application reacts, checking that the behavior is the expected one. Testing is typically performed as part of the software development process, before the application is released. We call this activity *in-house testing*. We use this terminology to differentiate it from *field testing*, which is done after the application is released in its operational environment (the *field*). In-house testing has its advantages over field testing: mainly developers and tester have a greater control over the application and they do not need to worry of any interaction of the testing process with the regular application usage. However there are also disadvantages:

- *Unknown scenarios*: modern testing techniques allow the simulation of a great deal of different use cases for software applications, but there still are scenarios that are unknown or impossible to test inhouse. It might be impossible to test a specific scenario either because the cost would be too high, or because the testing would rely on the occurrence of unpredictable natural events. An example of the first case is the testing of a mobile communication software on a full fledged mobile network. If the test objective is assessing the performance of the network under heavy load for example, the mobile operator cannot risk deteriorating the network performance, hence it will rely on simulations. In the second case think of an emergency broadcast in case of a earthquake, an event impossible (and not advisable) to replicate on a large scale. It is reasonable to assume that the testing process of an application also consists of observing how the application works in differently configured environments. However regardless of the thoroughness of this testing phase, there will always be configurations that are *unknown* for the application at a certain time (think, for example, of new device hardware or operating systems released after the application)

- *Combinatorial explosion of application parameters*: previously we pointed out that there might be configurations that are unknown at the testing time, but even the known configurations number might be too great to extensively test (think of all the possible combinations of operating system versions, drivers, specific hardware and so on).

If we look at the two main disadvantages of in-house testing that we pointed out, we can also understand how the field testing can be exploited to complement the in-house testing phase, mitigating such problems:

- *Unknown scenarios*: a field testing infrastructure could potentially make it possible to test irreproducible or hard to simulate scenarios as they happen, without the need to re-create them in an in-house environment. Also, field testing might allow to timely observe how the application reacts to previously unknown configuration (e.g., recently released hardware or software that interacts with the application itself).
- *Combinatorial explosion of configurations*: if software applications are released with a field testing infrastructure, the natural heterogeneity of the field can be exploited to test the application of many different configurations, without having to simulate them in-house. This can be greatly beneficial, for example in the context of mobile

android application: it has been calculated that as of 2015 there were more than 24 thousand different android devices [1], it is clearly unfeasible to test and application on all of them exhaustively.

2.3 The Vision of a Field Testing Process

In a field testing process the end-user environment could be exploited as testbed for running the many test cases that cannot be executed in house, because of both the limited resources available for testing and the challenges we explained above.

The field testing process offers the advantage of testing software in its target environment, with real data. Test cases could be executed perpetually to check how changes in the field impact on the application under test, possibly identifying potential failures before they are experienced by the end-users. Unfortunately, this field testing process brings forth a number of challenges that we explore in the next section.

2.3.1 Challenges

Field testing is inherently different from traditional in-house testing and poses a new set of challenges, that can be divided into two groups: the ones about the *test strategy*, that is, when and how to test the software in the field, and the ones about the *non-intrusiveness* of the testing process, that is, how to run the test cases without affecting the user data and user processes, with negligible impact on the user experience.

Test Strategy

We identified four key elements that should be part of a well-defined test strategy.

Test obligation. The test obligation defines which behavior should be tested in the field. Since field testing is executed after in-house testing, field testing should focus on complementary aspects, that is, on those cases that have not been tested in house, or, because of the combinatorial explosion problem, have not been tested thoroughly. Moreover, field testing should target those functionalities that cannot be effectively and efficiently tested in house, such as, functionalities that depend on field entities.

Test opportunity. The test opportunity defines when the software application running in the field should be tested. When testing is performed in house, the testing procedures are usually started in accordance with the development activities (e.g., a change in the source code). The activation of testing procedures for software running in the field should be based on different aspects. In particular we identified three potential testing opportunities. The first one is the application under test being in a specific state, which could be recognized as relevant for testing. Another opportunity is a change in software configuration: applications can be usually configured with a high number of parameters, a new combination of parameters observed can be the trigger for a field testing procedure. Similar to the previous opportunity is the change in the environment configuration: applications can typically run in different environments, which can be configured in different ways, a new environment configuration can hence be an interesting opportunity to test the application.

Lastly, we have to address the problem of available resources, which should be sufficient to run the test cases without impacting the user experience.

Test generation strategy. The test generation strategy defines how to obtain the test cases that should be executed in the field. We foresee two main classes of strategies: static and dynamic. Static test generation strategies consist of implementing and generating in house the test suites specifically designed to cover potentially interesting situations in the field. In this case the test suite would not change, unless new test cases are supplied from the developers, and the test cases that must be executed would be selected dynamically based on the state of the execution. Dynamic test generation strategies should instead generate or update test cases directly in the field. This can be done automatically, for example by mutating the available test cases or even producing new tests from scratch according to specific criteria. Dynamic test generation in the field can however be problematic because of potential overhead issues, and because the test obligation cannot be defined as clearly as with the static approach.

Test oracle. The test oracle [20] defines what the expected behavior of the software under test is. This information is necessary to detect failures. Differently from in-house testing, when performing field testing there are a number of factors, such as the environment configuration or the input values, that cannot be foreseen, hence the test oracles must be general enough to include different scenarios, but also precise enough to actually detect failures.

Non-intrusiveness

We identified two key elements to take into account when designing a non-intrusive field testing approach.

Isolation. A software application running in the field interacts with the elements that are available within its environment (e.g., the resources). Isolating field testing requires the guarantee that the application under test does not produce any disruptive effect while tested, that is, it cannot cause any loss of data and any impact on the other processes running in the field must not be noticeable by the user. This is a fundamental property to make the field testing technology acceptable by end users.

Performance overhead. Running test cases in the field requires the consumption of additional resources, definitely CPU cycles and memory, but also I/O and access to the network. The field testing procedure must guarantee that any additional resource consumption is rarely and hardly noticeable by the users of the application.

Discussion

In this section we explored the main aspects and the challenges that field testing poses. In this Ph.D. thesis we define a general framework that enables field testing of software applications, in particular, in Chapter 5, we define an architecture that uses a client-server structure to gather different application and field configurations and test them whenever a new one is observed. In Chapter 6 we introduce a way to generate static test cases that can be executed in the field using input from the field itself, and we explain how to write generalized oracles to be used in such test cases. The proposed field testing approach is general can be applied to different software architectures and domains.

Chapter 3

Field Faults Studies and Field Testing Techniques

This section discusses contributions related to this Ph.D. thesis, which can be organized into two main groups: studies about the characteristics of software faults, especially the ones detected in the field (presented in Section 3.1) and approaches to test the software in the field (presented in Section 3.2).

3.1 Studies About Software Faults

The problem of studying software faults has been tackled by different angles, namely the distribution of different fault types in the different software development stages, the distribution of faults across the application components, the root cause of faults and the relationship between faults and failures. We present representative work for each category in the next sections.

3.1.1 Fault Types Distribution and Localization

Hamill et al. [34] try to understand the relationship between fault and failures by focusing on two aspects: the location of faults and the fault types. Through a rigorous analysis performed on two real-world case studies, the authors investigated (1) if failures are triggered by faults belonging to the same unit (file, component, etc.) or spread across different units; and (2) if some types of faults are more common than others. With respect to fault locations, authors reported that about 60% of the failures were triggered by faults spread in two or more files; as for fault types, the conclusion was that the majority of fault types was due to either coding mistakes (~30%), requirements problems (~30%) and data problems (~15%). The authors put particular emphasis on the fact that the results are consistent across the two case studies they analyzed and the related studies as well.

Fan et al [28] present a study that aims to show the distribution of different types of faults in the context of nuclear power plants software. The authors make use of two models to perform the analysis: the Failure Mode and Effect Analysis (FMEA) model, which takes into account both the software development and software operation phases, is used to extract information related to the fault origination stage; while the Three-Frame Mode model is used to analyze how the faults cause inconsistencies between the control software and the hardware components of the power plant (sensors, actuators, etc.). The study presents statistical data about the development phases in which the faults were originated, the percentage of fault types, and about the failures that were triggered as a result. The authors conclude that good software engineering practices can drastically reduce the introduction of faults during software maintenance, in particular simple design and decoupling of functions.

3.1.2 Fault Distribution Across Code

Ostrand et al. [50] present a study performed on multiple releases (and multiple lifecycle stages within single releases) of a large industrial system that aims to understand if particularly fault-prone files exist. The authors confirmed the well known Pareto distribution of faults (about 80% of the faults can be found in 20% of the code), but also found out that, as the software evolves, the faults become increasingly concentrated in increasingly smaller proportions of the code. The authors also tried to understand if newly written source code files were more or less fault-prone than files written in older releases, finding that indeed new files are more likely to contain faults.

3.1.3 Fault Root Cause

Leszak et al. [44] present a root cause analysis performed on a transmission system, a peculiarity of this study is that the authors include the possibility of having more than a root cause for each fault: they consider three possible root cause areas (human, project and system lifecycle) and assign a value to each of them, with respect to the studied fault. The results of this analysis show that the majority of faults are found in the design and coding phase, the cost of fixing bugs increases linearly with each development phase, and the major problem for software quality is domain and system knowledge.

3.1.4 Fault-Failure Relationship

Hamill et al. [35] study the relation between fault types, failure detection, and failure severity. This study is performed on NASA software that is installed on satellites and specifically targets safety-critical faults. The aim of the analysis in particular is to understand how the fault type influences the likelihood of a failure to be triggered, and also to understand how do different activities influence the type of faults discovered. The study shows that nearly half of the safety-critical failures were due to coding faults, the second most likely cause (~25%) were requirement problems. As for the failure-detecting activities, the authors found out that about half of the reported failures were discovered during code analysis activities, and about (~40%) of the remaining ones were detected during testing.

Yuan et al. [56] investigated user-reported failures in distributed software applications aiming to discover the relationship between these failures and one or more underlying faults. They found that generating test inputs that trigger the failures is not trivial, since multiple input with definite order are needed. The authors also claim that state-of-the-art testing techniques do not work well on distributed systems because of large size of the input and state space. To tackle this problem they propose an approach that scans Java bytecode and, when it identifies an instruction that can throw an exception, it checks the corresponding `catch` block for possible issues. The authors show that using their approach they can trigger most of the critical failures.

3.1.5 Estimation of remaining faults after testing

Roper et al. [54] compares different estimators of residual faults that use capture-recapture models, with the aim of determining the accuracy of their estimates. The authors use the total number of faults that two independent testers found in multiple datasets as the baseline, and apply the estimators after different stages of testing, obtaining a predicted number of faults which can be checked against the total. The results show that the estimators can be used to make reasonably accurate estimates of remaining faults, although the accuracy tends to vary between datasets, and based on the coverage of the used test suites.

3.1.6 Discussion

Despite the growing interest in field faults and the design of approaches to address different kinds of faults, there is still no study on the nature of field problems that indicates whether they can be better addressed inhouse by improving testing techniques and methodologies, or in the field by exploiting the many instances of a same application running within several heterogeneous environments. None of the presented studies has focused on the causes of field failures and the reasons why failures have not been discovered at testing time, as well as the common characteristics of field failures, which is the starting point of our research. We aim to provide an initial characterization of field faults and the consequent failures in the field, in particular our study differs from previous research it introduces a set of characteristics that make faults hardly detectable in-house and discusses the factors that make these failures likely observable only in the field.

3.2 Field Testing

In this section we discuss research that tackles the problem of testing software applications in the field. This section is divided in two parts: Section 3.2.1 includes research work about in-vivo testing. In Section 3.2.2 we surveyed ex-vivo testing, different from field testing because it gathers data from the field and uses it to test applications in-house.

3.2.1 In-vivo Testing

In-vivo testing techniques aim to test the application while it is being used for normal operation. Different attributes are relevant when discussing in-vivo testing techniques, starting from the ones shared with in-house testing (e.g., oracle, granularity), to attributes that are specific to tests executed in the field (e.g., isolation, trigger). It follows an overview of the attributes we used to classify in-vivo testing techniques.

- *Objective*: defines what the testing technique wants to achieve (e.g., revealing faults, assessment of security)
- *Target*: defines the aim of the testing process (e.g., regression, updated functionality)
- *Granularity*: defines the level at which the testing is performed (e.g., unit, integration, system).
- *Trigger*: defines the event that starts the testing process.
- *Oracle*: defines the expected result of the testing process.
- *Isolation*: defines how the technique avoids side effects in the field environment.

It follows the description of the most representative work in the area of in-vivo testing. Table 3.1 reports how the discussed techniques approach the challenges presented by in-vivo testing.

In-vivo Testing Techniques

Murphy et al. [46] present an in-vivo testing approach which consists of deploying unit test cases that are executed with a user-defined probability while the application is running. The test granularity level is the single unit, with test code directly written in the source code of the application; the test code is triggered at specific entry points (usually method calls) with random probability. The test oracles are specified by the developers (typically they check invariants that must hold true after the test is executed) and the test cases do not cause any side effect by construction.

Configuration fuzzing [27] is an in-vivo testing approach which targets security faults: it mutates the configuration of a software application at

Table 3.1: *In-vivo techniques attributes*

Technique	**Objective**	**Target**	**Granularity**	**Trigger**	**Oracle**	**Isolation**
Murphy et al. [46]	revealing faults	regression	unit	execution of specific functions	user written invariants	granted by test case construction
Configuration fuzzing [27]	assessment of security	regression	system	execution of specific functions	user written invariants	forked process
StealthTest [23]	revealing faults	regression	unit	randomly at specific launch points	user written invariants	achieved with transactional memory
Goh et al. [31]	revealing faults	change of a functionality	unit	new version of a functionality deployed	user written invariants	lightweight virtual machine
Greiler et al. [33]	revealing faults	change of a functionality	integration	dynamic system reconfiguration	integration properties	testing mode (unpublished service)
King et al. [40]	revealing faults	change of a functionality	integration	component replacement	integration properties	stubbed services
Sammodi et al. [55]	assessment of performance	regression	unit	specific service calls	performance thresholds	guaranteed by the testing process

runtime and checks for the violation of security invariants. It makes use of the field to observe and mutate application configurations that might not have been conceived during the in-house testing phase. When the function that needs to be tested is called, the system uses the `fork` system call [52] to spawn another process, mutates the configuration and executes the original function, that is eventually checked for security invariants. Isolation is granted by the `fork` system call (that is, if the test case does not manipulate external resources).

StealthTest [23] presents a technique that performs system tests of C programs using transactional memory [42]. The technique tackles the problem of high performance overhead that results from using the fork-based testing techniques. This approach uses transactional memory: a mechanism that allows blocks of code to be executed atomically (either execute in its entirety or not at all). Using transactional memory, changes introduced by the execution of the test code can be rolled back to prevent side effects, hence guaranteeing isolation. As for the previous techniques, the tests are triggered when a specific piece of code is executed and the test oracle is the specified as invariant properties that must hold true.

Another approach to test isolation in the field is the one used by Goh et al. [31]: it makes use of the MONO CLI's application domain to isolate the execution of program code and the ownership of resources, allowing to confine the software under test in a simil-virtual machine.

Greiler et al. [33] present a technique that tests a service-oriented application whenever it undergoes a reconfiguration. As a first step, the new service is deployed in parallel to the old version, without the possibility to be called by other services (unpublished). A service test suite is then executed, checking proper communication with other services. If all the tests pass, the new service can safely publish its functionalities. The testable version of the service is built in a way that does not let it cause side effects when it is being tested, hence providing isolation. This technique targets integration faults and the test oracle is based on expectations about service communication constraints. The technique objective is to reveal integration faults but also assess non functional requirements (service response timing).

King et al. [40] specifically target autonomic software, hence the testing capabilities are integrated in the autonomic manager, which also functions as a test oracle by checking test results against system policies. As for the previous technique the focus of the technique is system reconfigurations, and the tests work at integration level. After a service is tested this technique also performs dependency analysis to identify the services that call it, and tests them as well. The isolation of the tested service from the environment is achieved by stubbing the other services.

Sammodi et al. [55] also present a technique that aims to test service oriented application in the field, with the goal of assessing the quality of the tested services (performance and availability). In this technique monitoring and testing are intertwined: the monitoring data gathered from normal use of the application is used to select which test cases to run and the test case results are used to adapt the monitoring strategies. The technique assumes no side effects from the testing process (stateless services) and tests at unit level (single services)

In the context of in-vivo testing, runtime testability [32] is a metric that aims to estimate the capability of a given software application to be tested at runtime: the metric takes into account which type of tests can be performed during runtime without affecting the running system, considering the characteristics of the system itself and the extra infrastructure needed to run test cases in isolation.

Discussion

From the surveyed techniques we can see that the problem of in-vivo testing has already been tackled in the past, multiple approaches have been proposed, and we can clearly see, from the domains that the various techniques operate in, that in-vivo testing is a cross-domain challenge and opportunity. From our analysis of the area we observed that in-vivo testing techniques generally lack a proper exploitation of field elements. Testing performed in the field should focus on revealing faults that are hard to detect during traditional in-house testing processes, to do this an in-vivo testing process should focus on actively checking the environment for new configurations, as well as the application for previously unseen states, and use this information to assess how the application works. In our work we gather important elements of the field and dynamically modify and execute test cases at runtime with the aim of improving the testing scenarios covered by standard test suites.

3.2.2 Ex-vivo Testing

Ex-vivo testing techniques use data gathered in the field by monitoring the application and use it to improve in-house testing for future releases. Ex-vivo testing has the advantage of not needing to build a testing infrastructure to deploy with the application in the field. Ex-vivo testing is particularly useful when testing data-intensive applications (e.g., embedded systems) where a great amount of data that the application handles must be simulated according to specific criteria. This simulation step can indeed be avoided by collecting data from the field and process it with the application. Moran et al. [45] present a technique that aims to ex-vivo test MapReduce applications, this framework uses production data to run the processing step in-house, and checks that different infrastructure configurations produce the same result. Another domain where ex-vivo testing is used is autonomous veichles: Neves et al. [48, 47] use logs that contain field data as input to a search-based algorithm that generates new input to test the application in-house. Work on ex-vivo testing has also interested the domain of Dynamic Software Product Lines (DSPL), Hansel et al. [36] present a technique that works by obtaining information from multiple systems derived from a DSPL, and adjusts the tests to properly cover the product line configuration space.

Discussion

Ex-vivo testing techniques can be useful when testing data-intensive applications, and, in general, whenever simulating meaningful field input is difficult. Also, if an application cannot be tested in a production environment (due to performance issues or other constraints), ex-vivo testing could be a viable option. The testing process is performed in-house, hence potential limitations typical of field testing do not apply to ex-vivo testing. Two of the discussed ex-vivo testing techniques are indeed used in data-intensive applications where testing in the field might be problematic, either for performance (map-reduce applications) or safety reasons (autonomous vehicles). The drawbacks of using ex-vivo testing (in particular compared to in-vivo testing) come from the fact that the field environment must be simulated when it is meaningful for the test that must be performed, and this might influence the test result, depending on how good the simulation is. Also, when testing specific application states, the application must be first brought in such states, a task that is often non-trivial and costly.

Chapter 4

A Study About Field Failures

Field testing aims to detect faults in the application while it runs in its deployment environment. To better understand the problem of applications failing in the field, we performed and presented a study that aims to understand the reasons why applications still fail after they are tested and deployed [30].

Field Failures can bring consequences on users and organizations, such as customer dissatisfaction, economic losses and legal issues. They are caused by faults that escape the in-house testing activities and are not detected and repaired before the software is released in the field. We denote such faults as *field faults*.

Field failures may depend on weak testing activities and poor development practices. However, they may also derive from factors that prevent the failures to be detected and the corresponding faults to be removed before the software is released, such as when the conditions that trigger the failure are impossible to reproduce in the testing environment and when the number of combinations to be executed goes beyond any reasonable limit.

Field faults that cannot be detected with in-house testing approaches might be more easily addressable in the field where the diversity and complexity of the execution environment could be exploited in the verification activity, for instance the great number of different android devices makes it hard to test an application on all of the devices in-house, but field testing capabilities can address this problem.

The study we design provides an initial characterization of field faults and the consequent failures in the field: we introduce a set of characteristics that make faults hardly detectable in-house, we study the characteristics of failures reported by the users from three ecosystems, and we discuss the factors that make these failures likely observable only in the field.

4.1 Subjects

We selected a set of desktop and web applications that are available with the source code, are widely adopted and are thus good representatives of well used applications, and give access to publicly available bug reports, which are needed to study bug reports submitted by end-users.

We thus selected multiple applications from three ecosystems:

- **Eclipse** and in particular its well-known and widely used plugins: the *Subversive* SVN client for Eclipse [8], the *EGit* Eclipse Team provider for Git [6] and the *EclipseLink* plugin for developing code using the Java persistence API [7]. The bug reports are accessible on the Eclipse Bugzilla bug tracking system [5].
- **OpenOffice** is one of the most popular open source office applications [3]. The bug reports are accessible on the Apache OpenOffice Bugzilla bug tracking system [4].
- **Nuxeo** is a Web-based content management system used to develop many popular Web sites [11]. The Nuxeo issue tracking is Jira [10].

4.2 Experimental Procedure

For our analysis, we identified as faults the bugs labeled as *confirmed*, *verified* or *resolved*, and we inspected all the bugs reported for the three Eclipse plugins from January 2015 to December 2015 for a total of 412 analyzed bug reports, and all the bugs reported for both OpenOffice and Nuxeo from September 1st 2016 to October 1st 2016, for a total of 99 and 56 bug reports inspected, respectively. For each bug report, we inspected the information about the failure, the inputs, the execution conditions and the failure impact. We discarded the bug reports containing only a memory dump and

Table 4.1: *Failure Types*

	Value Failures
Invalid value	The SIF produces an incorrect output value, although still in the domain of the output variable
Out of domain	The SIF produces a value outside the domain of the output variable
Error message	The SIF produces an error message
	Timing Failures
Early timing	The SIF produces an output too early, for instance before an expected waiting time
Late timing	The SIF produces a value after a required deadline, defined either explicitly or implicitly
Omission	The SIF never produces an output value in an asynchronous computation
	System Failures
Halting failure	The SIF never produces any output value in a synchronous computation
Crash	The SIF crashes and no services is delivered
Unstable behavior	The SIF shows an erratic behavior without receiving any input, for instance, a flashing blacklight in a smartphone

a stack trace, which might be useful for developers, but are not useful for the purpose of our investigation, and studied in detail a total of 119 bug reports: 63 for Eclipse, 26 for OpenOffice, and 30 for Nuxeo.

RQ1: Why software applications fail in the field?

We investigated why faults have not been revealed in house but have been detected only in the field by examining the conditions that caused the failures to identify the factors that contribute to the failures and are extremely hard to be tested in-house. We labeled each fault as *bad-testing*, if we could not find any of such factors, *field-intrinsic* otherwise. We identified four categories of *field-intrinsic* faults that we discuss in the next section, where we characterize the identified classes of faults.

RQ2: Which elements of the field are involved in field failures?

For each bug report, we identified the elements of the field shown in Figure 2.1 that play an essential role in the failure.

RQ3: What kind of failures can be observed in the field?

We studied the characteristics of field failures to identify their attributes and classify them. Better understanding the nature of field failures is essential for developing techniques for testing applications in the field without uncontrolled side effects. Some types of failures might be easier to detect and control than others. For example, exception and error messages are easy to detect and usually do not cause loss of data because the application itself detects and handles these erroneous situations; system crashes are also easy to detect, but may cause loss of user data; incorrect results may be hard to detect, and may silently compromise the user data and the overall computation.

We carefully analyzed the failure taxonomies proposed by Bondavali and Simoncini [24], Aysan et al. [19], Avizienis et al. [18], Chillarege et al. [25], and Cinque et al. [26] to identify the candidate attributes for field failures, and exhaustively inspected the bug reports in our data set to identify the most relevant attributes for characterising field failures: *failure type* and *detectability*.

Failure Type The failure type characterizes a failure according to the way it appears to an observer external to the system.

We identified three possible categories of failure types, *value*, *timing* and *system* failures, and we further detailed each type in three subtypes, for a total of nine failure types, which we use in the next sections to categorize bug reports, and which are summarised in Table 4.1.

- **Value failures** occur when the SIF produces incorrect outputs: an *invalid value*, a *value out of domain* or an *error message*. For example in a functionality that returns the ZIP code of a city, a value failure of type *invalid value* occurs when the SIF returns the ZIP code associated with a city different than the input one, a *value failure* of type *out of domain* occurs when the SIF returns a malformed ZIP code, a *value failure* of type *error message* occurs when the SIF returns a message the reports an internal error that prevented retrieving a ZIP code.

- **Timing failures** occur when the SIF produces some outputs at a wrong time: too early (*early timing*), too late (*late timing*) or never (*omission*).

- **System failures** occur when the SIF is blocked (*halting failure*) has stopped running (*crash*) or does not respond reliably to the input stim-

Table 4.2: *Failure Detectability*

	User	SIF
Signaled	$\surd$	$\surd$
Unhandled	$\surd$	$\times$
Silent	$\times$	$\times$
Self-healed	$\times$	$\surd$

Legend: $\surd$ *detected by,* $\times$ *NOT detected by*

uli (*unstable behavior*).

Detectability The detectability attribute characterizes the difficulty of detecting the failure.

We distinguish four levels of detectability, *signaled*, *unhandled*, *silent* and *self-healed*, based on both the ability of the system to detect the failure and an external observer to observe a misbehavior without specific system knowledge, as summarized in Table 4.2.

- **Signaled failure**: a failure that the system detects and reports. A simple example of a signaled failure is an application that opens a popup window to inform the user that the application will be unexpectedly closed because of a memory problem;

- **Unhandled failure**: a failure that the system does not handle and that leads to a crash. The system does not detect the failure, while the user trivially detects the uncontrolled crash of the application without requiring any knowledge about the application;

- **Silent failure**: a failure that the system does not detect letting the application continue operating without producing any signal that a user can recognize as a failure without prior knowledge about the application. A simple example of silent failure is a flight simulator that simulates the flight conditions imprecisely and that a user cannot detect without a specific knowledge of the flight simulation system.

- **Self-healed failure**: a failure that the system detects and overcomes transparently to the user. The user continues using the application without noticing any problem. Self-healed failures are common in systems exploiting redundancy to mask failures, such as Hadoop [9].

RQ4: How many steps are required to trigger a field failure?

For each failure, we identified a sequence of steps that are needed to cause the failure, aiming to, but not necessarily proved to be, a minimal sequence. For the interactive subjects, we identify steps with GUI actions like opening windows, entering data in some fields, clicking on menus and buttons.

We counted the steps that lead to a failure by considering the sequence of operations described in the bug reports submitted by users. When creating a bug report, users intuitively identify a critical state that may lead to the failure and submit both the information about the critical status, typically described in a declarative way, and a sequence of operations that lead to a system failure from the critical state, typically described in an operational way. For example in the Open Office bug report #126930, the state to trigger the failure is characterized by the availability of a certain file, and the steps to reproduce the failure consist of opening the file, scrolling the document, selecting a frame, and enlarging the frame. We identified the minimal subset of the actions reported by the user that are needed to cause the failure from the critical state indicated in the bug report, which often corresponds to the minimal number of actions needed to reproduce the failure [53].

The amount of steps needed to reproduce a failure is an important information for estimating the complexity of testing techniques that work in the field and reveal failures by monitoring the status of the application to detect failure-prone states and executing test cases of appropriate complexity when a failure prone state is detected.

4.3 Results

RQ1: Why software applications fail in the field?

We analyzed the bug reports to distinguish faults that are due to insufficient testing (*bad testing* (BT)) from *field-intrinsic faults*. We further analyzed the field-intrinsic faults and identified four types of conditions that lead to field-intrinsic faults and that we use to classify such faults: *Irreproducible Execution Condition* (IEC), *Unknown Application Condition* (UAC), *Unknown Environment Condition* (UEC), and *Combinatorial Explosion* (CE).

The identified classes of faults comprise a complete taxonomy for the faults in the bug reports that we analyzed, and represent an initial general framework for classifying field faults. It follows a detailed explanation of each fault class.

Irreproducible Execution Condition (IEC) Faults

IEC faults are faults that can be revealed only under conditions that cannot be created in-house. This may depend on the impossibility of reproducing the complexity of the whole field environment, the inability of creating the specific failing execution or the evolution of the environment and the interactions with the SIF.

The safety critical routines to be executed in the case of natural disasters are good examples of execution conditions that might be impossible to reproduce in-house. Although a disaster can be simulated to some extent, a major natural disaster, for instance an earthquake or a tsunami, cannot be fully reproduced in-house, and some field-intrinsic faults may depend on extraordinary combinations of events that can be observed only in real conditions.

Similarly, the behavior of a system for an increasing number of users who interact with the application according to patterns that are not entirely predictable is often hard to test, especially for extreme situations, such as the extraordinary online streaming services workload experienced in the Super Bowl night [2].

The evolving varieties of configurations, for instance versions of the operating systems, drivers and plugins, are good examples of unpredictable changes to interactions between the SIF and the environment (hereafter *SIF-environment interactions*). New versions or entirely new plugins or drivers distributed after the most recent SIF release might generate faults impossible to reveal in-house before the release itself.

An example of such situation is the fault described in the EclipseLink bug report #429992. EclipseLink is an Eclipse plugin for developing Java applications that uses JPA to map objects into databases. The bug report indicates that EclipseLink silently ignores classes that contain lambda expressions: even if an object should be persisted in the database because its class includes the *@Entity* annotation, no table for persisting the object is generated in the database. Since lambda expressions have been introduced only in Java 8, it was impossible to test the combination of lambda expressions with JPA annotations when the EclipseLink plugin was developed, before the release of Java 8. EclipseLink should not have been affected by the presence of lambda expressions and should have supported the persistency of the classes regardless of the presence of lambda expressions. However due to an unforeseen compatibility issue, EclipseLink stopped working correctly when processing classes with lambda expressions.

Unknown Application Condition (UAC) Faults

UAC faults are faults that can be revealed only with input sequences that although executable in-house depend on conditions about the application that are ignored before the field execution and thus cannot be captured in in-house test suites.

An example of field failure that derives from unknown conditions is the Eclipse Subversive report #459010, which indicates that Subversive fails when retrieving folders whose name terminates with a blank character. This corner case is not documented in the specifications, and is hard to reveal with in-house testing because of the lack of information that may suggest to design test cases covering this specific situation. Structural test suites do not address this problem either, since many problems of this type are due to missed code, as in the case of this fault.

Another example of a UAC fault is the Eclipse #440413 bug report, which describes a fault in method `convertObjectToString` of class `XMLConversionManager` that converts any object to a proper string representation. The method works properly except when used to convert a `BigDecimal` representing a number in scientific notation, since it returns a string that still encodes the number in scientific notation and not a plain number as expected. We verified that this case is not mentioned either in the `XMLConversionManager` specification or in the API documentation, and is thus hidden to the testers who did not reveal the bug during testing and discovered it in the field after the software has been released.

In our experimental analysis, we did not have always access to the specification of the software. When this happened, we classified a fault as UAC when the inputs that lead to the failure are largely unrelated with the purpose of the functionality that fails, assuming that such cases were not defined in the specifications. Thus our classification may not be perfectly accurate.

Unknown Environment Condition (UEC) Faults

UEC faults are faults that can be revealed only with information about the environment that is not available before field execution. UEC faults are hardly detectable with in-house test cases designed without a complete description of the constraints on the SIF-environment interactions.

The full range of behaviors of third-party services that the SIF accesses through the network is a good example of information rarely completely available at design time, and thus possible source of UEC faults.

An example of UEC fault is the Eclipse bug report #394400 that indicates that EclipseLink may fail with a `NullPointerException` when executed under heavy load on the Oracle JRockit VM. The issue depends on the behavior of the Just In Time compilation feature of the JRockit VM that may reorder the operations executed within method `isOverriddenEvent` so that it returns an incomplete result. This undocumented behavior is responsible for the EclipseLink exception.

Combinatorial Explosion (CE) Faults

Even when the behavior of both the application and the environment are fully specified and can be replicated in-house, the combination of the cases to be tested may increase to a magnitude of cases that cannot be fully tested in-house. There are many sources of combinatorial explosion in software applications, such as the many possible configurations and preferences, the combinations of inputs and states, the many environments, for instance operative systems and hardware devices, that can be used to run an application, and so on. A well known example of combinatorial explosion of combinations are the sets of hardware devices, operating systems and configurations that comprise the execution conditions of smartphone applications that can almost never be fully tested in-house.

An example of a CE fault is the fault described in the Eclipse bug report #484494, which indicates that the diff feature of the Subversive plugin does not work when comparing a file to a symlink of a file that has been moved. Changing the location of a file referred by a symlink and using the symlink as part of a comparison is a legal combination of operations among the huge set of combinations that comprise to the sequence: ⟨change the status of a resource, use the changed resource as part of a computation⟩. Systematically testing all these combinations commonly exceed any reasonable albeit impressively large testing budget, because of the many ways resources can be changed independently from each other.

In our analysis, we observed that only a small percentage of CE cases are due to specific inputs (18% of the cases), while the rest of the CE cases are due to field elements.

Bad Testing (BT) Faults

We conclude our taxonomy with a discussion of BT faults, which we classify in our experimental analysis as *field* but not *field-intrinsic* faults. BT faults are faults that are not detected in-house due to weaknesses of the testing process. We include in this class all the faults in the field that do not belong to any of the previously described classes.

An example of BT fault is the fault reported in the Subversive bug report #326694, which indicates that Subversive erroneously reports as conflicting two identical files that have accumulated the same set of changes on two different branches. Since detecting conflicts is a primary feature of this plugin, developers should have tested a basic case like the presence of the same changes in two distinct branches.

Quantitative Analysis

Figure 4.1 summarizes the quantitative results of our empirical investigation. The bar chart indicates the number of faults classified in the five categories discussed above, and shows that field-intrinsic faults (the sum of the IEC, UAC, UEC and CE columns) are the majority of the field faults in our data set. Field-intrinsic faults represent 70% of the analyzed bug reports, thus confirming that field faults cannot be addressed by simply enhancing the testing process, but calls for specific approaches.

The bar chart indicates that *combinatorial explosion* (CE) is the most frequent cause of field-intrinsic faults, while *Irreproducible Execution Condition* (IEC) is the least common source of faults. Unknown execution conditions of either the application or the environment (UAC and UEC faults) are also relatively frequent cases. The dominance of CE faults is not surprising: The behavior of SIFs is influenced by many factors that can be never exhaustively tested in house. The many combinations that are hard

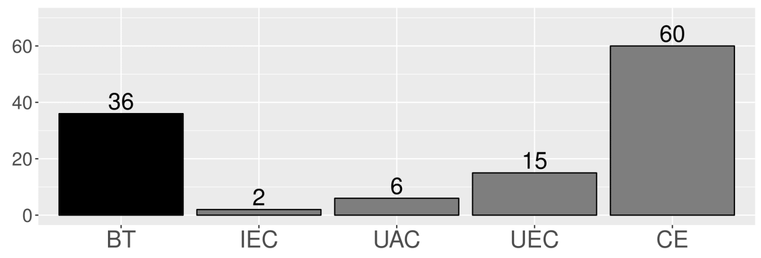

Figure 4.1: *Reasons why software applications fail in the field*

to design, foresee and test in house, can be order of magnitudes easier to address in the field, where such a diversity is spontaneously and implicitly available.

Our analysis identified few *Irreproducible Execution Condition* (IEC) faults, all caused by evolution of the SIF-environment interactions that emerged after the deployment of plugins not available at the time of testing, before the deployment of the SIF in the field. The scarce presence of IEC faults may depend on the nature of the applications that we analyzed. In other domains the presence of IEC faults might be higher. Consider for instance the domain of embedded software, where the interactions with the physical world might be sometime extremely hard to test. We observed a similar trend for the three subjects: a predominance of CE faults (openoffice being the highest at 73%) and a total of 10 - 20 % faults falling into the UEC/UAC category. It is worth noting that the two IEC faults we identified were both on the Eclipse platform.

RQ2: Which elements of the field are involved in field failures?

We analyzed the bug reports to study the role of field elements in field failures, and validate the intuitive hypothesis that many field-intrinsic faults may be hard to reveal in-house because their activation may depend on one or more elements that should be present in the field and should be in the right status to produce the failure. Below, we discuss the role played by the field elements that we introduced in Figure 2.1, provide concrete examples, and discuss the quantitative data from the experimental data sets.

Resources Software applications typically interact with many resources during the computation. For instance, many applications read from and write to persistent units, such as files and databases. Causes of field failures may involve resources in many ways. In our investigation, we observed two main cases: interactions between SIF and resources (hereafter *SIF-resource interactions*) that lead to performance problems and SIF-resource interactions leading to functional problems. The unbearable amount of time for SIFs to process some large resources and SIFs incorrectly handling resources of a particular type are examples of performance and functional problems triggered by SIF-resource interactions, respectively.

An example of SIF-resource interaction that triggers a performance problem is described in the OpenOffice bug report #95974. The OpenOffice writer crashes when trying to open a `.odt` document longer than 375 pages. The failure causes the CPU usage and the disk access rate to increase to 100%, and the application window simply crashes after one minute of unresponsiveness, activating the recovery wizard.

Plugins The plugin mechanism is a common solution to extend applications with new functionality in the field. In the presence of plugins, applications work as operating systems that embed the plugin executions, and interact with the plugins to access specific functionalities. Applications and plugins are developed and maintained independently. Evolution at either sides may trigger failures due to unforeseen interactions.

For example, the EGit bug report #383376 indicates that the repository search does not work on Github due to an unforeseen interaction with the Mylin Github connector plugin.

Operating system Many applications can be executed on different versions of different operating systems. The interactions of a SIF with a specific version of an operating system may trigger failures otherwise unexperienced.

An example of a problem involving the operating system is the failure documented in the OpenOffice bug report #126622 that describes how the OpenOffice writer does not correctly handle functionalities involving tables and queries under OSX. The failure prevents OpenOffice from closing, and forces the users to restart the operating system.

Drivers and services Applications often interact with third party drivers and services, whose availability depend on the production environment. During in-house development specific combinations might remain untested and failures unrevealed.

For example, the fault documented in the Eclipse Egit bug report #435866 indicates that the Eclipse Egit version control system fails to open the required network connections due to some unexpected changes of the authentication methods implemented in the Eclipse connection service.

Network Many software applications use the network to access resources or functionalities that are not available locally. With a plethora of different network protocols available, failures might be triggered when an application uses a specific protocol.

For instance, the fault described in the Nuxeo bug report #20481 describes a failure caused by a connection timeout that occurs when users download big zip files. Nuxeo does not handle connection timeouts properly and does not clean up temporary files, which leads to resources exhaustion.

None In a few cases the field-intrinsic faults do not depend on any interaction between the field elements and the SIF. Although not depending on any field element, these faults are still extremely hard to reveal at testing time, for instance because they can be revealed only by selecting a specific input out of a combinatorial number of cases.

This is the case of the OpenOffice bug report #126953, which indicates that when changing the format of a paragraph wrIECen with the Verdana font to italics bold, OpenOffice incorrectly adds blank lines before each occurrence of the brackets '(' and ')', and the text within the brackets disappears. This failure can be triggered only with a specific combination out of millions of input combinations: the use of Verdana font, and the presence of brackets when changing fonts to italics bold, out of the many combinations of font types, characters and font properties.

Quantitative Analysis

Figure 4.2 quantifies the impact of the different field elements on faults, by indicating the amount of faults affected by each type of field element. The causes are not exclusive, since a same fault may involve multiple field

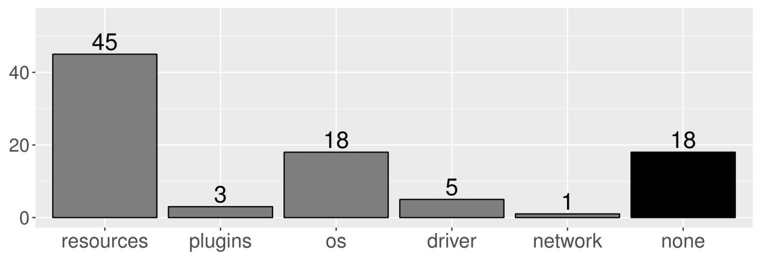

Figure 4.2: *Field elements involved in Field Failures*

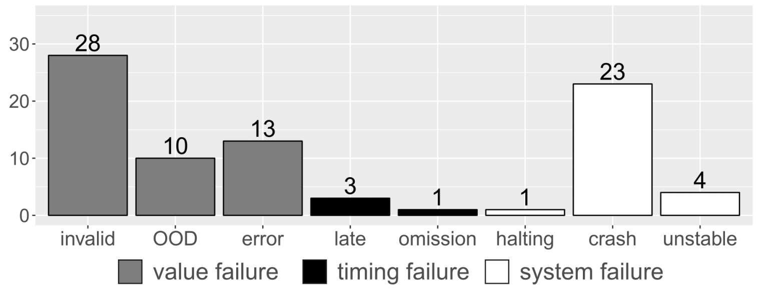

Figure 4.3: *Field Failure Types*

elements. In Figure 4.2, bar *none* reports the number of bug reports that describe failures that do not involve any field element.

Our analysis shows that interactions with the resources are the main cause of field-intrinsic faults (49% of the cases). Interactions with the operating systems are also a relevant cause of field-intrinsic faults (20% of the cases). Network, drivers & services, and plugins have been all observed as causes of field-intrinsic faults at least once, but they are collectively observed in a small proportion of the cases (10% of the cases in total). In total, 78% of the field-intrinsic faults interact with a field element.

Although the data reported in Figure 4.2 may be biased by the experimental setting, they already provide important information to define a research road map in the study of techniques to reveal and fix field-intrinsic faults.

RQ3: What kind of failures can be observed in the field?

We analyzed the distribution of failure types, and investigated the issues related to detectability. Figure 4.3 plots the distribution of the failure types presented in Table 4.1.

Most failures (51 out of 83 failures corresponding to 61% of the analyzed failures) are *value failures*, that is, executions that produce incorrect results. The most frequent case of *value failures* is the generation of invalid outputs, followed by the generation of error messages and the production of values out of domain (OOD in Figure 4.3). System failures are also frequent (28 out of 83 failures corresponding to 34% of the analyzed failures). They mostly lead to system crashes, and only occasionally to either unstable behaviors or system halt. Only a small set of the failures that we analyzed are due to the timing aspect (4 out of 83 failures corresponding to 5% of the analyzed failures). We observed few late timing and omission failures, and no early timing failures.

The results indicate that the generation of incorrect values (either invalid values, values out of domain or error messages) and systems crashes are the main classes of field failures (they represent 74 out of 83 failures corresponding to 89% of the analyzed failures). These results, and in particular the low frequency of timing failures, might depend on the domain that we investigated (desktop applications extensible with plugins and Web applications). We expect different frequencies of failure types in other domains: In particular, we expect an increasing frequency of timing failures in embedded systems, where the synchronization among the software components plays a relevant role.

Figure 4.4 plots the distribution of failures by detectability according to the classes presented in Table 4.2. A relatively high portion of failures are detected because the failures are either *signaled* by the application itself (14 out of 83 failures corresponding to 17% of the analyzed failures) or *unhandled* (25 out of 83 failures corresponding to 30% of the analyzed failures) causing a system crash. Such failures can be easily detected, on the contrary, *silent* failures (44 out of 83 failures corresponding to 53% of the analyzed failures) are hard to detect without some specific knowledge about the expected behavior of the application in response to certain stimuli, pointing to the well known oracle problem [21].

These results suggest that testing strategies working in the field without exploiting domain specific oracles could hardly reveal more than half of the field-intrinsic faults.

The considered subjects do not include mechanisms to automatically overcome from failures at runtime, and thus we have not observed any occurrence of *self-healed* failure.

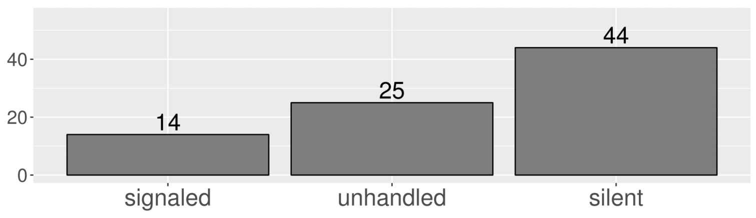

Figure 4.4: *Field Failure Detectability*

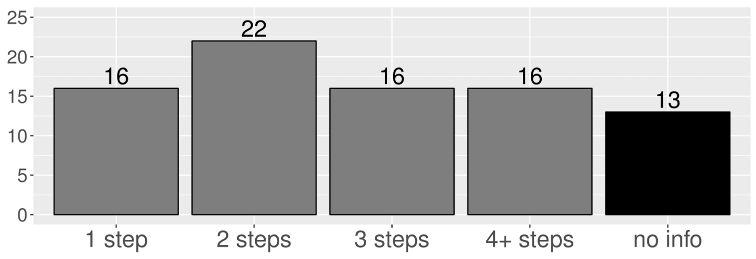

Figure 4.5: *Number of steps required to trigger a field failure*

RQ4: How many steps are required to trigger a field failure?

As discussed in Section 4.2, we computed the number of user actions necessary to trigger the failures by considering the operations that are described in the bug reports limiting to the ones essential for reproducing the failure.

Figure 4.5 plots the distribution of the field-intrinsic faults by the number of steps required to reproduce the failure. We were not able to determine the number of steps required to produce the failure in 13 out of the 83 analyzed failures (16% cases corresponding to bar *no info*), while we determined the number of steps required to reproduce the failure in 70 out of the 83 analyzed failures (84%), and observed that a large amount of failures can be reproduced with no more than three steps (54 out of 70 reproducible failures corresponding to 77% of the reproducible failures.)

These results provide useful information when designing field-testing approaches, since they suggest that only few actions are necessary to reproduce a failure once reached a failure prone state, and indicate that field testing strategies should focus more on detecting failure-prone states than on generating long action sequences to reproduce the failures.

4.4 Threats to Validity

We collected our experimental data from the bug reports of desktop applications extensible with plugins (Eclipse and OpenOffice) and Web applications (Nuxeo) by examining a limited although reasonable amount of bug reports. The results give early evidence of the nature of the failures that can be experienced in plugins and Web applications, and need further studies to be generalized to other kinds of applications and to be quantitatively assessed.

We defined the classification schema, and analyzed the bug reports manually. The author of this thesis and another researcher have independently analyzed the bug reports and discussed the conflicting cases until reaching a consensus. Although the process we followed should mitigate the risk of misinterpretation of the cases, we cannot fully exclude clerical errors in our analysis. The data and the detailed material that we refer to in this study are publicly available for independent inspections and further uses.

The bug reports that we examined might be inaccurate some times. They may for example include partial information about the failures. Although we cannot fully eliminate this potential issue, we believe that possibly incomplete bug reports considered in the experiments may have reduced the number of field-intrinsic faults that we identified, thus only pessimistically affecting the results. In particular, the lack of information about a failure may have increased the chance of a fault to be erroneously classified as irreproducible execution condition, while the unknown conditions about the application or the environment may have reduced the amount of faults classified as combinatorial explosion faults. We assume that our results that indicate a density of 70% of field-intrinsic faults among the analyzed bug reports is a conservative under approximation of the field-intrinsic faults that are present in the examined applications.

4.5 Findings

The experimental data that we collected lead to some interesting findings:

Most of the failures that can be observed in the field are caused by field-intrinsic faults: Our experimental data indicate that about 70% of the field failures that we analyzed are caused by field-intrinsic faults, that is, are caused by faults that might be hardly revealed in house. These faults are caused by four challenges: combinatorial explosion, unknown environment or application condition, and situations impossible to reproduce. This result calls for approaches that can deal with these classes of failures in the field.

Combinatorial explosion is a relevant cause of undetected field-intrinsic faults: Combinatorial explosions are notably hard to address in testing and analysis techniques. Our experimental investigation indicates that, despite numerous techniques developed to tackle the problem of generating test cases that adequately cover interactions of parameters in a software application [43, 49], combinatorial explosion still plays a prominent role in preventing the detection of field-intrinsic faults. Differently from other contexts, in the case of field-intrinsic faults, the source of combinatorial explosion is not the user input (only 18% of the failures are caused by specific combinations of inputs) but the status of the field elements.

The interaction with the environment is almost always a relevant factor in field-intrinsic faults: The vast majority of the field-intrinsic faults (78% in our study) requires some forms of interactions with the environment to be activated. Resources and operating systems are the most relevant field elements involved in field-failures, but also drivers, plugins and the network are often important. This result indicates that techniques to reveal field-intrinsic faults must take into consideration the production environment in which the system is executed.

Value and system field-faults are more frequent than timing field-faults: The ability to analyze the output produced by a system, including the ability to detect crashes, is sufficient to detect most of the field-intrinsic failures, with a rate of timing field failures as low as 5% of the cases.

The oracle problem affects about half of the field-intrinsic faults: Our experimental analysis indicates that 43% of the failures can be detected by intercepting unhandled events, for example system crashes, and error messages. Domain specific oracles are necessary to address the remaining 57% of the cases. This calls for techniques and methods to derive strong automatic oracles for field testing.

Field failures can be commonly revealed with short sequences of actions: Our experimental analysis provides evidence that few steps (three or fewer actions in 77% of the cases) are usually needed to make the SIF fail from a failure-prone state. This suggests that detecting states that offer opportunities for running test and analysis routines might be more important than studying techniques for generating tests composed of long sequences of actions.

4.6 Discussion

The taxonomy that we proposed in this paper opens new scenarios of increasing complexity. BT faults simply substantiate the need of improving the in-house testing process and do not introduce new challenges for the software testing community. UAC, UEC and CE faults call for new techniques to enrich well designed test suites with test cases identified in the field while experiencing faults caused by unpredictable (UAC and UEC) or impossible-to-exhaustively-test (CE) conditions. The main challenge that has been only partially addressed so far is to record execution sequences that lead to failures in the field, and reproduce them either in the field or in house to identify and remove the faults. Being not executable in house, IEC faults further challenge the software testing community with the problem of executing failing test cases in the field. The main challenges are to both reveal failures by executing test cases in the field, which requires to control the execution of the test cases in the usually complex field context, and prevent any side-effect for the users.

Motivated by the results of this study, we present a client-server architecture that allows to deploy and orchestrate testing capabilities in the field and an approach to write test cases that can be executed in the field.

Chapter 5

An Architecture for Field Testing

This chapter presents a client-server architecture designed to effectively test applications in the field.

5.1 Overview

The architecture is composed of a client and a server, the client is deployed as a background service running on the target platform, while the server is on the developer side, communicating with the client side components by means of remote communication. This architecture is designed for the purpose of executing test cases in the field (whenever possible), and sending the results back to the server side. The client service continuously monitors the field environment for changes in configuration (hardware components, operating system updates, application setting, etc.), when such changes are observed the new configuration is tested. If there are tests that cannot be executed client side, a testing environment with the same configuration observed in the field is simulated server side, and the tests are executed there. Both the server and the client maintain a configuration model that allows the system to understand when a new untested configuration is observed. This architectural design is meant to be general, and, while the server implementation should be installed on a machine with good computing power and network availability, the client side can be implemented in any device

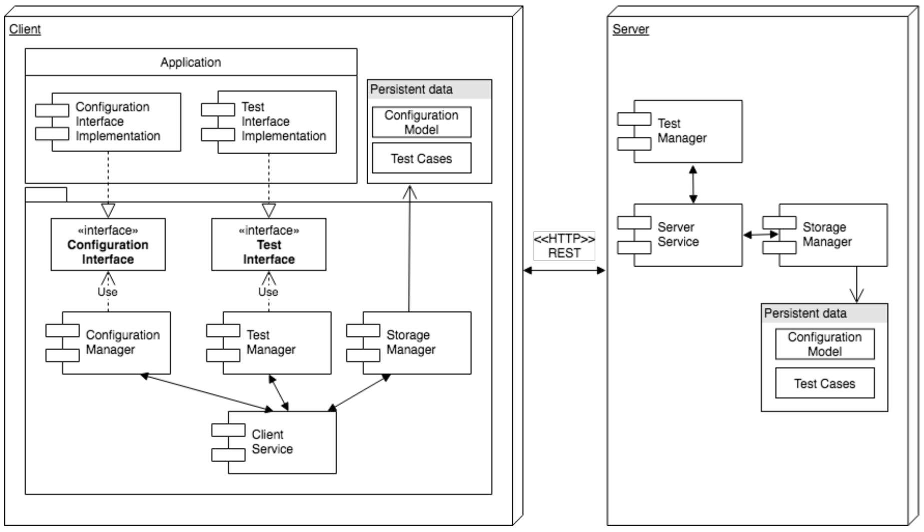

Figure 5.1: *High-level overview of the framework*

with an operating system that supports a programming language that can be used to implement the described components.

5.2 Client-side Components

The *Client Service* is the main service that runs in the background and carries out the main activities of the framework by orchestrating the other components of the framework. Also, it handles the remote communication with the server side.

The *Configuration Manager* manages the configuration model and is responsible for querying the application to discover runtime configuration changes.

The *Test Manager* manages the retrieval and execution of test cases whenever the *Client Service* deems it necessary to execute tests on the current configuration of the application. This component is also responsible for ensuring isolation of test executions.

The *Storage Manager* manages the access to local persistent data stored in the application's local storage area. Such data include the configuration model implementation and field test cases.

The *Configuration Interface* needs to be implemented by the application for the purpose of retrieving the current configuration. The client-side

Configuration Manager can poll the interface to scan for changes in configuration.

The *Test Interface* needs to be implemented by the application for the purpose of creating a (partially) isolated instance of the application and launching test cases at runtime. The client-side *Test Manager* can use the interface to launch the testing process.

5.3 Server-side Components

The *Server Service* is the main service that runs in the background and carries out the main activities of the framework by orchestrating the other components. Also, it handles the remote communication with the client side. The *Test Manager* manages the execution of test cases whenever a test request comes from the client, the objective of the server-side test manager is to run test cases that cannot be executed in the field. The *Storage Manager* manages the access to local persistent data stored in the application's local storage area. Such data include configuration models, configuration instances, test cases, etc.

5.4 Components Interaction

In this section we describe how the components interact to achieve the goal of testing the application when new configurations are observed. The interactions are divided into three main streams: checking the application for new configurations, test a new configuration and checking the server for updates.

Check the application for new configurations

Figure 5.2 illustrates the interaction between components when a new configuration check is performed. This check is periodically initiated by the Client Service in order to mantain an updated configuration model and react to new untested or unknown configurations. First the *Client Service* asks the *Configuration Manager* to get the current configuration of the application, once the configuration is obtained, a request is sent to the *Storage Manager* to check if the configuration is new or already present in the *Con-*

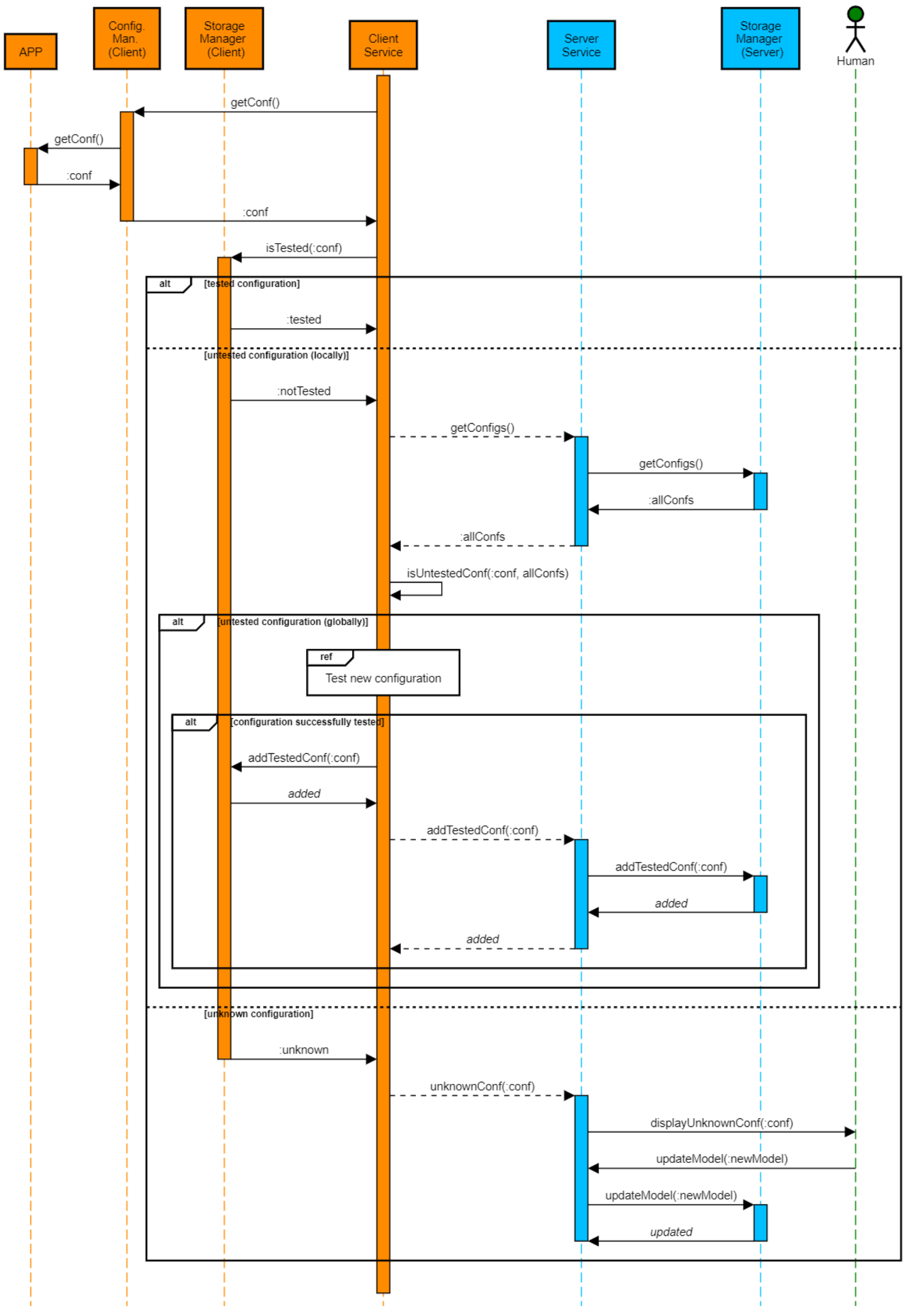

Figure 5.2: *Sequence diagram illustrating how the framework components interact when discovering new configurations*

figuration Model (this procedure can be intermediated by a caching mechanism).

An observed configuration can be *tested*, *untested* or *unknown*. If the configuration is tested, it has already been observed earlier and tests have already been performed on the application in such configuration. An untested configuration has never been observed and tested before. An unknown configuration contains at least an element that does not fit the configuration model used.

If the observed configuration is not present in the database (untested locally), the *Storage Manager* adds it and then returns the control to the *Client Service*. The *Client Service* at this point asks the list of the configurations tested so far to the remote *Server Service*, once the list of configurations is returned to the *Client Service*, this last component checks if the configuration that was identified as untested locally is also untested globally; if this is the case the new configuration is sent to the *Server Service* to be added in the server *Configuration model*. The other alternative is that the newly observed configuration does not fit the model (it is hence unknown), in this case the unknown configuration is sent to the *Server Service* that asks for human intervention to possibly update the configuration model.

Test a new configuration

Figure 5.3 and 5.4 illustrate the interaction between components when a previously untested configuration is tested by the framework. First (Figure 5.3) the *Client Service* delegates the control of the testing procedure to the *Test Manager (Client)*, then the *Test Manager (Client)* retrieves the test cases from the storage manager, and executes them on the *Mobile App (Client)*, but only if an isolation step is successful. The isolation process starts from the test case itself that, depending on the actions it performs has specific isolation requirements. The test case asks the test interface implementation in the application if the isolation requirements can be met, then, if so, the application instantiates the necessary level of isolation, the test case body executes and the control returns to the *Test Manager (Client)*, which collects the result of the executed tests and the list of test cases that could not be executed, and then returns them to the *Client Service*. The *Client Service* then calls the *Server Service* (Figure 5.4) asking it to perform

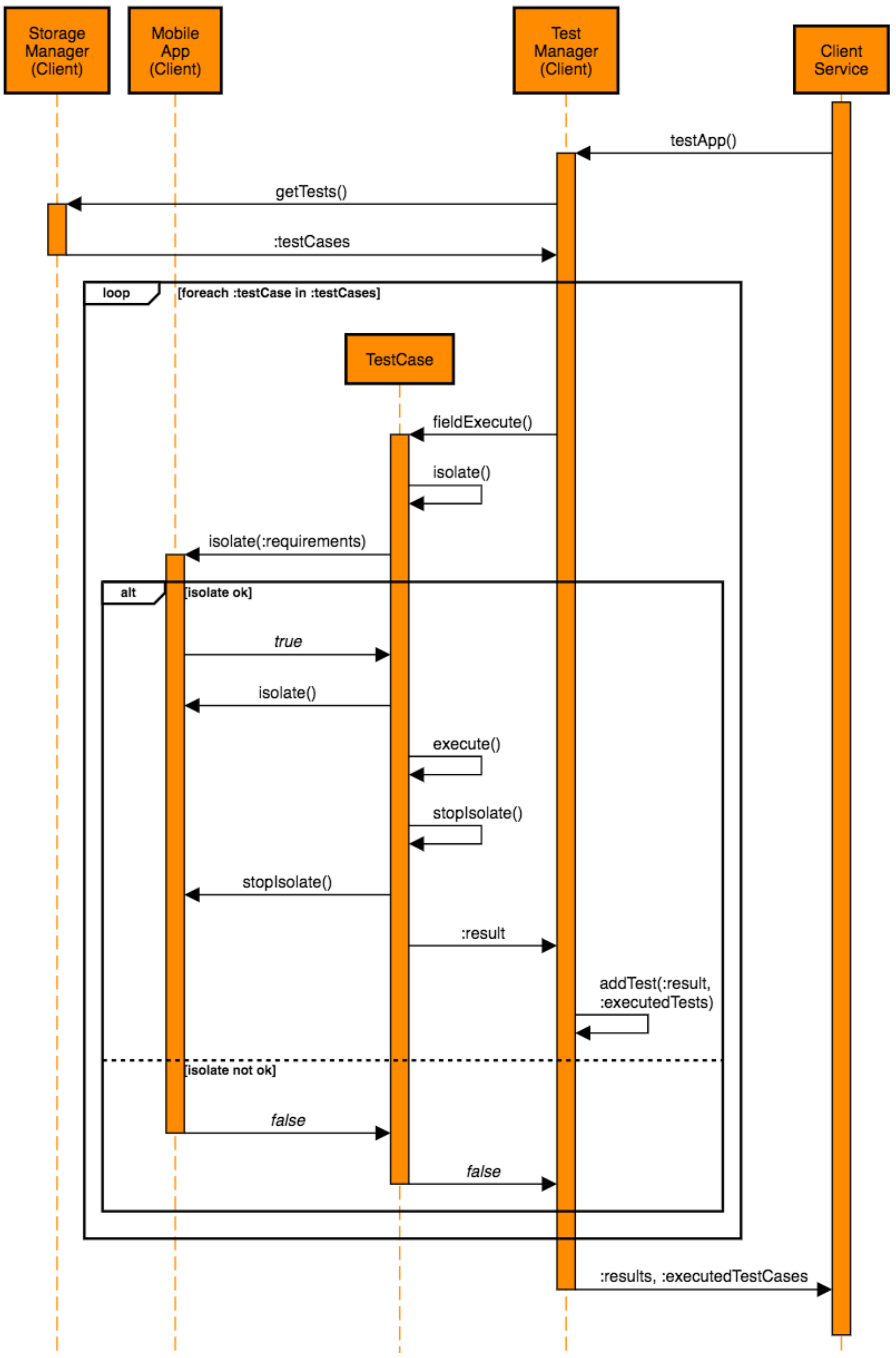

Figure 5.3: *Sequence diagram illustrating how the framework components interact when testing a new configuration*

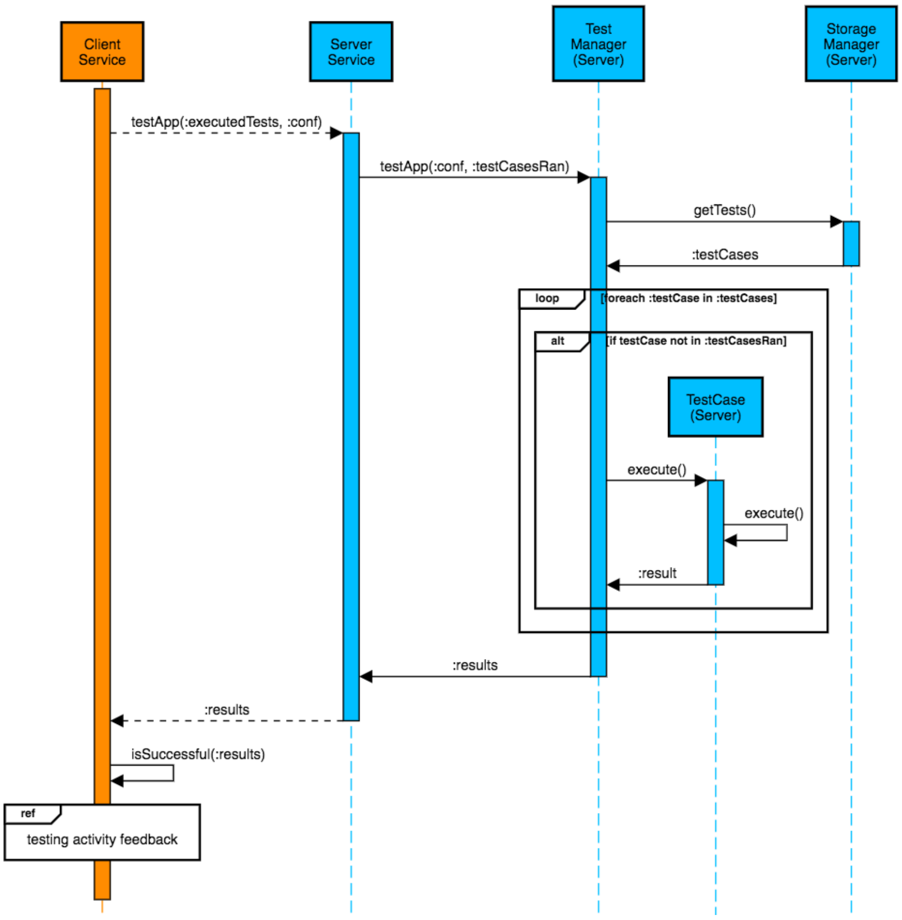

Figure 5.4: *Sequence diagram illustrating how the framework components interact when testing a new configuration*

the tests that could not be performed in the field. The tests are executed in the same way as the client size (Without the need for isolation). The test results are then returned to the *Client Service* that can continue to update the tested configurations.

Check the server for updates

Figure 5.5 illustrates the interaction between components when clients query the server to obtain an updated configuration model or new tests. The procedure starts with a *Client Service* asking the *Server Service* for the updated configuration model and new tests. The *Client Service* then checks if the received model matches the local one and, if not, it updates it through the *Storage Manager*. The textboxes in the diagram (for example *update configurations*) refer to activities which are already illustrated in Figure 5.2 and 5.5. This update activity is common to all the clients installed on different instances of the target application, hence the server must be able to efficiently and effectively deal with multiple requests that might occur at the same time.

5.5 Implementation of the Architecture

This architecture can be realized in different application domains, although the difficulty in the implementation might depend on the capability of the target technical domain to provide isolation in the field. Testing the application in the field in isolation is a relevant challenge, as we discussed in Section 3.2, some approaches can be used by the Test Interface Implementation to obtain the correct level of isolation.

Android, for example, offers the possibility of defining different user profiles [12], which can contain copies of the applications installed on the system; a *testing profile* can be thus used to isolate the application under test and its data.

In the domain of cloud-based service-oriented applications the isolation problem can be tackled at the service level by using a new instance of the service under test, in this case the system has to implement stub services that can interact with the one that is being tested.

Isolation can be achieved in Unix-based operating systems by using the fork system call, which spawns a copy process with a separate address

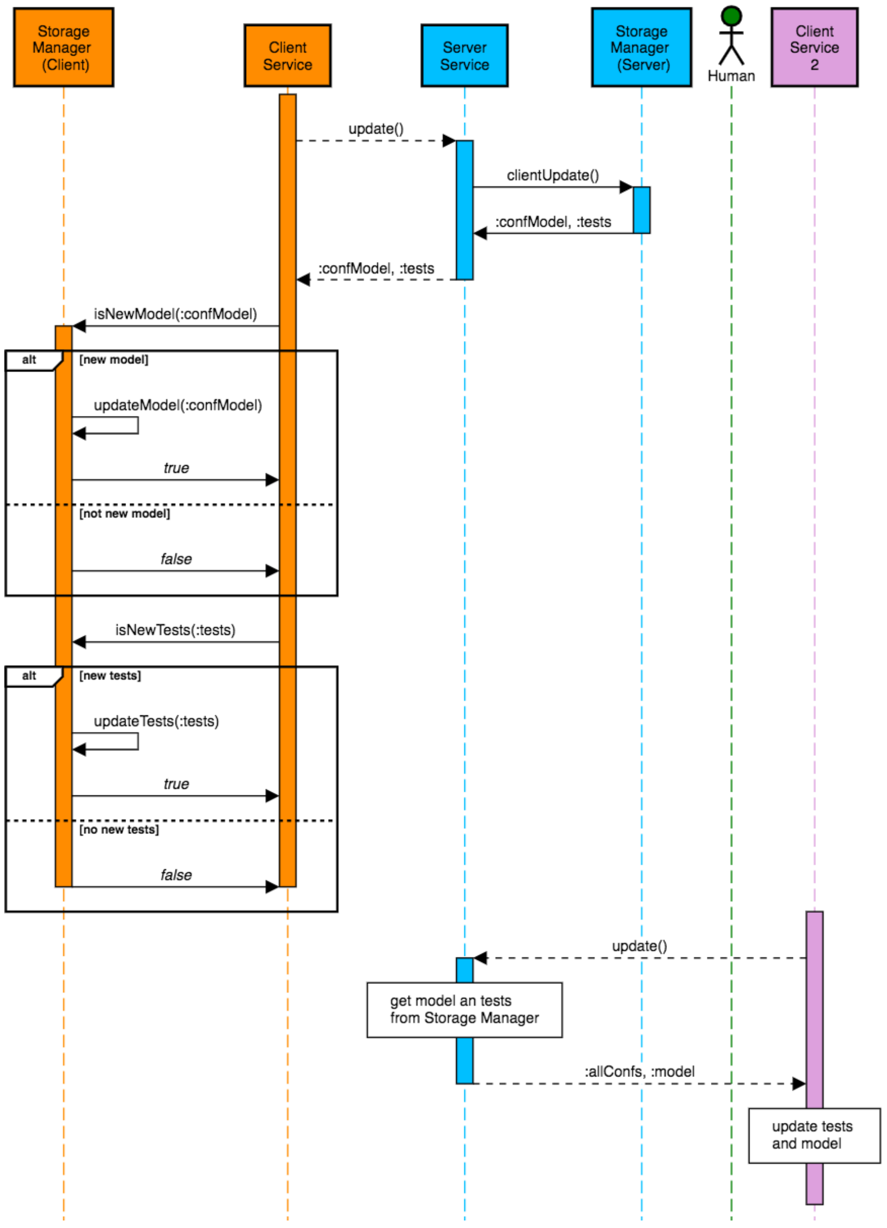

Figure 5.5: *Sequence diagram illustrating how the framework components interact when querying the server for changes in the model and configurations*

space. In this case the application under test must not support multi-threading (the fork system call only copies a single thread).

Virtual machines are a possible approach to achieve isolation, but in general it is computationally expensive to run them. Containers are in general more lightweight, and could be a valid alternative.

A relevant challenge that is related to isolation is handling the interaction of the software application with external resources. There are multiple resources the application can interact with in the field (e.g., files and databases) and testing these interactions without causing any side effects requires the field testing framework to implement counter measures such as using a virtual file system or a database mock-up.

Executing the test cases at runtime is another relevant technical challenge. Java test cases that work at unit level can be executed in the field using the JUnit framework [17] runners, using the same approach we present in Chapter 6, however system tests, especially if they involve the use of a graphical user interface, might require stubbed system components.

The configuration model mantained by the Storage Manager is another component of the architecture that can be implemented in different ways. We believe that a valid implementation is a tree feature model [22], which represents the configuration space in a tree structure, where leaves are single features. With a hierarchical structure and the possibility to express constraints on combinations of features, this solution can be used to effectively represent the configuration space.

In Android for instance the model can be constructed with the information present in the application preferences files and in the `android.os.Build` class, which contains information about the device and operating system configuration. The Configuration Interface Implementation, which is charged by retrieving configuration information, can extract information from the shared preferences files and the `Build` class at runtime.

5.6 Discussion

We presented an overview of an architecture that can be used to effectively perform field testing. This architecture uses field configuration updates as a trigger for the testing process, and introduces the possibility of performing traditional in-house testing for the functionalities that cannot be tested in the field.

We understand that whenever data can be collected from the field, privacy issues can arise. In this case the problem is mitigated by the fact that when testing the application in the field the data can be manipulated, but only the test result is sent to the developers. The problem however could be observed if the configuration model includes sensitive information, in this case a data anonymization process is needed to preserve user privacy.

Chapter 6

An Approach to Field Testing

In this chapter we present a novel approach to write tests that can be executed in the field, either from scratch, or starting from a standard program test suite. In this chapter we first present background information on aspect-oriented programming (specifically AspectJ), which is a key enabler of our approach. We then overview how the proposed field testing approach works at a high level and finally we discuss details about its implementation.

6.1 Aspect-Oriented Programming

AspectJ [14] is an aspect-oriented programming (AOP) [38, 39, 29] extension for the Java programming language. AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding behaviors to existing code without modifying the code itself. Instead, using AOP, the programmer can declare separately which code is to modify and how. AspectJ uses Java-like syntax, and it lets programmers define special constructs called aspects. Aspects can contain several entities unavailable to standard classes, allowing the programmer to alter the regular execution flow of the program that is being executed. This is done through the join-point / advice construct: a programmer can specify *join-points* (well-defined places in the execution of a program, like method call, object instantiation, or variable access), and *advices* which define code to run at a matched join point. We present a simple example to explain how AspectJ works.

We can see in Listing 6.1 a simple Java method that divides a `double a` by a `double b`. Now, suppose that we want to add a guard that checks that the `double b` is not zero, but we want to do this without modifying the code of the method. What we can do is first define a join-point that captures every call to the method `divide(double a, double b)`, as we can see in Listing 6.2. This join-point captures all method calls that match the signature `double Math.divide(double, double)`, note that AspectJ also allows wildcards in the method signature: for example `* Math.*(double, double)` captures any method from the Math class with any return type and two `double` variables as argument. In Listing 6.3 we can see how the control flow of the original method can be modified: the `around` construct allows to shortcut the advised method execution by returning its own return value or throwing an exception. In this case, if `b`, the divisor, is zero, the advice throws an `IllegalArgumentException`, if not, the control is given back to the original method with the `proceed` statement.

```
public double divide(double a, double b) {
        return a/b;
}
```

Listing 6.1: *Simple method that divides* `a` *by* `b`

```
pointcut callDivide(double a, double b) :
        call(double Math.divide(double, double)) && args(a, b);
```

Listing 6.2: *Join-point for the calls to* `divide(double a, double b)`

```
boolean around(double a, double b) :
        callDivide(a, b) {
        if (b == 0) {
                throw new IllegalArgumentException("Divisor cannot be
                    zero!");
        }
        return proceed(a, b);
}
```

Listing 6.3: *Advice that checks the divisor value, and throws exception if it is zero*

6.2 Moving Test Cases to the Field

6.2.1 General Idea

Software applications behavior in the field is difficult to predict, because users can stimulate applications with a great number of different inputs and also the field environment can be configured in many different ways. Testing software applications in-house for all possible combinations of user inputs and field configurations is practically impossible.

As we saw in Chapter 4, our study suggests that combinatorial explosion is one of the most prominent causes of software failures in the field. Hence the idea of exploiting the field configuration and the user input that comes from the field as a test case parameter.

This testing process requires test cases specifically written to run in the field. In particular, given the fact that the tester has not the complete control over the test inputs, specific assumptions are needed to manipulate the inputs consistently. Also, the oracle problem, which already exists for in-house testing, is magnified by field testing, since the test oracle is no longer referred to a specific input case, but has to be general enough to work with different test inputs that can be found in the field, hence properties checked after the execution of the tests could be less strict.

A different approach could be inserting the assertions directly in the application source code. While this would allow us to check properties that we deem interesting at runtime, a drawback is the fact that this would require the modification of the original application source code. Another drawback is the fact that assertions represent invariant properties that must always hold, while test cases allow us to specify properties that differ depending on the test inputs used.

The key elements that characterize our field test cases are:

- *aspects*, which allow the modification of an application behavior without modifying its code, and can be used to trigger the field test cases when specific conditions hold.
- *parameterized test cases*, which exploit the input extracted from the field to test the software instead of using predefined values and configurations.
- *test input checks*, which filter out input values and configurations that

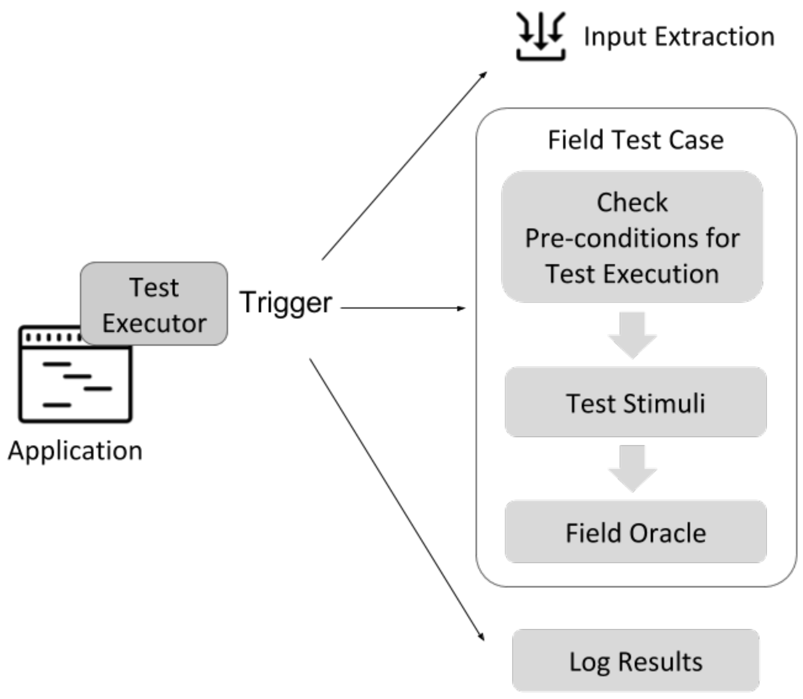

Figure 6.1: *Our field testing approach*

cannot be used by the field test case.

- *generalized oracles*, which are assertions that can be checked on the test result produced in the field.

6.2.2 Field Testing Structure

Figure 6.1 shows the structure of a field test case. Specific events in the application (e.g., a method call that matches a join-point) trigger the testing process: first the Test Executor extracts relevant inputs from the object that triggered the testing process, then the Field Test Case itself is executed with the captured input. In the test case structure we can see that in a first phase the field input is checked cor consistency (Check Pre-conditions for Test Execution), then exploited by the body of the test (Test Stimuli), and finally the test oracle defines the outcome of the testing process (Field Oracle). In the end the control returns to the Test Executor which logs the test results (Log Results).

6.2.3 How to Write Field Tests

This section explains how to write test cases that use inputs captured at runtime instead of statically generated. Note that we choose to work with Java and AspectJ because of the popularity of the Java programming language, but the general approach can be applied to programs written in other programming languages.

The first operation is identifying the events that should trigger the field testing process (e.g., method call), then an aspect that captures the event and the involved data can be implemented. The aspect must pass the data that will be used as input to the field test case. The test case must be implemented in a way that it can exploit the data produced by the aspect. The aspect will call the test when it is triggered and then log the test results.

An example faulty version of the JFreeChart project (fault number 2) in the Defects4J dataset [37] is used to explain the procedure. In the code snippets in Listing 6.4 and 6.5 we highlight the parts which are specific to the example and that must be changed when considering different test cases (the definition of the pointcut and the test class name), note that the majority of the code does not change.

Input Extraction

Suppose we want to test the method `iterateDomainBounds`, and instead of creating a new `XYDataset` object, we want to use the one that is actually created in the field as argument of a field test case. To do so we need to intercept the method call and save a reference to the calling object. Listing 6.4 shows the instructions that can be used to achieve such goal: with the `pointcut` instruction we define a join-point that captures every method call with the signature matched by the regular expression in the `call` method. The instrumentation also gives us the possibility to read and manipulate the object that called the captured method (`target`) and the method arguments (`args`). In this case we are interested in intercepting a method call, in other cases it might be more meaningful to capture the creation of an object (i.e. constructor) and test the newly created object; in general each test case requires a specific entry-point to capture its input.

```
pointcut
     callIterateDomainBounds( XYDataset dataset, boolean includeInterval ):
     call( * DatasetUtilities.iterateDomainBounds(XYDataset, boolean) )
     && args ( dataset, includeInterval ) && if(!TestFlags.testing);
```

Listing 6.4: *pointcut*

Once a pointcut is defined we can write some code that executes *before* the captured method, this is done by defining an advice (Listing 6.5). First we set a flag in line 3 which prevents the advice to trigger also on the test case code, then at line 4 we save a reference to any interesting item that we want to use as input in the test case to be executed, the references are saved in a simple static field of the class `TestStorage`. Line 8 launches the test case using the standard JUnit runner, and in line 18 the test failures (if any) are collected and logged.

```
1   before( XYDataset dataset, boolean includeInterval ):
         callIterateDomainBounds( dataset, includeInterval ){
2
3     TestFlag.testing = true;
4     TestStorage.dataset = dataset;
5     Logger logger = SingletonLogger.getInstance();
6
7     JUnitCore jUnitCore = new JUnitCore();
8     Result result = jUnitCore.run( DatasetUtilitiesTestField.class );
9
10     logger.info("test class: DatasetUtilitiesTestField");
11     logger.info("ran: " + result.getRunCount() + " failed: " +
          result.getFailureCount());
12     List<Failure> failures = result.getFailures();
13     if(!failures.isEmpty()) {
14      for(Failure f : failures) {
15       logger.info(f.getTrace());
16      }
17     }
18
19    TestFlag.testing = false;
20
21  }
```

Listing 6.5: *advice that captures calls to* `DatasetUtilities.iterateDomainBounds` *and runs the corresponding test case*

Test Case

We can see the general field test case structure in Listing 6.6. There is a first phase of pre-condition checking, then the test manipulates the input data and in the end the test oracle checks post-conditions on the test data.

```
public void fieldTest() {
  <<check input pre-conditions>>
  <<perform test stimuli>>
  <<check post-conditions>>
}
```

Listing 6.6: *pseudo-code for a field test case*

Check pre-conditions Now we can look at the code of the field test code for the JFreeChart `iterateDomainBounds` method. We can see in Listing 6.7, Line 3 that the test case does not statically create the `XYDataset` instance, but it obtains the object from the advice. Note that, having no control over the object used in the test case, we have to check that the object meets the requirements to be used in the test case, in this case the user might have called the `iterateDomainBounds` method with an empty dataset, so we avoid this possibility by aborting the testing process if this is the case (Line 2), because in this case it would not be meaningful to test a null object.

```
1 public void testFindDomainBounds() {
2   assumeTrue(TestStorage.dataset.getSeriesCount() >= 1);
3   XYDataset dataset = TestStorage.dataset;
4   Range r = DatasetUtilities.iterateDomainBounds(dataset);
5   assertNotNull(r.getLowerBound());
6   assertNotNull(r.getUpperBound());
7 }
```

Listing 6.7: *test case for DatasetUtilities.iterateDomainBounds using the captured object*

Test stimuli As in a traditional testing process, the method under test is called. In this case the test input is the input extracted from the field (Line 4). The test stimuli must not modify the state of the objects the application is using, hence any manipulation on the test input must be done taking this under consideration. To write this test case we used the test case `DatasetUtilitiesTests.testFindDomainBounds` in the original JFreeChart test suite as a starting point.

Check post-conditions (Oracle) After the body of the method has executed, the oracle checks if the test has been successful (Lines 5,6). Oracles for field tests are weaker than regular oracles because the tester does not have complete control over the test inputs. In this example we can see that the test result is checked for non nullness. If we look at the original test case code in Listing 6.8 we can see that the assertions instead check that the method `getLowerBound` returns a specific value, because the dataset values are known, since it is statically created by `createXYDataset1()` (Listing 7.3).

In other cases the oracle can be tighter, depending how complex the application domain is. Take for example the test case in Listing 6.10: this test case has a static object initialization and hence the assertions at lines 6, 7, 8, 9 are very specific and cannot be used in the field, since we cannot know in advance the state of the object used in the field to run the test cases. Instead, in the field version of this test case, we have to weaken these assertions, as we can see in Listing 6.11, Line 6.

```
public void testFindDomainBounds() {
    XYDataset dataset = createXYDataset1();
    Range r = DatasetUtilities.findDomainBounds(dataset);
    assertEquals(1.0, r.getLowerBound(), EPSILON);
    assertEquals(3.0, r.getUpperBound(), EPSILON);
}
```

Listing 6.8: *DatasetUtilitiesTest.testFindDomainBounds()*

```
private XYDataset createXYDataset1() {
    XYSeries series1 = new XYSeries("S1");
    series1.add(1.0, 100.0);
    series1.add(2.0, 101.0);
    series1.add(3.0, 102.0);
    XYSeries series2 = new XYSeries("S2");
    series2.add(1.0, 103.0);
    series2.add(2.0, null);
    series2.add(3.0, 105.0);
    XYSeriesCollection result = new XYSeriesCollection();
    result.addSeries(series1);
    result.addSeries(series2);
    result.setIntervalWidth(0.0);
    return result;
}
```

Listing 6.9: *DatasetUtilitiesTest.createXYDataset1()*

```java
public void testAddOrUpdate3() {
  XYSeries series = new XYSeries("Series", false, true);
  series.addOrUpdate(1.0, 1.0);
  series.addOrUpdate(1.0, 2.0);
  series.addOrUpdate(1.0, 3.0);
  assertEquals(new Double(1.0), series.getY(0));
  assertEquals(new Double(2.0), series.getY(1));
  assertEquals(new Double(3.0), series.getY(2));
  assertEquals(3, series.getItemCount());
}
```

Listing 6.10: *`XYSeriesTests.testAddOrUpdate3()`*

```java
public void testAddOrUpdate3() {
  XYSeries series = (XYSeries) TestStorage.series.clone();
  series.addOrUpdate(1.0, 1.0);
  series.addOrUpdate(1.0, 2.0);
  series.addOrUpdate(1.0, 3.0);
  assertTrue(series.getItemCount() >= 3);
}
```

Listing 6.11: *`XYSeriesTests.testAddOrUpdate3()` with relaxed assertions*

6.3 Discussion

In this Chapter we presented how to implement field test cases that use field data to test the software. Field test cases share common elements with in-house test cases, such as test stimuli and a test oracle, but we can observe substantial differences:

- field test cases use input extracted from field executions.
- the test input needs to be checked for consistency before being manipulated by the test case.
- field test cases have to use weaker oracles in order to check test results.

Chapter 7

Empirical Evaluation

This chapter describes the empirical evaluation of the field testing approach described in Chapter 6. We decided to assess how field testing compares to traditional in-house testing by measuring the number of additional faults that can be revealed by our approach compared to in-house testing.

In Section 7.1 we present the dataset we decided to evaluate our approach with, Section 7.2 introduces the experimental procedure we followed, in Section 7.3 we present the quantitative results we obtained and in Section 7.4 we further explain the field testing procedure by analyzing two concrete field testing cases.

7.1 Subjects of the Study

We decided to validate our approach using the Defects4J dataset [37], a dataset that includes 395 real faults spread across six Java projects. This is one of the most used datasets for the evaluation of testing approaches of Java programs. For each bug the dataset stores the faulty program version, with at least one test case that exposes the fault, and the fixed version that passes all the test cases, with a minimal fix that does not introduce any irrelevant changes: this allows the users to study each bug in isolation. We used two of the six java projects that the dataset includes: JFreeChart [16], and Apache Commons Lang [13], the first is a Java chart library and the second a library that consists of utilities for Java core classes. Both projects are popular in the Java ecosystem and they are reasonably mature and well tested. The first project includes 26 versions of the program, and the second one 65; each one with its own test suite. Each version contains a real fault, reported on fault reporting platforms and confirmed by the developers. Each of these faults is exposed by one or more test cases. For each buggy version of the program, the dataset also includes a fixed version, where the corresponding fault has been fixed. We chose not to use the same software we performed our failure study on because of the ease of use and documentation that the Defects4J dataset provides, and because part of software analyzed for the study is not written in java.

7.2 Experimental Procedure

The objective of our experiment is to assess whether the field test cases can reveal faults that were not been revealed by the original test suite. The faults reported in the dataset were found after the application was released with a passing test suite, hence they were not revealed by in-house testing. We defined a research question to drive this evaluation:

RQ: What percentage of faults missed by in-house testing can be revealed by field testing?

We wrote field test cases starting from the original test suite of the programs under test, to avoid any bias we started from the original test suite and adapted the test cases so that they could work in the field. We define a rigorous procedure to transform a test case into one that can be executed in the field and we assess if it can reveal a fault that was missed by the original test suite.

We studied if any of the passing test cases could have exposed the fault, that is, if any of the passing test cases adapted to run in the field failed. We then took the original test suite, eliminated the failing test cases added to reveal the bug and adapted the passing test cases to run in the field. Revealing additional faults is not obvious because the existing test cases may cover cases that are unrelated with the fault. The procedure to validate our testing approach (for each fault in the dataset) is:

1. Eliminate the failing test cases from the test suite.

2. Modify the test case that executes the faulty method to take test inputs from the field.

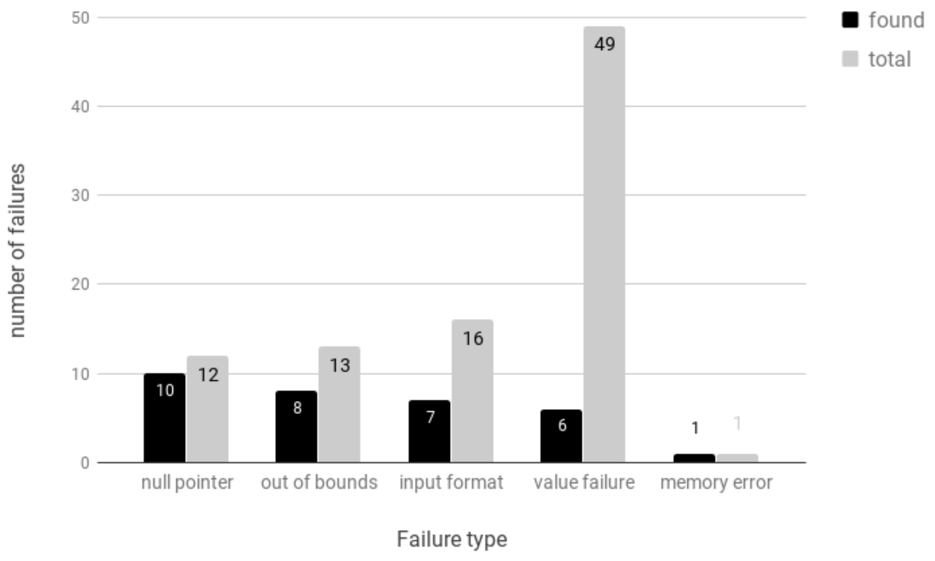

Figure 7.1: *failure types observed*

3. Write a join-point that intercepts calls to the tested method (or constructor).

4. Write an advice that supplies the test case with field input and runs it.

5. Run the test cases with different inputs to check if the fault can be revealed.

7.3 Results

To answer *RQ* we applied the procedure explained in Section 6 to the 26 versions of JFreeChart and the 65 versions of Apache Commons Lang in the Defects4J dataset. We were able to successfully reveal the fault in the version in 12 out of 26 cases (46%) for JFreeChart and in 20 out of 65 cases (30%) for Apache Commons Lang. In total, out of the 91 faults considered, 32 (35%) were revealed by our approach.

Table 7.1 shows the fault and failure types revealed for each of the dataset item where our testing approach was successful. We used the failure classification we proposed in Chapter 4 to identify the reason why the

Table 7.1: *Failure and Fault types revealed*

Fault n.	**Fault type**	**Failure type**	**Failure classification**
Chart 2	Missing field setup	NullPointerException	Unknown Application Condition
Chart 4	Missing null check	NullPointerException	Bad Testing
Chart 5	Missing bounds check	IndexOutOfBoundsException	Bad Testing
Chart 9	Missing value consistency check	IllegalArgumentException	Combinatorial Explosion
Chart 11	Wrong variable use	Assertion violation	Combinatorial Explosion
Chart 13	Wrong argument used	IllegalArgumentException	Combinatorial Explosion
Chart 14	Missing null check	NullPointerException	Bad Testing
Chart 15	Missing null check	NullPointerException	Bad Testing
Chart 16	Missing field setup	NullPointerException	Bad Testing
Chart 17	Missing value consistency check	IllegalArgumentException	Bad Testing
Chart 25	Missing null check	NullPointerException	Bad Testing
Chart 26	Missing null check	NullPointerException	Bad Testing
Lang 1	Missing input consistency check	NumberFormatException	Combinatorial Explosion
Lang 3	Missing input consistency check	Assertion violation	Combinatorial Explosion
Lang 7	Missing input consistency check	Assertion violation	Combinatorial Explosion
Lang 12	Missing bounds check	ArrayIndexOutOfBoundsException	Combinatorial Explosion
Lang 14	Missing input consistency check	Assertion violation	Unknown Application Condition
Lang 16	Missing input consistency check	NumberFormatException	Unknown Application Condition
Lang 17	Wrong loop instantiation	Assertion violation	Combinatorial Explosion
Lang 19	Missing bounds check	StringIndexOutOfBoundsException	Combinatorial Explosion
Lang 20	Missing null check	NullPointerException	Bad Testing
Lang 24	Missing input consistency check	Assertion violation	Combinatorial Explosion
Lang 27	Missing bounds check	StringIndexOutOfBoundsException	Combinatorial Explosion
Lang 36	Missing input consistency check	NumberFormatException	Bad Testing
Lang 39	Missing null check	NullPointerException	Unknown Application Condition
Lang 43	Missing loop advancement operator	OutOfMemoryError	Combinatorial Explosion
Lang 44	Missing input consistency check	StringIndexOutOfBoundsException	Bad Testing
Lang 47	Missing null check	NullPointerException	Combinatorial Explosion
Lang 51	Missing return statement	StringIndexOutOfBoundsException	Combinatorial Explosion
Lang 58	Missing input consistency check	NumberFormatException	Unknown Application Condition
Lang 59	Wrong variable use	ArrayIndexOutOfBoundsException	Bad Testing
Lang 61	Wrong variable use	ArrayIndexOutOfBoundsException	Bad Testing

faults were not discovered during the in-house testing phase. We observe that combinatorial explosion is the most common reason behind failures in the cases where our approach was successful (44% of the cases), bad testing also plays a prominent role (40%), we also observe a few instances where unknown application conditions were the cause of the failures (16%).

These results suggest that field tests are a powerful tool that allows to detect field-intrinsic faults (faults in the Combinatorial Explosion and Unknown Application Conditions categories are inherently difficult to detect with in-house testing) and to integrate the in-house testing activity (Bad Testing faults should ideally not be present if in-house testing has been properly performed, by we understand that in practice this is not always the case).

From the chart in Figure 7.1 we can see the percentage of failures that can be revealed with the proposed field testing approach for each failure type, compared to the total number of failures in the dataset. The results show that on the considered datasets field testing reveals 83% of faults that cause null pointer failures, 62% of faults that cause out of bounds failures, 44% of faults that cause input format failures and 12% of faults that fail with a wrong output value. Finally the only memory error failure in the two datasets was also revealed by our field testing approach.

The distribution of the failures somehow reflects their difficulty to be revealed by a testing procedure: while null pointer exceptions do not require a specific oracle, value failures do, in particular more knowledge about the tested functionality is required to understand if it returns a correct or a wrong value. Field oracles suffer from the mandatory weakening that field testing implies, hence the low result on detection of value failures.

7.4 Qualitative Analysis

In this section we analyze two concrete applications of the proposed field testing approach, in Section 7.4.1 the adapted test case uses a general oracle to check the result correctness, while in Section 7.4.2 we can see an example of stronger oracle that captures a fault that results in a wrong return value.

7.4.1 Null Pointer Failure

This section explains how we applied our field testing approach to JFreeChart to detect the fault number 2 in the Defects4J dataset. In the test suite available with Defects4J there are two failing test cases for this fault:

- `DatasetUtilitiesTests.testBug2849731_2`
- `DatasetUtilitiesTests.testBug2849731_3`

these two test cases fail with a `NullPointerException` when the method `getLowerBound` of the class `DatasetUtilities` is called. The problem here is that the method does not handle properly `Double.NaN` values when calculating lower and upper bounds of an interval. These two test cases are written ad hoc to reveal this fault, we can see one in Listing 7.1. In this case the dataset contains a series that has some `Double.NaN` that need to be processed when calling `DatasetUtilities.iterateDomainBounds`, triggering the failure.

```
public void testBug2849731_2() {
        XYIntervalSeriesCollection d = new
            XYIntervalSeriesCollection();
        XYIntervalSeries s = new XYIntervalSeries("S1");
        s.add(1.0, Double.NaN, Double.NaN, Double.NaN, 1.5,
            Double.NaN);
        d.addSeries(s);
        Range r = DatasetUtilities.iterateDomainBounds(d);
        assertEquals(1.0, r.getLowerBound(), EPSILON);
        assertEquals(1.0, r.getUpperBound(), EPSILON);
        . . .
}
```

Listing 7.1: *DatasetUtilitiesTests.testBug2849731_2()*

```
public void testFindDomainBounds() {
        XYDataset dataset = createXYDataset1();
        Range r = DatasetUtilities.findDomainBounds(dataset);
        assertEquals(1.0, r.getLowerBound(), EPSILON);
        assertEquals(3.0, r.getUpperBound(), EPSILON);
}
```

Listing 7.2: *DatasetUtilitiesTests.testFindDomainBounds()*

```java
private XYDataset createXYDataset1() {
        XYSeries series1 = new XYSeries("S1");
        series1.add(1.0, 100.0);
        series1.add(2.0, 101.0);
        series1.add(3.0, 102.0);
        XYSeries series2 = new XYSeries("S2");
        series2.add(1.0, 103.0);
        series2.add(2.0, null);
        series2.add(3.0, 105.0);
        XYSeriesCollection result = new XYSeriesCollection();
        result.addSeries(series1);
        result.addSeries(series2);
        result.setIntervalWidth(0.0);
        return result;
}
```

Listing 7.3: `DatasetUtilitiesTests.createXYDataset1()`

The two test cases that expose the fault are not the only two that call the faulty method, for example the test case in Listing 7.2 also calls it. This test case does not expose the fault because the `XYDataset` object that it uses (created by `createXYDataset1()`, shown in Listing 7.3) does not cover the faulty code present in the `getLowerBound` method. The problems is that the value `Double.NaN`, that makes the program fail with a `NullPointerException`, is never used when initializing the `XYSeries` objects.

In our approach, instead of statically creating objects at testing time the field test gets the input object from regular application usage. In this case the test needs the `XYDataset` object which is the argument of the `DatasetUtilities.iterateDomainBounds` method. The idea is to extract this object from a normal use of the JFreeChart library, and use it in the field test case.

Listing 7.4 shows the complete code of the aspect that intercepts any call to
`DatasetUtilities.iterateDomainBounds`, allowing the use of its `XYDataset` argument in the adapted test case. Listing 7.5 shows the adapted test case.

```
public aspect IterateDomainBoundsTester {

 pointcut callIterateDomainBounds(XYDataset dataset, boolean
     includeInterval): call(*
     DatasetUtilities.iterateDomainBounds(XYDataset, boolean)) && args
     (dataset, includeInterval) && if(!TestFlags.testing);

 before(XYDataset dataset, boolean includeInterval):
     callIterateDomainBounds(dataset, includeInterval){

  TestFlags.testing = true;

  TestStorage.dataset = dataset;

  Logger logger = SingletonLogger.getInstance();

  JUnitCore jUnitCore = new JUnitCore();
  Result result = jUnitCore.run(DatasetUtilitiesTestField.class);

  logger.info("test class: DatasetUtilitiesTestField");
  logger.info("ran: " + result.getRunCount() + " failed: " +
      result.getFailureCount());
  List<Failure> failures = result.getFailures();
  if(!failures.isEmpty()) {
   for(Failure f : failures) {
    logger.info(f.getTrace());
   }
  }

 TestFlags.testing = false;

}
```

Listing 7.4: *IterateDomainBoundsTester*

```
public void testFindDomainBounds() {
        assumeTrue(TestStorage.dataset.getSeriesCount() >= 1);
        XYDataset dataset = TestStorage.dataset;
        Range r = DatasetUtilities.iterateDomainBounds(dataset);
        assertNotNull(r.getLowerBound());
        assertNotNull(r.getUpperBound());
}
```

Listing 7.5: *DatasetUtilitiesTestField.testFindDomainBounds()*

To test that the approach works and indeed reveals the fault when a specific `XYDataset` instance is observed, we used the input data from the failing test case from the original test suite in a simple application to trigger the field testing process. Listing 7.10 shows a simple method that can be used to trigger the field testing process.

```java
public void trigger() {
        XYIntervalSeriesCollection d = new
            XYIntervalSeriesCollection();
        XYIntervalSeries s = new XYIntervalSeries("S1");
        s.add(1.0, Double.NaN, Double.NaN, Double.NaN, 1.5,
            Double.NaN);
        d.addSeries(s);
        Range r = DatasetUtilities.iterateDomainBounds(d);
}
```

Listing 7.6: *UsageJFree2.trigger()*

7.4.2 Value Failure

This section explains how we applied our field testing approach to `apache.commons.lang` to detect the fault number 14 in the Defects4J dataset. In the original test suite the following test case fails:

- `StringUtilsEqualsIndexOfTest.testEqualsOnStrings`

this test case fails with an `AssertionFailedError`. The assertion that fails checks that the `StringUtils.equals` method returns true when comparing a `CharSequence` object and a `StringBuilder` object casted to `CharSequence`, both containing the string *foo*. The problem here is that the `equals` method does not discriminate between instances of `String` and `CharSequence` when checking for equality, hence it returns false where the correct return value is true. This is due to the `StringUtils.equals` method not checking if the objects to be compared are instances of the `String` class. We can see in Listing 7.7 the assertion that reveals the fault.

```java
public void testEquals() {
 final CharSequence fooCs = "foo";
 ...
 assertTrue(StringUtils.equals(fooCs, (CharSequence) new
     StringBuilder("foo")));
 ...
    }
```

Listing 7.7: *StringUtilsEqualsIndexOfTest.testEquals()*

```java
    public void testEqualsOnStrings() {
        assertTrue(StringUtils.equals(null, null));
        assertTrue(StringUtils.equals(FOO, FOO));
        assertTrue(StringUtils.equals(FOO, new String(new char[] {
            'f', 'o', 'o' })));
        assertFalse(StringUtils.equals(FOO, new String(new char[] {
            'f', 'O', 'O' })));
        assertFalse(StringUtils.equals(FOO, BAR));
        assertFalse(StringUtils.equals(FOO, null));
        assertFalse(StringUtils.equals(null, FOO));
        assertFalse(StringUtils.equals(FOO, FOOBAR));
        assertFalse(StringUtils.equals(FOOBAR, FOO));
    }
```

Listing 7.8: *StringUtilsEqualsIndexOfTest.testEqualsOnStrings()*

The test case that exposes the fault, however, is not the only one that calls the faulty method (`StringUtils.equals`), for example the test case in Listing 7.8 also calls it. This test case does not expose the fault because the `StringUtils.equals` method never uses a `CharSequence` object as input.

```java
public void testCustomCharSequence() {
 CharSequence cs1 = TestStorage.cs1;
 assertTrue(StringUtils.equals(cs1, new StringBuilder(cs1)));
}
```

Listing 7.9: *StringUtilsTestField.testCustomCharSequence()*

```java
public void trigger() {
        StringUtils.equals("foo","foo");
}
```

Listing 7.10: *UsageLang14.trigger()*

Listing 7.11 shows the complete code of the aspect that intercepts any call to `StringUtils.equals`, extracts the `CharSequence` argument and starts the adapted test case.

```
public aspect StringUtilsTester {

 pointcut callEquals(CharSequence cs1, CharSequence cs2): call(*
     StringUtils.equals(CharSequence, CharSequence)) && args(cs1, cs2)
     && if(!TestFlags.testing)

 before(CharSequence cs1, CharSequence cs2): callEquals(cs1, cs2){

  TestFlags.testing = true;

  TestStorage.cs1 = cs1;
  TestStorage.cs2 = cs2;

  Logger logger = SingletonLogger.getInstance();

  JUnitCore jUnitCore = new JUnitCore();
  Result result =
      jUnitCore.run(StringUtilsEqualsIndexOfTestField.class);

  logger.info("test class: StringUtilsEqualsIndexOfTestField");
  logger.info("ran: " + result.getRunCount() + " failed: " +
      result.getFailureCount());
  List<Failure> failures = result.getFailures();
  if(!failures.isEmpty()) {
   for(Failure f : failures) {
    logger.info(f.getTrace());
   }
  }

  TestFlags.testing = false;

}
```

Listing 7.11: *StringUtilsTester*

As we can see from the field test case in Listing 7.9, Line 3, the test case uses the first argument of the captured `StringUtils.equals` call and compares it to itself, so that the `StringUtils.equals` method in the test case must return true. To test that the approach works and indeed reveals the fault when a specific `CharSequence` instance is observed, we used the input data from the failing test case in a simple application to trigger the field testing process. Listing 7.10 shows a simple method that can be used to trigger the field testing process.

7.5 Threats to Validity

We applied our field testing approach to two widely used Java libraries. Although this gives an initial estimate of the effectiveness of the approach, its application to other domains is needed to generalize the results. Efficiency also needs to be evaluated, trying to understand how impactful is the approach on the application performance; this evaluation might be also helpful to understand when field tests can be safely run without causing noticeable overhead (e.g. exploit idle cpu time to run tests).

Isolation is another part of the approach that needs further work. The isolation of the test cases is now granted by construction, but the implementation of a completely isolated test layer would allow the approach to further exploit the field environment for testing. Several techniques can be exploited to achieve test isolation: process forking [27, 52], snapshotting [15], rollback mechanisms [23, 42], and the implementation of test interfaces [41]. Each isolation technique targets a specific system and has advantages and disadvantages, so it is important to asses which one integrates best with the proposed approach.

To test our approach and check if it can reveal the failures in the dataset, we used the test input from the faulty test cases that we excluded from the original test suite. It is not clear how likely it would be to encounter such inputs in the field, a potential way to assess this likelihood could be deploying the proposed testing approach in multiple real usage scenarios and observe the results after a given amount of time.

Chapter 8

Conclusion

In this Ph.D. thesis work we focus on the problem of testing applications in the field. We performed a study to characterize field failures and the reasons why they are hard to detect with traditional in-house testing. The study led to the definition of the concept of field-intrinsic faults as faults inherently hard to detect in-house and more effectively detectable in the field. We report our findings about the high frequency of field-intrinsic faults in the analyzed bug reports (field-intrinsic faults represent 70% of the analyzed field faults), obtaining initial evidence that there is a relevant number of faults that cannot be effectively addressed in-house and should be addressed directly in the field. This analysis could be extended with a larger fault base and replicated in different domains, to further understand the problem of applications failing in the field.

To tackle the problem of testing applications in the field, we designed an architecture that orchestrates the field testing process and an approach to write and execute field test cases.

The proposed architecture has a client-server structure that can be applied to different software domains, it allows to keep track of different field configurations and can use them to understand when to trigger the field testing process. The client instances can execute test cases directly in the field, on different instances of the same application, and can rely on the server when field testing cannot be performed.

Finally, we studied how to write test cases that take input data from the field, check it for consistency, exploit it to interact with the target software system and use a general test oracle to check the results. We built the framework that allows to execute these test cases at runtime and evaluate the testing approach on 91 real software faults that were discovered in the field. The results show that 31 (35%) of the faults could have been revealed using the proposed technique.

The proposed field testing approach and the obtained results open the possibility for further research in the context of field testing, that should aim to tackle the different challenges that this area presents and to better define how the testing process of software applications should continue in the field. Future directions for this work include extending the field testing approach by implementing a framework that deals with the test isolation, that is now granted by construction, and by further improving the test oracles, that need to be precise enough to detect failures, being at the same time general enough to work with inputs from the field.

Bibliography

[1] Number of different android devices. https://opensignal.com/reports/2015/08/android-fragmentation/, 2015.

[2] The cbs app for streaming the super bowl is crashing and burning. http://uk.businessinsider.com/cbs-app-streaming-super-bowl-is-not-working-2016-2, 2016.

[3] Apache open office. https://www.openoffice.org/, 2017.

[4] Apache openoffice bugzilla. https://bz.apache.org/ooo/, 2017.

[5] Eclipse bugzilla. https://bugs.eclipse.org, 2017.

[6] Eclipse git plugin. eclipse.org/egit/, 2017.

[7] Eclipse java persistence plugin. http://www.eclipse.org/eclipselink/, 2017.

[8] Eclipse subversive svn plugin. http://www.eclipse.org/subversive/, 2017.

[9] Hadoop. http://hadoop.apache.org, 2017.

[10] Jira. https://www.atlassian.com/software/jira, 2017.

[11] Nuxeo. https://www.nuxeo.com/, 2017.

[12] Android managed profiles. https://source.android.com/devices/tech/admin/managed-profiles, 2018.

[13] Apache commons lang. https://commons.apache.org/proper/commons-lang/, 2018.

[14] The aspectj project. https://www.eclipse.org/aspectj/, 2018.

[15] Checkpoint-restore in userspace (CRIU). https://criu.org/Main_Page, 2018.

[16] Jfreechart. http://www.jfree.org/jfreechart/, 2018.

[17] Junit. https://junit.org/junit5/, 2018.

[18] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[19] H. Aysan, S. Punnekkat, and R. Dobrin. Error modeling in dependable component-based systems. In *Computer Software and Applications*, pages 1309–1314. IEEE, 2008.

[20] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering (TSE)*, 41(5):507–525, 2015.

[21] E. T. Barr, M. Harman, P. McMinn, and M. Shahbaz. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering (TSE)*, 41(5):507–525, 2015.

[22] D. Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

[23] J. Bobba, W. Xiong, L. Yen, M. Hill, and D. Wood. Stealthtest: Low overhead online software testing using transactional memory. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 146–155. IEEE, 2009.

[24] A. Bondavalli and L. Simoncini. Failure classification with respect to detection. In *Workshop on Future Trends of Distributed Computing Systems*, pages 47–53, 1990.

[25] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering (TSE)*, 18(11):943–956, 1992.

[26] M. Cinque, D. Cotroneo, Z. Kalbarczyk, and R. Iyer. How do mobile phones fail? a failure data analysis of symbian os smart phones. In *International Conference on Dependable Systems and Networks*, pages 585–594, 2007.

[27] H. Dai, C. Murphy, and G. Kaiser. Configuration fuzzing for software vulnerability detection. In *International Conference on Availability, Reliability, and Security (ARES)*, pages 525–530. IEEE, 2010.

[28] C. F. Fan, S. Yih, W. H. Tseng, and W. C. Chen. Empirical analysis of software-induced failure events in the nuclear industry. *Safety Science*, 57:118 – 128, 2013.

[29] R. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-oriented software development*. Addison-Wesley Professional, 2004.

[30] L. Gazzola, L. Mariani, F. Pastore, and M. Pezzé. An exploratory study of field failures. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 67–77. IEEE, 2017.

[31] O. Goh and Y. Lee. Schedulable online testing framework for real-time embedded applications in vm. In *International Conference on Embedded and Ubiquitous Computing*, pages 730–741. Springer, 2007.

[32] A. Gonzalez-Sanchez, E. Piel, H. Gross, and A. van Gemund. Runtime testability in dynamic high-availability component-based systems. In *International Conference on Advances in System Testing and Validation Lifecycle*, pages 37–42. IEEE, 2010.

[33] M. Greiler, H. Gross, and A. van Deursen. Evaluation of online testing for services: a case study. In *International Workshop on Principles of Engineering Service-Oriented Systems*, pages 36–42. ACM, 2010.

[34] M. Hamill and K. Goseva-Popstojanova. Common trends in software fault and failure data. *IEEE Transactions on Software Engineering (TSE)*, 35(4):484–496, 2009.

[35] M. Hamill and K. Goseva-Popstojanova. Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system. *Software Quality Journal*, 23(2):229–265, 2014.

[36] J. Hänsel and H. Giese. Towards collective online and offline testing for dynamic software product line systems. In *International Workshop on Variability and Complexity in Software Design (VACE)*, pages 9–12. IEEE, 2017.

[37] R. Just, D. Jalali, and M. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440. ACM, 2014.

[38] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es):154, 1996.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[40] T. King, , A. Allen, R. Cruz, and P. Clarke. Safe runtime validation of behavioral adaptations in autonomic software. In *International Conference on Autonomic and Trusted Computing*, pages 31–46. Springer, 2011.

[41] M. Lahami, M. Krichen, and M. Jmaiel. Safe and efficient runtime testing framework applied in dynamic and distributed systems. *Science of Computer Programming*, 122:1–28, 2016.

[42] J. R. Larus and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 1(1):1–226, 2007.

[43] Y. Lei, R. Kacker, D. Kuhn, V. Okun, and J. Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *International Conference on Software Testing, Verification and Reliability (STVR)*, 18(3):125–148, 2008.

[44] M. Leszak, D. E. Perry, and D. Stoll. Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software*, 61(3):173 – 187, 2002.

[45] J. Morán, A. Bertolino, C. de la Riva, and J. Tuya. Towards ex vivo testing of mapreduce applications. In *International Conference on Software Quality, Reliability and Security (QRS)*, pages 73–80. IEEE, 2017.

[46] C. Murphy, G. Kaiser, I. Vo, and M. Chu. Quality assurance of software applications using the in vivo testing approach. In *International Conference on Software Testing Verification and Validation (ICST)*, pages 111–120, 2009.

[47] V. Neves, M. Delamaro, and P. Masiero. An environment to support structural testing of autonomous vehicles. In *Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 19–24. IEEE, 2014.

[48] V. Neves, M. Delamaro, and P. Masiero. Combination and mutation strategies to support test data generation in the context of autonomous vehicles. *International Journal of Embedded Systems*, 8(5-6):464–482, 2016.

[49] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[50] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 55–64, 2002.

[51] M. Pezzé and M. Young. Software testing and analysis: Process, principles and techniques. 2008.

[52] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[53] T. Roehm, S. Nosovic, and B. Bruegge. Automated extraction of failure reproduction steps from user interaction traces. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 121–130, 2015.

[54] M. Roper. Estimating fault numbers remaining after testing. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 272–281. IEEE, 2013.

[55] O. Sammodi, A. Metzger, X. Franch, M. Oriol, J. Marco, and K. Pohl. Usage-based online testing for proactive adaptation of service-based applications. In *Computer Software and Applications Conference (COMPSAC)*, pages 582–587. IEEE, 2011.

[56] D. Yuan, Y. Luo, X. Zhuang, G. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, pages 249–265, 2014.