European Research Council
Established by the European Commission

Self-assessment Oracles for Anticipatory Testing

TECHNICAL REPORT: TR-Precrime-2021-02

Antonia Bertolino[a], Pietro Braione[b], Guglielmo De Angelis[c], Luca Gazzola[b], Fitsum Kifetew[d], Leonardo Mariani[b], Matteo Orrù[b], Mauro Pezzè[e], Roberto Pietrantuono[f], Stefano Russo[f], Paolo Tonella[e]
[a]*ISTI-CNR*, [b]*UNIMIB*, [c]*IASI-CNR*, [d]*FBK*, [e]*USI*, [f]*UNINA*

A Survey of Field-based Testing Techniques

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

Technical report num.: TR-Precrime-2021-02
Date: June, 2021
Organization: Università della Svizzera italiana
Authors: Antonia Bertolino[a], Pietro Braione[b], Guglielmo De Angelis[c], Luca Gazzola[b], Fitsum Kifetew[d], Leonardo Mariani[b], Matteo Orrù[b], Mauro Pezzè[e], Roberto Pietrantuono[f], Stefano Russo[f], Paolo Tonella[e]
[a]ISTI-CNR, [b]UNIMIB, [c]IASI-CNR, [d]FBK, [e]USI, [f]UNINA
Dissemination level: Public
Revision: 1.0

Disclaimer:
This Technical Report is a pre-print of the following publication:

Antonia Bertolino, Pietro Braione, Guglielmo De Angelis, Luca Gazzola, Fitsum Kifetew, Leonardo Mariani, Matteo Orrù, Mauro Pezzè, Roberto Pietrantuono, Stefano Russo, Paolo Tonella: *A Survey of Field-based Testing Techniques*. ACM Computing Surveys (CSUR), Vol. 54, No. 5, 2021

Please, refer to the published version when citing this work.

Università della Svizzera Italiana (USI)

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

Abstract

Field testing refers to testing techniques that operate in the field to reveal those faults that escape in-house testing. Field testing techniques are becoming increasingly popular with the growing complexity of contemporary software systems. In this paper, we present the first systematic survey of field testing approaches over a body of 80 collected studies, and propose their categorization based on the environment and the system on which field testing is performed. We discuss four research questions addressing *how* software is tested in the field, *what* is tested in the field, which are the *requirements*, and how field tests are *managed*, and identify many challenging research directions.

Contents

1 Introduction

Software testing involves a set of pervasive, critical, time-consuming and effort-demanding activities in the software lifecycle. It is widely practiced and extensively studied [85].

Testing activities are commonly conducted on a Software Under Test (SUT) during the development cycle to both reveal faults before deployment and study failures reported from the field. No matter whether testing activities aim to reveal development bugs or study field failures, they are commonly executed in the development environment, and we refer to such activities as *in-house* or *in-vitro* software testing. In-vitro testing is generally conducted independently from the production context, except for the failures reported from the field, which may trigger in-house testing activities. Indeed, field failures cannot be fully prevented, and may sometime lead to catastrophic consequences. A recent study by Gazzola *et al.* [39] identifies several categories of field failures, and provides empirical evidence of the unavoidability of failures in the field even for mature and well-tested software systems.

The impossibility of dealing with all the faults using classic in-house testing approaches raised interest in testing software systems in the field, by crossing the border between in-house validation and field execution [7]. As we illustrate in Fig. 1, this trend of moving the testing activities from the laboratory towards the production environment can be actualized in different nuances. Test cases can be executed directly in the field on the same instance of the software used in production, which we call *online testing* (rightmost flow of activities in Fig. 1), or on a separated instance still running in production, called *offline testing* (middle flow), or even in-house but on data collected from the field, called *ex-vivo testing* (leftmost flow).

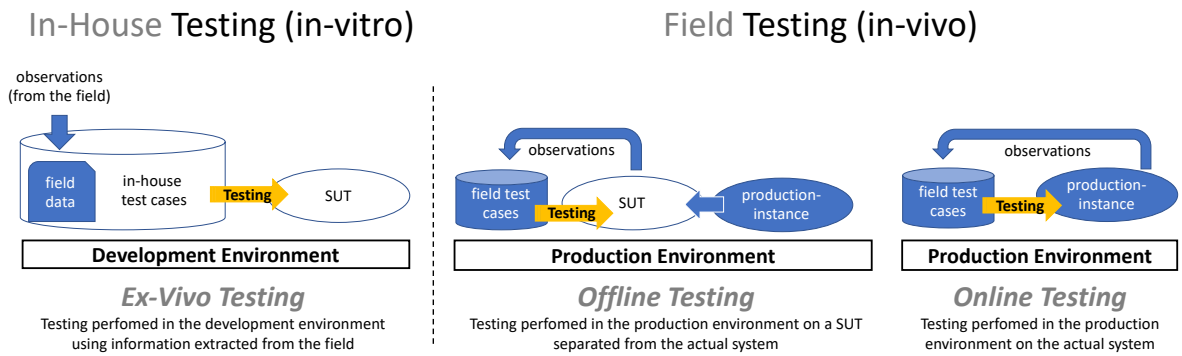


Figure 1: Classes of field testing approaches

Classic in-house (in-vitro) testing is a well understood discipline with studies that span many decades: The first specialized workshop dates back to the mid seventies.[1] Research in field testing has emerged fairly recently and has not been comprehensively surveyed as a discipline yet.

Field testing was first considered an opportunity to deal with failures that were hard or too costly to reproduce in the laboratory, with a few studies in the nineties addressing autonomic systems [51, 100], and real-time issues [98]. It attracted then steady interest in the early years of the first decade of this century, with a sudden burst of results from 2007 on, mostly pushed by the advent of service-oriented architectures, which lacked centralized control for testing purposes.

Nowadays, the emergence of more and more agile and distributed paradigms of development, towards the view of a *continuous software engineering* discipline [35], emphasizes the need for a *continuous testing* mindset, making developers realize that testing must continue after deployment, and that no clear boundary can be set between development and operation [7]. Such a need, in combination with the increasing dynamism and pervasiveness of software, as observed, for instance, in Internet of Things, Systems of Systems, and Cyber-Physical systems, led to a relevant set of approaches and frameworks for field testing, although not well contextualized yet. In particular, while inheriting many of the problems and solutions of its "traditional" counterpart, field testing obviously introduces new challenges, for example in terms of isolation from side-effects, security preservation, controllability and observability of the test executions, among others. For these reasons, a systematic study of the literature is required to rise awareness of the topic and provide evidence for the need of further research.

To answer such need, in this paper we provide a comprehensive survey of the state-of-the-art of ex-vivo, offline and online field testing approaches. We present the results of a systematic analysis of the scientific literature that identified 80 distinct relevant studies since 1989. We discuss the characteristics of the different

[1] The ISSTA community refers back to the former TAV community with roots in the 1975 Workshop on Currently Available Program Testing Tools.

approaches, and propose a taxonomy based on the environment and the system where the test cases are generated and executed. We distinguish between approaches that address *functional* and *non-functional* faults, and observe that the relatively few ex-vivo and offline field testing approaches address functional faults, while the many online field testing approaches span from functional to non-functional faults [2].

We survey the approaches for field testing, and for each class of approaches we systematically discuss their characteristics based on four main research questions that guide our effort throughout this survey. In particular, we consider *how* software is tested in the field, *what* is tested in the field, the *requirements* to successfully execute the tests in the field, and the *management* of the field tests.

The remainder of the paper is organized as follows. Section 2 discusses the methodology of the survey, providing details about the process that we followed to select the relevant studies and the corresponding statistics. Section 3 introduces the key concepts and the terminology that we use in the paper, and frames the boundaries of our analysis. Section 4 introduces the research questions that we address in this paper, and that we discuss in Sections 5–9. Section 10 summarizes the state of the art in field testing, indicates relevant open research directions and presents guidelines for practitioners. Section 11 discusses threats to validity. Section 12 provides final remarks.

2 Methodology

In this section, we present the methodology that we followed to identify the relevant papers in line with the best practices by Kitchenham *et al.* [56].

2.1 Selection process

The aim of this survey is to provide a comprehensive summary of the scientific literature on field testing and propose a taxonomy of the approaches. Our research questions, which we describe later on after we have introduced the needed background and terminology, are broad and inclusive. Accordingly, in our search strategy we aimed at a search string wide enough to represent all the many ways researchers may indicate field testing related work, as detailed below:

(1) Initial search and first-stage filtering. An initial set of papers was selected by searching the *SciVerse Scopus* digital library with the following search string. We selected all papers whose title, abstract or keywords match any of the keywords in the query:

```
(Runtime testing OR Online testing OR On-line testing OR Dynamic testing OR Adaptive testing OR Field
testing OR On-demand testing OR In-vivo testing OR Ex-vivo testing) AND software
OR
(Runtime software testing OR Online software testing OR On-line software testing OR Dynamic software
testing OR Adaptive software testing OR Field software testing OR On-demand software testing OR In-vivo
software testing OR Ex-vivo software testing).
```

The initial search produced a set of 1,238 studies. Due to the conservative query, the initial set of papers included many papers not related to computer science, and artifacts of different nature, research articles, editorials, standards, and welcome messages. We pruned both papers clearly not related to computer science, and irrelevant artifacts, such as editorials, standards, welcome messages, by manually inspecting titles and abstracts, and we obtained a set of 434 studies.

(2) Selection criteria. We refined the obtained set of papers with inclusion and exclusion criteria, to retain scientific studies about software field testing, and eliminate papers that address neither software nor field testing. We retained papers that satisfy all the following inclusion criteria:

- *Inclusion Criterion 1*: studies targeting the definition, application or experimentation of software field testing solutions.
- *Inclusion Criterion 2*: studies subject to peer review.
- *Inclusion Criterion 3*: studies written in English.

and discarded papers that meet at least one of the following exclusion criteria:

[2] In this paper we use the terms *failures*, to indicate executions that lead to wrong results, and *faults*, and sometime the common jargon term *bugs*, to indicate issues in the code that may cause the system to fail under specific execution conditions, in line with the IEEE standard terminology.

- *Exclusion Criterion 1*: studies proposing field testing solutions not related to software systems, such as firmware or hardware testing, including processors, systems-on-chip, FPGA, and controllers.
- *Exclusion Criterion 2*: studies purely focusing on techniques other than testing, most notably we exclude works centered on: runtime verification [68], or debugging, even approaches for debugging in the field such as [65], or continuous experimentation via user surveys [90] and A/B testing [57], or usability evaluation [49, 97].
- *Exclusion Criterion 3*: studies that move testing away from the developer's laboratory in order to improve scalability and elasticity, but with no specific focus on field operation: for example, we exclude works proposing Testing-as-a-Service and Cloud testing solutions [11].
- *Exclusion Criterion 4*: studies focusing on the interleaving/adjustement of test generation to test execution (which sometimes is also referred in literature as "online" or "adaptive" testing, keywords included in our search), but have no relation with the field, e.g., on-the-fly model based testing [96], or those works that dynamically adjust the testing strategy using testing results, such as [19].
- *Exclusion Criterion 5*: studies about in-house operational profile-based testing [91]: although operational testing refers to how the system is used in the field, this topic has been the subject of an extensive literature that has evolved independently from the notion of testing in the field.
- *Exclusion Criterion 6*: studies about crowdsourced testing: testing performed by the crowd on released software could be considered as a specific form of testing in the field, however we decided not to include them in this study, also because for this topic we can refer the reader to a recently appeared systematic review of literature [4].
- *Exclusion Criterion 7*: secondary or tertiary studies (e.g., systematic literature reviews and surveys).
- *Exclusion Criterion 8*: studies not available as full-text.

In a first pass, the authors of this survey independently assessed the 434 papers, and classified them as 'Included', 'Excluded' and 'Unclear', based on the inspection of title, abstract and publication venue. We excluded 334 papers, included 34, and identified 66 unclear papers for further analysis. Then, we collaboratively classified the 66 papers, by reading the full papers, and discussing them in dedicated conference calls. We included 14 more papers, ending up with 48 selected papers.

(3) Snowballing. We completed the selection process with a snowballing procedure. We applied a full *backward snowballing*, by considering all the references included in the analyzed studies, and adding further relevant studies, provided they were indexed by at least one of these major digital libraries: *SciVerse Scopus*, *IEEEXplore*, and *ACM DL*. We conducted a partial *forward snowballing* starting from the most popular papers selected so far. In particular, we selected both the 10% most cited papers and the top 10% of the papers with the highest number of *normalized* citations (i.e., citation/year), identifying a total of 12 highly popular studies. We considered all the papers that cite at least one of the identified popular studies, obtaining 71 possibly relevant studies. We pruned this set with the inclusion and exclusion criteria, and added 32 new papers to our set of papers.

The process produced a set of **80 studies for our survey.** During this process, we scheduled six plenary (physical or online) meetings in one year to define the include and exclude criteria, to discuss the studies, and to clarify and resolve doubtful cases.

2.2 Data extraction

We identified several dimensions for analyzing the selected papers, based on the research questions presented in Section 4. For each paper, we checked if the presented study could be classified according to each identified dimension, and when possible we produced such a classification. We describe the studies and the analyzed dimensions in Sections 5–9.

We tuned the analysis process by assigning a small set of papers (14/80) to multiple authors, to obtain redundant classifications for each paper, and then discussing the results of the classifications in a plenary meeting. In the plenary meeting we agreed on the semantics of each dimension, and the criteria for classifying the papers. We relied on the results of the plenary meeting to safely distribute the analysis of the remaining papers to subsets of authors who worked in parallel.

2.3 Descriptive Statistics

The data collection was performed in 2018. The search period was not constrained; the time period of the resulting papers, after filtering and classifications, was 1989-2017. Fig. 2a plots the selected studies by year and publication type. The figure indicates that a substantial activity on field testing started in 2002. Since then, an average of 4.68 studies per year were published, with more papers published in the last 10 years. Most of the considered studies are conference papers (51/80), followed by journals (19/80) and few workshop papers (8/80) and book chapters (2/80).

Fig. 2b details the main publication venues, reporting venues that hosted at least two studies. There is a considerable variety of publication venues, with 60 different venues for the 80 studies. It is worth to note the presence of venues like TSC (*Transactions on Service Computing*) and WEBIST (*International Conference on Web Information Systems and Technologies*), with several studies on testing of service-oriented architectures, where testing in-the-field is of particular relevance.

2.4 Replicability of the Study

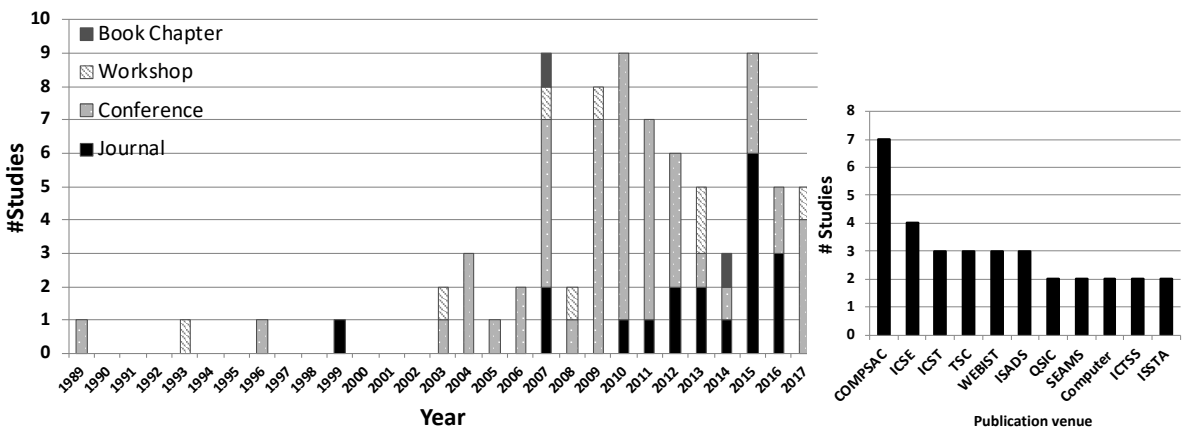
The outcome of our classification work as well as the list of all the 434 returned by the execution of the query string are made available for the interested researchers on the ACM Digital Library as supplemental online material of this manuscript.

3 Field Testing

This section introduces field testing. The analysis of the literature revealed no uniform and consistent way of referring to testing solutions in the field, thus in this section we define the terminology that we consistently use in the paper. While in-house testing refers to activities in controlled environments specifically arranged to support the testing activities themselves, field testing refers to a range of activities closely related to the production environment that we characterize first, before introducing field testing.

Definition 1 (Production Environment) *The production environment of an application is any environment where the application can be fully operational.*

Production environments include hardware, software, such as system software, libraries, middleware, application-level software, and any other element that may affect the behavior of an application, such as sensors, input devices and network components. As an example to help clarify the following definitions, we could think of a provider responsible for an e-commerce platform FooZon: we consider the case that while developing a new extended feature, they need to perform an extensive testing across differing configurations, as well as differing customer profiles. The same application can be in fact deployed and used in multiple production environments, for instance an app FooZon can be installed in millions of different



(a) Field testing studies per year and venue type

(b) Field testing studies by venue

Figure 2

devices with different settings, or the platform can be accessed directly from its web application, again using different browsers and versions.

The production environment of an application is a dynamically evolving entity whose characteristics change at different speeds. For example, hardware components tend to remain the same (e.g., a component might be upgraded only after several years to improve the hardware equipment of a server), configurations change relatively rarely (e.g., a user may decide to change some settings to accommodate some emerging needs), while contextual elements may change quickly (e.g., the battery level and network connectivity may change quickly for mobile apps).

Since an application can be installed and executed in many production environments, testing activities must take such a variety and heterogeneity of environments into consideration. The notion of *field* refers to the many production environments relevant for a software application.

Definition 2 (The Field) *The field of a software application is the set of all its (possible) production environments.*

Field testing techniques address the challenge of validating the behavior of applications in the field, that is in all (or more practically in most) of its production environments, while considering the dynamic evolution of their characteristics. Fig. 1 illustrates the main kinds of field testing, distinguishing between (right-side) testing activities performed in the field (aka in-vivo) and (left-side) activities performed in-house (aka in-vitro). Although in-house activities do not directly interact with the field, there are forms of testing that benefit from activities performed in the field.

Definition 3 (In-house/In-vitro Software Testing) *In-house (in-vitro) software testing indicates any type of software testing activities performed in a testing environment completely separated from the production environment.*

In-house software testing implies the presence of both the SUT (e.g., the considered example of the FooZon e-commerce platform) and a set of test cases that are executed against the SUT within the development environment (e.g., a user access, a user browsing different products, a user concluding a transaction, etc) . This survey considers only in-house testing activities influenced by information obtained *from the field*.

Definition 4 (Ex-vivo Software Testing) *Ex-vivo software testing indicates any type of software testing performed in-house using information extracted from the field.*

Ex-vivo testing is a specific type of in-house testing, which can be very useful for the testing of applications processing large amounts of structured data that are costly and/or difficult to build artificially. For example, instead of creating a series of accesses by different categories of customers trying to mimic many different operational contexts and client behaviors, the testers of FooZon could keep track of actual invocations coming from real transactions across different countries and relative to different types of products, and use these for more realistic and extensive testing of newly deployed versions. In our survey of literature, we found a few examples, as in the work by Morán at el. [74], where the authors employ data collected from actual user requests for the testing of MapReduce distributed data processing applications, thus overcoming the costs of building huge amount of data as those used in these applications. Another example is the work by Elbaum and Hardojo [34], where multiple techniques for enhancing an existing test suite by leveraging field data collected by profiling a web application under different strategies are assessed. Although ex-vivo testing does not imply running the test cases in the production environment, it is a relevant class of approaches in the scope of this survey because they extensively use field data. Fig. 1 represents the case of ex-vivo testing as the only kind of in-house testing activity relevant to this study.

Related to ex-vivo testing, there are a number of works that exploit failure/crash data from field executions for the generation of test cases that would help in reproducing the failures for in-house debugging (e.g., [50, 88, 52, 92, 29]). Such works aim at generating test cases that recreate the failures observed in the field, guided by information about the failure collected from the field. The failure information from the field could be obtained by explicit instrumentation of the program before deployment for the purpose of capturing essential data (e.g., [50, 52]). In other cases, crash data produced by the runtime environment (such as stack traces or core dumps) are used for the purpose of test generation (e.g., [88, 92, 29]).

The test cases generated in this context are primarily used to recreate reported field failures for in-house debugging. Their scope is limited to helping developers understand and fix reported field failures, and eventually verify that the fix resolves the reported issue. They are mentioned here as related work as they make use of field data for test generation, however they are not included in our survey as they are focused on debugging rather than the process of field testing.

Definition 5 (Field/In-vivo Testing) *Field (in-vivo) testing indicates any type of testing activities performed in the field.*

Field testing activities can be executed *offline* or *online*:

Definition 6 (Offline Testing) *Offline testing indicates field testing activities performed in the production environment on an instance of the SUT different from the one that is operational.*

The software used in offline testing activities and the operational software may be distinguished in different ways: for example, with reference to the FooZon platform, the provider could opportunistically duplicate the components responsible for completing purchase transactions, and in any field testing execution involving a purchase replace the invocation to the actual component with the dummy duplication that will not really conclude any transaction, thus limiting the degree of intrusiveness of field tests on the operational software system because no actual product is bought. Obtaining distinct instances of the SUT for off-line testing can be expensive, and may not guarantee perfect isolation of the testing process. For instance, a program can still interact with some environment elements, and test execution must be properly sandboxed to prevent side-effects, as illustrated in the work by Murphy *et al.* [75], who proposed an approach to duplicate the field environment by forking the process under test.

Definition 7 (Online Testing) *Online testing indicates field testing activities performed in the production environment on the actual software system.*

Online testing pushes forward the concept of field testing by directly testing the operational software system. Online testing might be preferable to offline testing in terms of the representativeness of the test outcome, but online testing can be extremely complex, since the testing activity might easily interfere with the normal activity of the software under test. Considering again FooZon, online field testing may entail launching some browsing executions pretending to be a customer and placing actual transactions, which are issued for testing purposes side-by-side with visits and transactions performed by real customers. In addition to possibly sustaining the real costs of purchases concluded while testing and their expeditions, this practice may also negatively impact customers experience, for example competing with them for the purchase of products of limited availability, or slowing down actual deliveries to customers.

Notwithstanding the possible side-effects, online testing is nowadays increasingly performed in practice by some providers, of course when the SUT has no safety-critical aspects. For example, in NetFlix the concept of Chaos engineering for online testing the robustness of their cloud services has been established a decade ago and still continues to be expanded [9].

In the next section we introduce the research questions addressed in the scope of this survey to study ex-vivo, offline and online testing approaches.

4 Research Questions

The *goal* of the survey is to characterize the existing research on field testing, in terms of *objectives* of field testing activities and *approaches* to achieve them. We want also to understand the practical implications of the proposed approaches, in terms of required field *resources* and overall *management* of field testing, as opposed to management of traditional in-lab testing. Correspondingly, we designed two research questions focused on the "how" (RQ1) and "what" (RQ2) dimensions, which cover respectively approaches and objectives. Furthermore, with RQ3 and RQ4 we investigate the resources and management activities required for field testing. Overall, this survey addresses four main research questions, all characterized by several dimensions.

RQ1 - *How* is software tested in the field? This research question addresses the type of testing activities performed in the field. We detail the general question in two sub research questions:

RQ1.1 - What are the approaches, fault types, test case generation and platforms used in field testing?

RQ1.2 - What are the source, strategy, triggers, resources and oracles used in field tests?

To answer this research question, we classify the approaches based on their category (*ex-vivo*, *offline*, *online*), the type of addressed faults (*functional*, *non-functional*), the strategy used to *generate* test cases, and the developed support platforms, and provide an organized map of field testing solutions (Section 5). We study *where* field test cases are generated (which may be a different location from where they are executed),

the events that may *trigger* their execution, the *resources* that are typically considered in the field testing process, and the used *oracles* (Section 6).

RQ2 - *What* is tested in the field? This research question addresses the software elements that are tested in the field. We consider the *test target*, for instance new or modified features, the *granularity* of the software elements under test, for instance single components or the system as a whole, and the *type of tested applications*, for instance mobile or server applications (Section 7).

RQ3 - What is *required* to execute tests in the field? This research question addresses the features that field testing solutions require to execute the test cases in the field. We consider four main dimensions: *monitoring*, which includes the techniques used to extract information about the behavior of the SUT; *isolation*, which includes mechanisms to guarantee that the execution of the field tests does not interfere –or has negligible interference with– the regular operation of the tested software; *privacy* and *security*, which include solutions to guarantee the privacy and the security of the users despite the activity performed in field testing (Section 8).

RQ4 - How is field testing *managed*? This research question addresses management and control aspects of field testing, also in view of possible evolution of the software under test. The overall problem of managing a testing session in the field is that it should have minimum impact on the end users and on the normal production activities. Moreover, key decisions to be made in the field include which tests to select for field execution and how to prioritize them for execution, under the assumption that there might not be enough resources to run all the available tests. Hence, we identify three dimensions: *test selection* to select the test cases to be executed in the field to validate changes; *test prioritization* to sort the test cases to be executed in the field to validate changes; and *test governance* to control the testing process and the involved stakeholders (Section 9).

5 RQ1.1: Approaches, Fault Types, Test Case Generation and Platforms

Table 1: Testing techniques

Testing approach		Test Generation Strategy: *Specification-based*	*Structure-Based*	*Fault-Based*	*Pre-existing*
ex-vivo	*functional*	*Mutation of Field Executions* [74, 78, 28, 79] *Test Suite Adaptation* [48, 47, 38]	*Profile Data* [34]		*Field Triggers* [72]
offline	*functional*	*IO Data Pattern* [81] *Metamorphic Relations* [10, 76]			*Built-in Tests* [94, 98, 75, 54, 55, 93] *Test planning and management* [43, 42, 77] *Adaptation and Reconfiguration* [62, 64]
	non-functional	*Security Specifications* [25, 24]			
online	*functional*	*Choreographies and Service-based specifications* [22, 2, 12, 6, 99, 5] *Finite-State Models* [71, 95, 18, 33, 20] *Metamorphic Relations* [21] *Graph grammars* [83]	*Event Interface* [101]	*Fault Injection* [102]	*Test planning and management* [60, 1, 59, 14, 27, 43, 42] *Adaptation and Reconfiguration* [44, 53, 64, 32, 62] *Isolation* [17]
	non-functional	*Stochastic Models of User Behavior* [89, 82] *Security Specifications* [12, 26, 13, 46] *Usability Models* [69] *Δ-grammars for QoS* [83] Timed-automata [71, 95]		*Fault Injection* [3, 102] *Operational Profile* [73]	*Adaptation and Reconfiguration* [70, 32, 67]

Table 2: Platforms for field testing

Offline Testing Platforms		Online Testing Platforms	
Evolution [66, 61, 58, 63] *Shadow Instances for V&V* [45, 40]	*SOA and Component-Based Testing* [105, 104, 8, 16, 81, 103]	*Built-In and Pre-Existing Test Cases* [81, 51, 100, 31, 30, 41] *SOA Online Testing* [80]	*Evolution* [23, 58, 61, 63, 37] *Distributed Quality Assurance* [87]

In this section, we analyze field testing techniques by considering the *field testing approach* and the *test case generation strategy* dimensions of the classification. We distinguish between approaches that address functional and non-functional faults (Table 1), and classify approaches based on the target platforms (frameworks and architectures) (Table 2).

The studies are not evenly distributed with respect to the *testing approaches*: We found few ex-vivo and offline field testing techniques, and many online field testing techniques. This indicates a strong interest

towards techniques that test exactly the operational application.

The nature of the *requirements* that are tested is not evenly distributed within each testing approach. Most ex-vivo and offline field testing approaches focus on functional requirements. Only two approaches address offline testing of security requirements to assess the security of software systems in the operational environment as early as possible [25, 24]. Online testing approaches address both functional and non-functional requirements.

Concerning the problem of *test case generation*, in field testing a test case has a broader scope than just the test input data. The actual novel challenges in generating field tests descend from identifying and reproducing with each test case the relevant interactions with the field, whereas a-priori we do not expect that new, ad-hoc techniques need to be specified for generating the test inputs data. We cluster techniques in four categories [85] depending on the information used to generate the test cases, as is also done for in-house testing: *(i) Specification-based* techniques derive test cases from formal or informal requirements specification; *(ii) Structure-based* techniques use structural information, mostly the code structure, to derive test cases; *(iii) Fault-based* techniques use models of potential faults (*fault models*) to derive test cases that address the faults represented in the fault model; *(iv)* Techniques working with *pre-existing* test cases exploit already available test cases. We observe that most commonly black-box techniques are used. Namely either test cases are generated from specifications, to suitably cover certain behaviors, or strategies are defined to execute test cases that are already available, for instance from field monitoring. Only few approaches derive test cases from structural information and fault-models.

The available platforms, as shown in Table 2, offer a range of solutions for both offline and online testing targeting several environments (e.g., service and component-based systems) and scenarios (e.g., continuous quality assurance and evolution).

The classification in Tables 1 and 2 provides a roadmap to researchers, who can easily understand the positioning of the individual studies with respect to the existing literature along the identified dimensions (testing approach, test generation strategy) and along the respective sub-dimensions. It is also potentially useful to practitioners, when they face a specific field testing problem. Thanks to the presented classification, they can find relevant research papers and tools that address the kind of field testing approach they want to implement (ex-vivo, offline, online) and the kind of test generation strategy they intend to adopt (structural, fault oriented or pre-existing tests). *The detailed descriptions of the individual approaches that appear in Tables 1 and 2 are available in a report that is part of the online supplemental material associated with the paper.*

5.1 RQ1.1: Findings

We conclude this section summarizing the key findings with respect to the type of addressed faults, the test case generation strategies, and the platforms proposed so far:

• *Approaches privilege functional versus non-functional faults, which are still under-investigated:* The majority of approaches address functional faults. The few approaches that address efficiency, security, reliability and usability, shed some initial light on a largely unexplored domain that calls for non-intrusive testing techniques to properly address non-functional properties.

• *Quality of service is a relatively well-studied quality attribute among non-functional aspects:* Many of the approaches that deal with non-functional properties address the relevant problem of predicting the QoS of applications executed in different and heterogenous production environments.

• *Specification-based test case generation approaches are by far the most studied approaches:* Many approaches rely on specifications to generate field test cases, leaving open the hard problem of generating test cases in absence of specifications, as in the many cases of systems that evolve beyond the initial specifications to adapt to emerging execution conditions and configurations.

• *Automation is still limited:* Many approaches rely on relevant human contribution and already available test cases, and automation of field testing is still limited.

6 RQ1.2: Source, Strategy, Triggers, Resources and Oracles

In this section, we discuss where field test cases are generated (Section 6.1), how field test cases are executed and triggered (Section 6.2), what resources field test cases require (Section 6.3), and which oracles validate the result of field test execution (Section 6.4).

6.1 Where Field Test Cases Are Generated

Field test cases may be generated and executed at different times and locations: in-house, in-house with field data, and in the field. Field test cases generated *in-house* are produced during development. Field test cases generated *in-house with field data* are produced during development by exploiting information observed in the field. Test cases generated *in-house* can be executed either in the development environment (ex-vivo testing) or in the production environment (in-vivo testing). Test cases generated *in-the-field* are generated in the production environment.

Table 3 shows the distribution of the different approaches with respect to environments for generating test cases. Relatively few approaches generate test cases in-house with field data (12%), most approaches generate test cases either in-house or in-the-field with an even distribution between the two sets (44% each). Test cases generated *in-house* address scenarios known to be possibly field-relevant but not completely available at design time yet, such as configurations that depend on dynamic information, for instance dynamically discovered services. Test cases generated *in-the-field* address scenarios that emerge and can be identified only in the field and cannot be identified in early development phases. Generating test cases in-house is easier than in-the-field, but produces test cases with a scope limited to at least partially predictable scenarios, while test cases generated in-the-field may address a wider set of scenarios.

Table 3: Place *where* tests are generated

Place	Number of papers	%
In-house	32	44%
In-house with field data	9	12%
In-the-field	32	44%

6.2 Test Strategy and Triggers

Testing strategies and triggers refer to the way approaches identify critical events that activate field test cases. Testing *strategies* are *proactive* if they primarily aim to anticipate failures that could occur in the production deployment, *reactive* if they primarily aim to manage the effect of field failures after their occurrence. We refer to the events that lead to the activation of field test cases as *triggers*. A trigger is any kind of event, scenario, or configuration whose occurrence leads to the execution of some field test cases.

Most studies of field testing strategies propose proactive strategies: 63 papers, that is 86% of the papers that deal with test strategies, propose proactive strategies; 5 papers, 7%, propose reactive strategies, that is, activate testing sessions *in-the-field* as a consequence of observed failures; 5 papers, 7%, support both strategies. For instance, Kawano et al.'s approach [51] proactively activates test cases when modules change, and reactively responds to failures.

Fig. 3 classifies the strategies according to the triggers they react to. The taxonomy identifies two main kinds of triggers, *IT Operation* and *Evolution*. IT Operation triggers are events that derive from some either internal or external operations of the system. Evolution triggers are events that derive from some dynamic transformation of the system or its components. Triggers are not exclusive, in fact some approaches can react to multiple triggers.

IT Operation triggers may be periodic or asynchronous events. While few studies focus on periodic events, the majority of approaches that react to IT operation triggers refer to asynchronous events. Some approaches react to asynchronous events internal to the SUT, related to custom events, unchecked exceptions, idle states, data and transmissions. Other approaches react to external triggers, related to test sessions, system policies, or external functionalities. Triggers related to test sessions derive from inputs by a client or another system/module, a QA team member, or the run-time infrastructure, and decouple the decision-making aspect from the technical field testing solution. Triggers-related system policies depend on explicit decision policies that can be defined by ether the service integrators or testers, or can be based on data observed in the field. Triggers related to external functionalities depend on the activation or usage of particularly relevant functionalities, and are defined either statically before or dynamically at runtime.

Evolution triggers react to the evolution of either the whole SUT or some components. System evolution triggers react to either dynamic reconfigurations or environmental changes, component evolution triggers react to discovering, binding, failing or removing components.

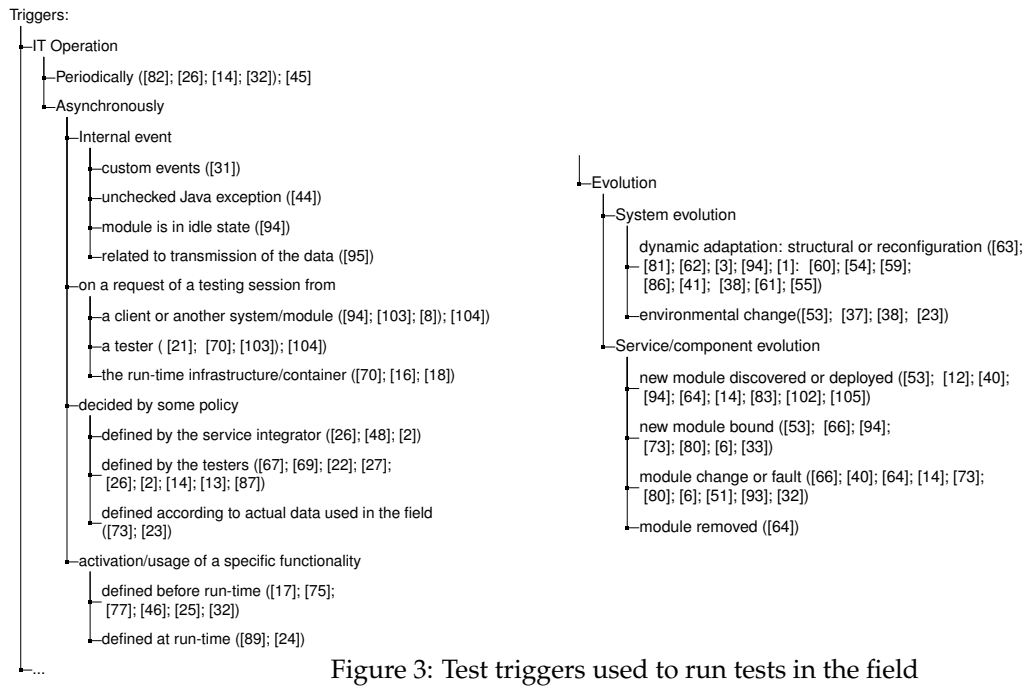


Figure 3: Test triggers used to run tests in the field

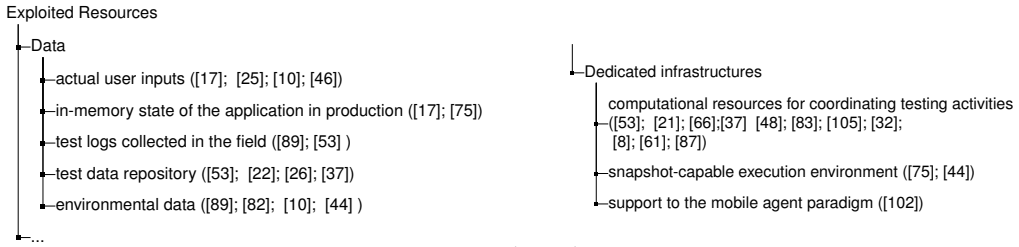


Figure 4: Exploited resources

6.3 Field Resources

In-vivo testing approaches require the availability of different *resources* in the field. In this section, we overview both the required field data and the infrastructures dedicated to field testing. We discuss the resources required to isolate executions of field tests in Section 8.2.

Fig. 4 shows both the kinds of field data and dedicated infrastructures that different approaches require. Several approaches rely on the data obtained from production. Bobba et al. [17] exploit *user inputs* to detect failures, Dai et al. [25] exploit *user inputs* to detect security vulnerabilities introduced by changes in the configurations. Bell et al. [10] mutate user inputs and observed outputs, leveraging metamorphic testing or weak-mutation strategies, to produce new test cases, Hui et al. [46] mutate inputs and outputs to reveal security vulnerabilities.

Other approaches rely on the *in-memory state* of the application in production. Bobba et al. [17] and Murphy et al. [75] exploit the in-memory state to discover faults hard to reveal in-house. Some works exploit data from the *logs* or from the verdicts collected in the field while executing in-vivo testing sessions. Sammodi et al. [89] use such information to predict adaptations, while King et al. [53] detect symptoms of undesirable SUT states. Many field testing approaches require specific data, and rely on dedicated *test data repositories* to store regression test cases [22], authentication/authorization cookies [26], or more generally mutable test artefacts so as to keep them consistent with the current operating conditions [37]. Some approaches take advantage of information on both the *environment* and the execution context. For instance, Gu et al. [44] infer useful information for testing from the state of the hosting virtual machine.

Field testing approaches may require additional *infrastructures* that are not part of the SUT to be available *in-vivo*. A common case is the request for *computational resources for coordinating* the testing activities, necessary when multiple components are involved in field testing. While approaches based on replicas use either *snapshot-capable execution environments* to simplify replica management [75, 44] or *agent-based paradigm* to migrate components between hosts [102].

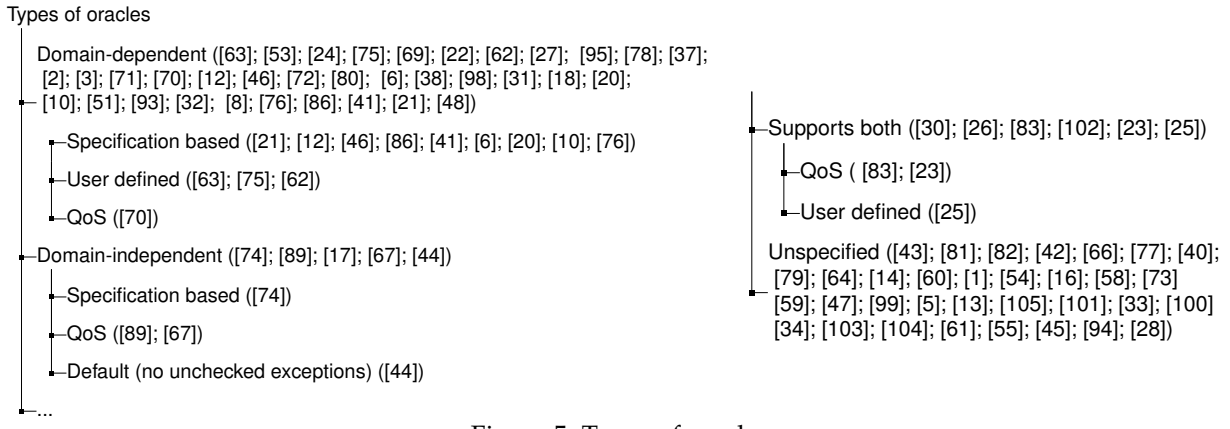


Figure 5: Types of oracles

6.4 Field Oracles

Field testing approaches rely on different kinds of test *oracle* to decide the test outcome. *Domain-dependent* oracles require information about the SUT application domain, for instance oracles that assert the results expected for some operations; *Domain-independent* oracles, such as oracles that assert the availability of the SUT, do not require the specific knowledge of the SUT's application domain. Fig. 5 classifies the approaches depending on the test oracles as *domain-dependent*, *domain-independent*, and hybrid, if they *support both* types of oracles.

Many studies (34 papers $\approx$ 43%) do not present specific oracles. The majority of studies that propose some oracle (76%) rely on domain-dependent oracles, 5 approaches rely on domain-independent oracles, 6 approaches rely on a combination of both types of oracles. The effort on oracles indicates that many field failures do not cause crashes and require some knowledge about the SUT to be detected. This is not a surprise, since in-vivo testing usually addresses stable applications, and is designed to reveal problems related to corner cases and infrequent scenarios.

Most approaches that consider some form of oracle do not explicitly discuss the techniques used to implement the oracles. Only 20 out of 46 approaches (43%) explicitly describe the proposed oracles. The majority of approaches that explicitly define an oracle use some form of specification-based oracles: component specifications [12], SUT models or specifications [6], BPEL specifications [20]. Several approaches refer to metamorphic relations [74, 21, 46, 10, 76]. Some approaches rely on user-defined oracles [63, 62, 25].

Some domain-independent oracles rely on QoS attributes of the SUT, such as general performance metrics, like response time [89, 70, 23], or execution duration and availability [83]. Few studies exploit the default oracle, that is, the detection of crashes or unchecked exceptions [44].

6.5 RQ1.2: Findings

We conclude this section summarizing the key characteristics of field testing approaches with respect to the source (where test cases are generated), strategies, triggers, resources and oracles.

- *Sources of Field Test Cases:*

Approaches for field test cases are evenly distributed between in-house and in-the-field generation: While several approaches investigate the opportunistic generation of test cases in the field to address unforeseen issues, test cases generated in-house are often sufficient to address field issues. Indeed, complex strategies that operate in the field are not always required. Simultaneously, improving the opportunistic and dynamic generation of field test cases is still an objective.

Ex-vivo testing strategies are still largely under-explored. Only few approaches take advantage of data from the field to generate effective field test cases, and execute them in the field, leaving a large space of opportunities for further study.

- *Strategies and Triggers:*

Most field testing approaches aim to predict failures, thus confirming the intuition that the main goal of *in-vivo* testing is to prevent failures to occur.

Field testing activities are triggered both by events in the SUT and in the environment, and by evolutions of the SUT or its components: Only few approaches periodically activate testing sessions, while most approaches

rely on asynchronous triggers related to the SUT, such as structural changes and reconfigurations, and the environment, such as new configurations and components.

- *Resources:*

In-vivo testing approaches heavily rely on data from the final production environment: Field data are important sources of information for field testing to identify new scenarios and corner-cases. Field data include user inputs, state information, logs, and environmental data.
In-vivo testing often demands additional engineering not required for the SUT: Gathering data from the field and coordinating field testing activities require non-trivial (hardware, virtual and software) infrastructures. Designing these infrastructures can be challenging, due to their impact on the complexity of the SUT, and the possible introduction of threats to safety and security that must be carefully addressed and compensated by the benefit of running field testing.
Many approaches do not explicitly mention the resources needed in the testing sessions: The additional resources required by field testing are not always explicitly discussed as part of the approaches, despite the required extra-cost and the extra-engineering effort.

- *Oracles*

Specification-based oracles are the most common field testing oracles: Field testing approaches often rely on specification-based oracles, with metamorphic relations frequently used both as oracles and as a support to generate new test cases.
The oracle problem is overlooked: In many cases the oracle either is not specified or the default one is used, resulting in approaches that might miss relevant failures due to lack of proper oracles.

7 RQ2: Test Target, Granularity and Type of Tested Applications

In this section, we discuss the targets of field testing (Section 7.1), the granularity of the software tested in the field (Section 7.2), and the type of applications considered for field testing (Section 7.3).

7.1 Target Features

Field testing can be designed to address different target features:

- *New features*: the feature is either new or tested without using information about past versions.
- *Regression issues*: the feature is tested after a change that is expected to have no effects on it.
- *Changed feature*: the feature is tested after a change that is known to affect its behavior.
- *Changed environment*: the feature is tested after detecting a change in the environment.

Table 4 shows the distribution of the different approaches with respect to the target features. It is worth noting that some approaches address more than one type of targets.

Table 4: Targets of testing approaches

Target	Number of approaches	%
New feature	24	35%
Regression	37	54%
Changed feature	28	41%
Changed environment	19	28%

Most approaches deal with regression testing (37 papers). In-vivo regression testing is used especially for software systems that can be dynamically reconfigured or adapted while running in the production environment, such as autonomic computing systems [54], and component-based systems [63]. Regression testing is also considered in ex-vivo approaches, as a way to obtain additional test cases that can reveal the side-effects of changes [72].

Field testing has been also significantly employed to test the impact of environment evolution. In fact, it is hard to exercise in-house every possible environment and configuration. Leveraging the natural diversity of environments available in the field is a clear strength of field testing.

When a new feature is released or an existing feature is changed, the validation activity cannot always be completed in-house, especially if the behavioral space of the SUT is large. Field testing has been exploited to continue validation activities in the field and discover the missed faults.

Although revealing regression problems has attracted more attention than other possible targets, all the four scenarios have been significantly investigated in the domain of field testing.

7.2 Granularity of the Tested Elements

Granularity refers to the granularity level of the tested elements: unit/component/service, integration/subsystem, system, or system of systems (SoS). Table 5 shows the distribution of the different approaches with respect to the addressed granularity level. Indeed, the majority of the studies consider unit, integration, and system testing, with most approaches focusing on unit level. In many cases, field testing approaches target multiple levels. Our analysis indicates the testing of System of Systems as a largely unexplored area. Given the growing complexity, size and degree of interoperability of modern software systems, such a level deserves greater attention in the future.

Table 5: Granularity of testing approaches

Granularity	Number of papers	%
Unit	45	57%
Subsystem	29	37%
System	25	32%
System of Systems	2	3%

7.3 Type of Tested Applications

The type of application directly influences field testing techniques, since it impacts on the core mechanisms, namely runtime test execution, isolation, monitoring. Current field testing approaches address four main classes of applications: *Desktop* applications, which run on desktop or laptop devices; *Mobile* applications, which run on mobile devices, such as smartphones, tablets, smartwatches; *Remote* applications, which run on servers, usually accessed via client applications, installed on desktop or mobile devices; and *Embedded* applications, which run on dedicated components with time and robustness constraints, and that are not as frequently updated as other types of applications.

Table 6 shows the distribution of the different approaches with respect to the type of addressed application. Most field testing techniques address remote applications, whose many resources facilitate the design of key features, such as isolation (often obtained by replicating components) and monitoring (ofter provided with little interference on the running systems). This is especially true for cloud infrastructures that provide virtually unlimited computing resources.

Table 6: Approaches per application type

Category	Number of papers	%
Remote	59	74%
Embedded	10	13%
Desktop	11	14%
Mobile	1	1%

Field testing approaches that target embedded applications consider interactions with hardware and environment, and focus primarily on ex-vivo testing [78, 28, 79] that can be safely performed in-house. Only few approaches deal with online testing for systems with strong fault tolerance and assurance requirements [95, 3, 40, 16, 51, 100].

Desktop applications receive little attention, probably due to the relatively low popularity nowadays. We found only one approach designed for mobile applications. The limitations imposed by mobile devices (limited computational resources) and mobile operating systems (security constraints) are probably quite challenging for field testing technology.

7.4 RQ2: Findings

We conclude this section summarizing the key characteristics of field testing approaches with respect to the target features, the granularity, and the type of the tested elements.

- *Target Features:*

Field testing addressed a variety of test targets, with a focus on the presence of unexpected side effects as consequence of changes, and on new and changed features, and environment changes.

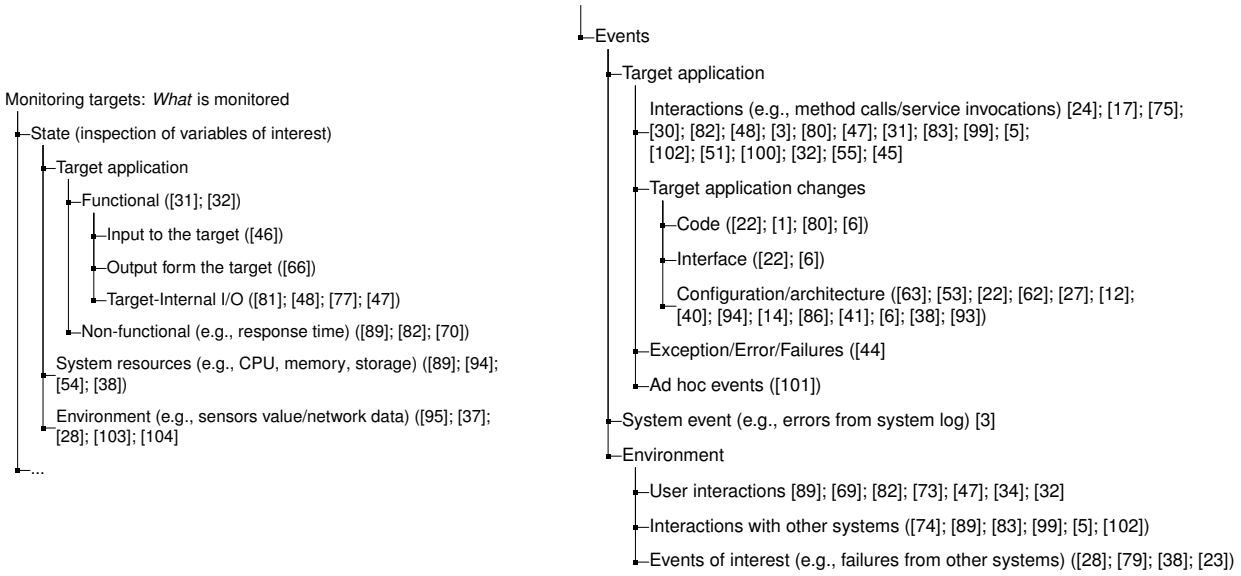


Figure 6: Field testing approaches by monitoring targets

• *Granularity of Tested Elements:*
Field testing is applied to all levels: units (functions, components, services), integration, and system. Indeed, field testing is a general solution that can address elements of different size and complexity.
Systems of Systems (SoSs) deserve more attention: SoSs are systems composed of multiple independent systems that cooperate opportunistically. Although so far they received little attention, they are complex systems that require field validation techniques to be properly addressed.

• *Type of Tested Applications:*
Most approaches apply to server applications that provide enough resources to easily address some of the key challenges of in-vivo testing.
Mobile applications are challenging: Mobile applications can be deployed on a huge diversity of devices, and can interact with the environment in a rich way thanks to the many sensors they can be connected to. It is thus an extremely interesting context for field testing. However, so far, field testing is still substantially unexplored. This is probably due to the constraints imposed by the mobile computing environment, such as the security and resource constraints, that make the deployment of field testing difficult. We expect more work in this domain in the future.

8 RQ3: Monitoring, Isolation, Privacy and Security

In this section, we discuss how field testing approaches monitor the SUT (Section 8.1), isolate tests in the field (Section 8.2), and address privacy and security in field testing (Section 8.3).

8.1 Monitoring

Monitoring is the process of dynamically gathering, interpreting, and elaborating data about the execution of the SUT. Monitoring is extremely important in field testing, as it captures data about events and states of both the SUT and the environment. Such data are needed to trigger the testing process, generate and select the test cases and identify the testing activities.

We distinguish direct and indirect monitoring, based on the source of information, that is, who produces the information. Monitoring is *direct* if the monitored information comes directly from the SUT, *indirect* if the monitored information comes from the software, physical or human environment where the SUT operates, for instance OS resources consumption, data read by sensors or data about user interactions with the system. We discuss *what* is monitored, that is, the information the monitoring facilities gather to support field testing, and *how* monitoring is implemented, that is, what techniques are used. The activities for *gathering* information include:

• *logging*: the process of recording textual and/or numerical information about *events of interest*,

- *tracing*: the process of recording information about the *control flow* of a SUT during its execution.

Logging and tracing differ in their goal, even when implemented with similar techniques, for instance by instrumentation: Tracing records the execution flow of the SUT execution without referring to specific classes of events of interest, while logging focuses on the events of interest, for instance, recording errors or failures. In the following, we discuss the monitoring solutions in field testing, by focusing on (1) what is monitored and (2) how software is monitored.

Fig. 6 groups field testing approaches according the targets of the monitoring activities, that is, the information of interest. Approaches that monitor different kinds of information are associated with more than one branch in the tree in the figure. We distinguish *(i)* approaches that monitor the *state* of the SUT itself, the system in which the SUT is executed, or the environment with which the SUT interacts, which entails reading the values of the variables of interest; from *(ii)* approaches that monitor *events of interest*, that is, events that cause the state to change, for instance, an action of a user that caused a reconfiguration or some method calls. Our survey indicates that 55 out of 80 (69%) studies explicitly specify *what* is monitored to support the testing process.

Many field testing approaches monitor the *state of the target*, namely the values of variables of interest. In several studies, the monitoring solution intercepts *functional* values of the variables of interest, namely the input/output exchanged to/from the target and within the target's modules. For instance, Hui et al.'s approach [46] detects integer overflows by online metamorphic testing. The approach monitors *inputs of insecure integer data* along with sensitive code paths to detect untrusted sources of integer values with their paths to security sensitive sinks. The approach uses the information to trigger testing, which exploits metamorphic relations to test the same path with untrusted values. Lee et al.'s approach [66] proposes an architecture for field testing of embedded systems that extends the Simplex architecture, to allow components to be upgraded and tested online. The approach compares the output of the component under test to the output of the component that is replaced in the Simplex configuration, in order for the system to benefit from the new component while still assuring a correct computation.

In many cases, field testing accesses the Internal I/O operations of the SUT modules. Hummer et al.'s approach [47] focuses on integration testing of dynamic service compositions, in which the data flowing within the BPEL composition is observed to generate test cases. Murphy et al.'s approach [77] monitors the values of variables in the scope of the function under test to understand if the application is traversing a previously unseen state and need to be tested.

Some approaches monitor *non-functional* attributes. Ma et al.'s approach monitors the system response time to perform adaptive performance testing of web services [70].

Other approaches monitor system-level information about *resource utilization* or about the *environment*. King et al.'s approach monitors the state of managed resources in autonomic computing systems to trigger self-testing routines [54]. De Olivera Neves et al.'s approach monitors the environment with sensors to trigger environment-dependent field test cases [28].

Some approaches monitor *events* at the level of the *target application*. In many cases, the events are about the *interaction within the application*, such as method calls or service invocations. In other cases, the events are about *changes that may occur in the target application* (e.g., module/service upgrades, interface changes and reconfigurations) and tests assess the impact of these changes.

Other approaches monitor *exceptions/error/failures* (e.g., unchecked Java exception [44]) or domain-specific *ad hoc* events (e.g., events specifically defined for coverage assessment and published in the event interface [101]). Approaches can also exploit *system events*, such as data present in the logs, to assess the expected versus observed behavior during testing [3]. Some approaches monitor interactions of the target application with the external *environment*, by capturing *interactions* with *users* or *other applications/services* such as in the case of service compositions.

Fig. 7 groups field testing approaches according to the technique adopted to monitor the SUT, that is, *how* monitoring is implemented. Approaches that monitor different kinds of information are associated with more than one branch of the tree in the figure.

Our survey indicates that 53 out of 80 (66%) of the studies explicitly indicate the monitoring technique, that is, *how* monitoring is performed (direct vs. indirect). Many approaches implement direct monitoring by *logging* runtime information either with or without instrumenting the SUT, in the latter case by exploiting information natively logged by the target application. Few approaches rely on ad-hoc components designed to capture a specific class of events. For example, Cooray et al. rely on the ODE-BPEL extension that provides the BPEL event listener API to monitor the Web service process execution [22].

Some approaches monitor the system by tracing the *control flow* of the execution, either with or without

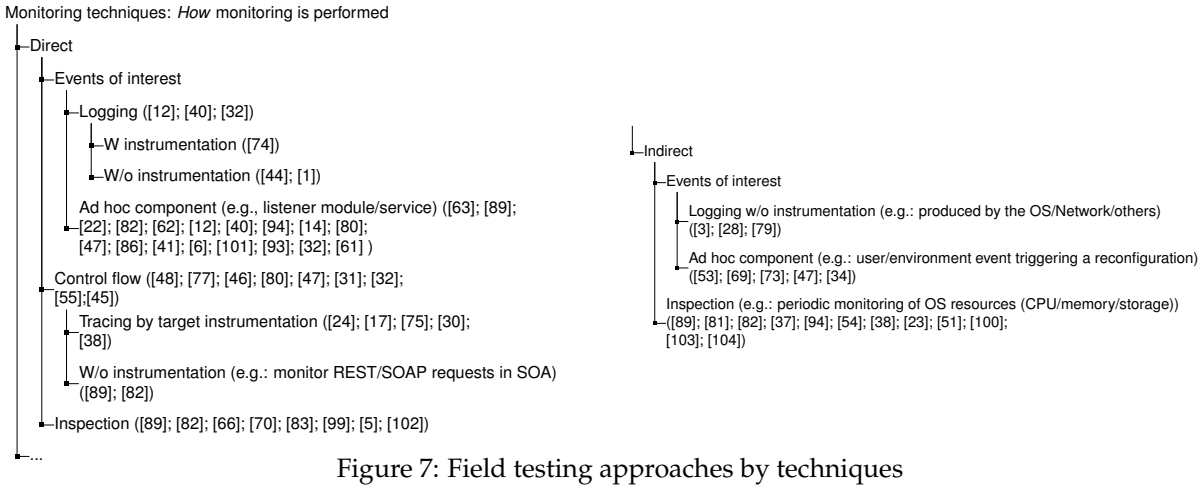


Figure 7: Field testing approaches by techniques

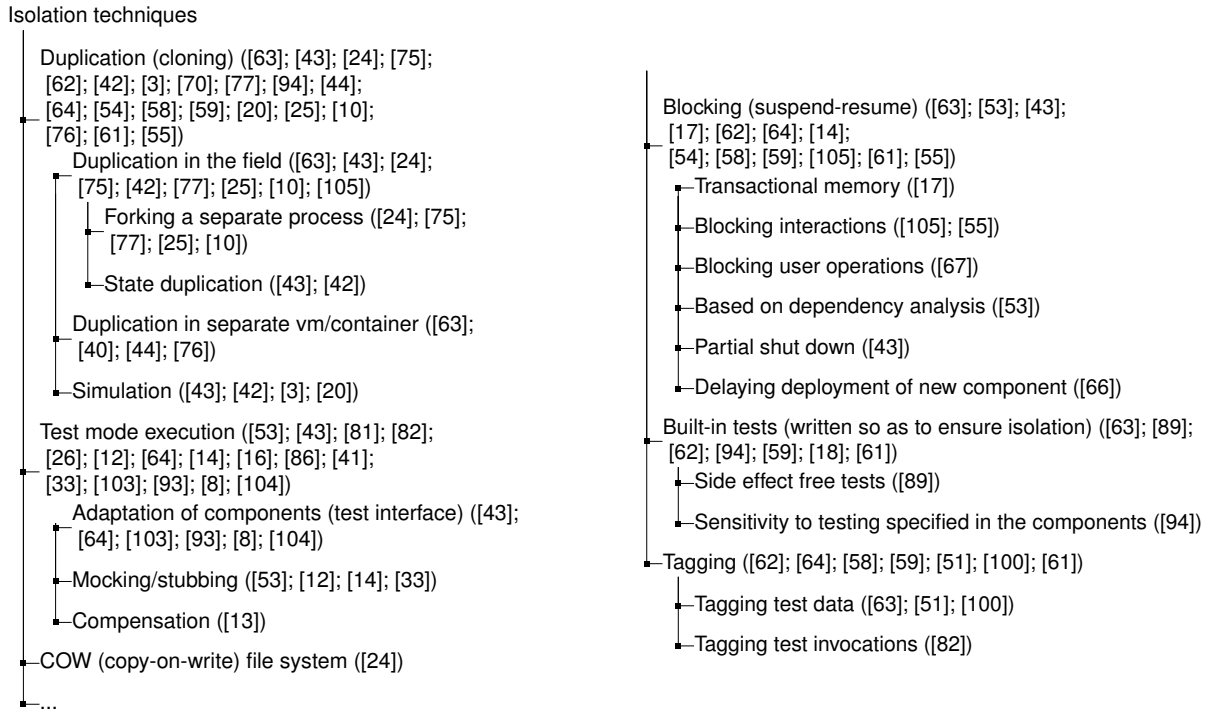


Figure 8: Isolation techniques

(static or dynamic) instrumentation. Other approaches periodically *inspect* SUT attributes.

Approaches that implement *indirect monitoring* are driven by information from the system and the environment. Some approaches access *logged information*, such as operating systems, network, and sensors logs [3, 28]. Other approaches propose *ad hoc components* for monitoring users' or environment's events that trigger reconfiguration [53], [69]. Yet other approaches rely on periodic inspection of attributes of the system and the environment, such as system resource consumption [54] and network data [51], [100].

8.2 Isolation

Field testing should not interfere with normal operations nor produce undesired side effects. To prevent side effects on the execution flow of the SUT, field testing approaches either execute field tests in sandboxed environments or do a roll-back before restarting normal execution. Field testing approaches are isolated at different levels. The *isolation level* of a field test is the computational unit that is subjected to isolation during field testing, and that can be safely field tested with warrantied no side-effects on the running system. Depending on the granularity of field testing, which we discussed in Section 7.2, the isolation level may scale from small computational units, such as classes or functions, to large units, up to the entire SUT.

Many approaches do not discuss the isolation problem, and do not explicitly propose solutions. Isolation is not a problem for either *ex-vivo* approaches or *in-vivo* approaches that simply compare the behavior of the

SUT observed in the field to the expected one without interfering [30].

Many approaches that target service-oriented applications either assume side-effect-free SUTs or suitable *compensation mechanisms* [13], that is, they assume that side effects can be either rolled back or compensated (e.g., by paying proper testing fees) when services are executed for testing purposes. In-vivo approaches that assume compensation mechanisms for allowing testers to (partially) ignore the side effects of in-vivo testing require careful engineering to avoid such compensation mechanisms to be abused or used beyond reasonable limits.

Some approaches address the isolation problem by assuming that test cases are side-effects free by design [89]. The difficulty of designing field test cases with no side effects depends on the test granularity: it may be not too difficult at small granularity levels, but it becomes very difficult at high granularity levels.

Fig. 8 summarizes the isolation mechanisms proposed for field testing. The most polular mechanism is *duplication*, also called *cloning*, that implements isolation by executing the field tests after duplicating the execution state, hence ensuring no interference with the execution in the production environment. Some approaches clone the execution state in the same execution space of the production environment (*duplication in the field*) by (i) *forking* a separate process devoted to in-vivo testing, (ii) *cloning the objects* involved in testing, or (iii) deploying redundant instances in the field for testing purposes (*state duplication*).

Other approaches separate in-vivo testing from the production processes, by either executing the field test cases in a *separated virtual machine* that duplicates the runtime environment or reproducing the main process in a *simulator*, possibly based on information gathered from the main execution by means of probes.

Another extensively used isolation mechanism is based on specific *test execution modes* that differentiate the execution in testing versus normal operational mode. The testing mode ensures that test cases do not affect the normal execution state. Test modes are implemented by (i) *adapting the behavior of the components*, for instance by using a test interface that in-vivo test cases use instead of the production interfaces, (ii) using *stubs and mocks*, or (iii) activating mechanisms that *compensate* the effect of test execution. All these variants assume that components are designed and developed for testability, with a test execution mode that prevents side effects.

A less investigated isolation mechanism is the usage of *copy-on-write* (COW) file systems [24], whose effectiveness is limited to side effects that leave a persistent trace in the file system.

Some approaches isolate field testing with *blocking* mechanisms that block the execution of components with potentially undesirable side effects for the whole duration of in-vivo testing. Field testing approaches implement blocking in various ways. A clean and elegant solution exploits *transactional memory*: In-vivo test cases are executed within a transaction that is rolled back when returning to normal execution. Variants of this solution include blocking either the *interactions* between components or *user operations* that may interfere with the execution process, identified with *dependency analysis*. Another way to implement blocking consists of *shutting down* all components that could interfere with the main process during in-vivo testing, to inhibit side effects.

Other approaches delegate isolation to the design of test cases (*built-in tests*). Writing *side-effect free* tests is not a straightforward solution, highly dependent on the skills of testers, who are in charge of defining proper tests for in-vivo execution. Other approaches delegate isolation to the design of the components, by requiring the components of the system to declare their sensitivity to testing, in terms of side effects that may depend on executing test cases in the field. It becomes then a responsibility of the test cases to check that the components have an adequately low sensitivity to testing.

Few approaches implement isolation by *tagging* the information generated during in-vivo testing, so that it can be distinguished and handled separately from the information originated by normal execution. It is possible to tag either the generated *data* or the performed *invocations*.

Few approaches define explicit *isolation policies*. Lahami et al. [63] propose multiple policies, and assume that *developers choose* among the alternatives (a subset of those in Fig. 8) once and for all, before deploying the test suite.

Gonzalez-Sanchez et al. [43] propose to declare policies in the components under test *based on the level of testability*, and configure specific isolation mechanisms based on the possible side effects declared in the components under test. Lahami et al. [58] define test cases *optimized for isolation*, by choosing the best isolation technique among the available ones, depending on component-system interactions.

Most isolation techniques work at *module* granularity level: *service*, *subsystem* and *class*, as shown in Fig. 9. Indeed, many of the techniques that exploit isolation strategies based on duplication, test mode, built-in tests, blocking and tagging operate at module level. Isolation at the *process* level is also quite common, while *application*, *system* and *host* isolation levels are less frequent and often more challenging than isolation

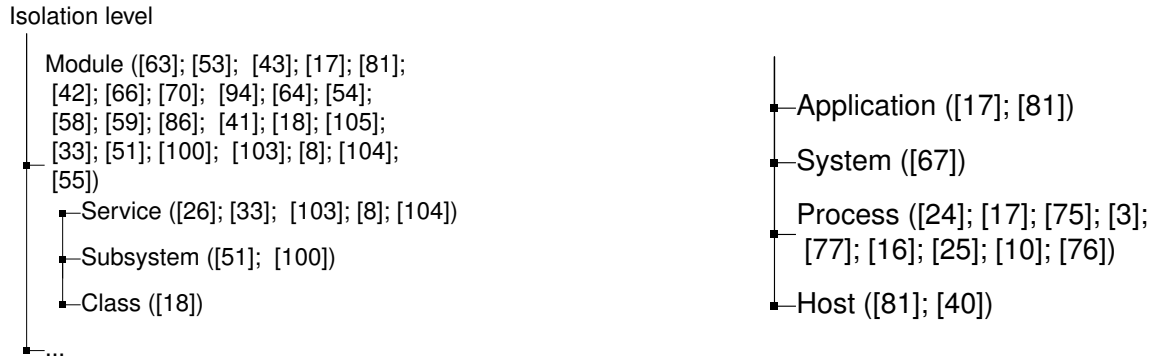


Figure 9: Granularity of the computational unit being isolated during field testing

at low granularity levels.

8.3 Privacy and Security

Privacy and security issues are still largely unaddressed. Only 8 studies out of 80 (10%) address such aspects. A common solution to privacy and security is *designing testable units* with specific features to ease testing and exposing internal information, for guaranteeing a given level of security and privacy. Ye et al. [101] propose a method for white-box testing of service compositions with minimal exposure of internal information about the participating services. They augment methods with event interfaces that expose encapsulated events and relaxed constraints, to check white box coverage, thus guaranteeing information hiding and privacy.

Zhu et al. [103] propose testing variants of existing services to enable third-party organizations to test the SUT while assuring non-disclosure of information, for instance, source code. The testing variants can be used in a collaborative setting in which test tasks are completed through the collaboration of various "test services", that are registered, discovered, and invoked at runtime using the STOWS ontology of software testing [104].

Bartolini et al. [8] propose testable versions of services to collect coverage reports, by means of services that do not disclose the internals of the SUT, while still providing coverage information.

Di Penta et al. [32] propose testable services to allow testers to send assertions checked by services. In this way, testers can obtain test results without directly accessing the monitored data.

Security is often addressed by creating *networks of trusted entities that communicate using secure channels*. Zhang [102] proposes an approach for dynamically testing web services for reliability before integrating a discovered service. The approach enables the node that hosts the web service to trust the testing mobile agent. Cooray et al. [22] rely on secure communication to assure the privacy of test execution logs and reports. Bertolino et al. [13] propose a method to test service compositions under the governance of a service federation. An online testing component triggers service testing requests that are indistinguishable from regular requests, thanks to an assertion signed by an identity provider, which grants a regular role to the test component. Tests are selected and executed proactively, so as to ensure the trustworthiness of the federation.

Finally, both Hummer *et al.* [48] and Murphy *et al.* [75] explicitly define privacy and security requirements, but do not propose solutions to satisfy them.

8.4 RQ3: Findings

We conclude this section summarizing the key findings about monitoring, isolation, privacy, security.

• *Monitoring:*
Monitoring is often overlooked: Many studies do not report details about monitoring, despite its importance: 31% of the studies do not explicitly indicate what they monitor, and 34% do not report on *how* they monitor the SUT.
Custom solutions for monitoring are prevalent: most monitoring solutions largely depend on the application context, testing objective and granularity.
Heterogeneous monitoring is quite common: Many studies privilege events and states of the SUT over the environment. However, a significant number of studies monitor several information sources (9 out of 53

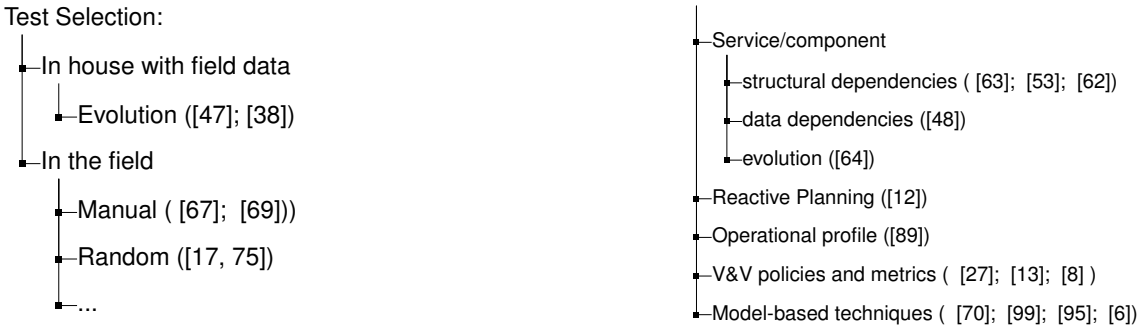


Figure 10: Field testing approaches by test selection criteria

studies that indicate *how* monitoring is done) and capture heterogeneous information (18 out of 58 studies that specify *what* is monitored) to retrieve the data required for field testing.

• *Isolation:*
The isolation problem is still largely open: Most approaches assume the availability of isolation mechanisms, leaving largely unexplored the many issues that derive from possible side effects on the state of the execution and on the environment, and from the interference with non functional properties, in particular performance.
"Design for isolation" is often advocated: Many approaches assume the execution environment is aware of and supportive to field test execution, and define test cases that take advantage of the isolation primitives available in the execution environment, such as test execution mode and compensation mechanisms. This is frequent in approaches designed for the web service domain.
Isolation policies largely ignore performance issues: Although some approaches proposes multiple alternative solutions for isolating field tests, there is no comparative evaluation of the performance footprint of the alternatives. As a consequence, the isolation policy is based on a-priori assumptions on the effects of each choice, rather than on objective measures of their impact. Experiments that measure the performance overhead of the proposed isolation mechanisms are quite rare.

• *Privacy and Security:*
Security and privacy are largely unaddressed: Very few studies address privacy and security issues in field testing, despite their importance, especially in some application domains, like web and service-based applications. Only one of the studies reported in this survey indicates security and privacy as a main focus [26].

9 RQ4: Field Test Selection, Prioritization, and Governance

This section discusses test selection and prioritization (Section 9.1), and test governance (Section 9.2).

9.1 Field Tests Selection and Prioritization

Test selection is the activity of choosing a suitable subset of test cases to execute. Our investigation reveals a broad spectrum of techniques spanning from simple strategies like manual and randomized procedures, to sophisticated strategies like reactive planning and model based approaches. Fig. 10 groups field testing approaches according to the proposed test selection strategies.

Few approaches select test cases in-house with data collected from the field. Frederiks et al. [38] and Hummer et al. [47] select ex-vivo test cases when observing changes of the system or its environment.

Lee et al. [67] and Loustarinen et al. [69] rely on *manual* test case selection based on domain knowledge, which may be accurate, but slow and error-prone. Many approaches propose automatic procedures. Bobba et al. [17] and Murphy et al. [75] rely on *random* selection, which is simple and fast, but gives no guarantees of accuracy and effectiveness of the selected test cases. Some approaches drive test selection using *information about the service or component* to be tested: King et al. [63] and Lahamani et al. [62] select test cases based on the structural dependencies to be tested, Hummer et al. according to data dependencies [48], and Lahami et al. according to information about the evolution of the unit under test [64].

Bertolino et al. investigate *reactive planning* solutions where the selection of test inputs is performed on-the-fly without requesting the generation of a test suite in advance [12]. Their auditing approach selects test cases at run-time guided by both the behavioral specification of the SUT (i.e., STS [36]) and the output

returned by the SUT at each interaction. Some approaches select test cases based on the *operational profile*. De Angelis et al. refer to modules that have been exercised less by users [89].

Other approaches select test cases according to policies and metrics relevant to the *V&V software process*. The works in [27, 13] consider test selection as part of a test governance framework among a federation of services (see Section 9.2). A governance framework includes the validation of the conformance of each member to the specifications/behaviors prescribed in the federated context. Bartolini et al. [8] propose a framework that supports the anonymous collection of coverage information. The latter can be used either to support regression testing activities on the SUT or to reduce a test suite by selecting only those tests that actually contribute to the coverage improvement.

Some approaches drive test selection with *models* either inferred from observations or provided by testers: to capture the behavior of the SUT and select the test cases that are likely to reveal failures, Vain et al. rely on Markov models [95], Wang et al. on Petri Nets [99]. Bai et al. [6] and Ma et al. [70] use reinforcement learning and agent-based paradigms to reason about the behavior of the SUT, and select the test cases to be executed in reaction to changes in the operational conditions or in the software system.

Only few approaches address test prioritization, that is, a strategy for scheduling the order of execution of field test cases. Mei et al. [72] propose test prioritization without a selection strategy. They prioritize the execution of in-house tests based on changes that occur in the field to service compositions, assigning higher priority to the tests that are more likely to exercise the change. Vein et al. [95] use a probabilistic strategy. Few other approaches barely refer to some prioritization strategies without providing details [89, 73, 32].

9.2 Field Testing Governance

In general, governance is the act of administering, and relates to a set of policies, measures, practices and responsibilities to control and direct a complex system of relations and interactions. In the context of software development, governance refers to a framework that defines and coordinates the tasks, activities and roles of the software process. In the context of field testing, a governance framework provides a setting to execute and control testing activities, and establishes policies that guide the decision of when, how much, and how to test: *Test Governance* concerns the establishment and enforcement of policies, procedures, notations and tools that are required to enable test planning, execution and analysis of a given SUT [15].

We identify two dimensions that are relevant for field testing governance: Orchestration and User awareness. *Orchestration* concerns the strategy adopted or assumed to ensure that a proposed method or technology can be suitably embedded within the target application domain and successfully managed by all involved stakeholders. *User awareness* refers to the required or assumed degree of awareness of the end users of the SUT about the field testing activities: in some cases the users might be informed and could be even asked to cooperate to the testing campaign, in other cases the test could be conducted leaving the users blind about the fact that some executions are launched for testing purposes.

Fig. 11 summarizes the orchestration strategies proposed for field testing. Only a small set of approaches explicitly deal with an orchestration strategy. Some approaches acknowledge the need to make assumptions behind the tested application, but do not propose an orchestration strategy. King et al. [53] mention that a *Test Manager* is in charge for test planning and management, and for evaluating test results 'against the predefined test policies', which are available from a *knowledge repository*. Ma et al. [70] apply the BDI (*Belief-Desire-Intention*) model for adaptive online performance testing, and assume a series of rules for deciding which services to test and how to allocate the testing tasks. In similar way, Cooray et al. [22] rely on the users of the testing system for supplying a test policy for each target service, whereby a test policy specifies the test configuration and schedule. However, not many details are given about such policies.

De Angelis et al. [27] and Bertolino et al. [14] propose different types of policies to support the orchestration strategy. They propose policies to decide when, how and what to test during field testing, without assuming a specific test framework. Skoll also allows the definition of strategies that control the testing process [87]. The TT4RT framework for runtime testing by Lahami at al. [58, 59] includes a description of policies for limiting side effects.

Several approaches propose an explicit orchestration strategy. Ali et al. [2] illustrate the strategy behind their proposed online testing framework for service choreographies. They describe both the involved stakeholders, including a choreography board, the testing engineers, the service providers and the choreography end users, and a schematic process for field testing. The PLASTIC framework [12] includes an online testing session for service admission, which requires an interaction protocol involving the requesting service, the registry, a Test driver, and a Proxy/Stub service factory. The framework proposed by Zhu and Zhang [104, 105] (originally outlined in a preliminary work [103]) is orchestrated around the provision of

Orchestration:
- Need for strategy is only acknowledged ([53];[70];[22])
- Governance rules and policies are specified ([27];[14];[58];[59])
- An explicit strategy is developed ([2];[12]; [104];[105];[103];[37];[87])
- Ad-hoc strategies for evolving autonomic systems are conceived ([54]; [93];[55])
- Strategies for domain specific systems are devised ([8];[101]; [69];[95])

Figure 11: Field testing approaches by orchestration strategy

dedicated test services, both general-purpose and specific ones, and the STOWS ontology for web service testing that saves the information needed for the registration, discovery and invocation of such test services. Proteus [37] provides runtime testing for self-adaptive systems. The framework can adapt test suites and test cases so that both remain relevant despite changing operating conditions. The Proteus orchestration strategy is based on two basic rules: an adaptive test plan is provided at design time for each configuration and a testing cycle is executed at each new configuration.

King et al. [54, 93] and Stevens et al. [55] propose ad-hoc strategies for on-line self-testing of autonomic systems, for instance, at system evolution.

Few studies develop domain-specific strategies. In service-oriented architecture, the SOCT approach for *Service-oriented Coverage Testing* [8] is possible thanks to an orchestration strategy requiring that (i) the service provider releases an instrumented service, a.k.a *testable service*; (ii) a third-party service called TCOV provider collects coverage information, while (iii) a service consumer performs field testing of the service. Ye and Jacobson's approach [101] uses a similar orchestration strategy, assuming that instead of coverage information, the testable services espose events that can be monitored through a dedicated interface by the third-party service. Luostarinen et al. [69] use field testing for remote usability testing, by requiring to natively integrate a dedicated API within the user interface. Vain et al. [95] propose testing in operation for mission-critical systems, with an orchestration strategy based on a model-based conformance testing approach.

Concerning user awareness, only few approaches explicitly address if and how the users are explicitly aware and involved into field testing. Niebuhr et al.'s approach [80] specifies the test cases executed at runtime by the Service Users. Luostarinen et al. [69] foresee that users actively participate to field testing by driving the requested usability tests. Skoll requires nodes to explicitly join the infrastructure in order to participate to the testing process [87].

In some cases even though the authors do not explicitly discuss user awareness, we can infer that users might indirectly become aware of the field testing activity because during testing the system might be blocked, or delayed. This may happen for the fault-injection technique by Alnawasreh et al. [3], or in the work by Lahami et al. [62], depending on which policy, among the four implemented ones, is selected to reduce the impact of field testing. In the work by Murphy et al. [76] the user would be warned in case the field test reveals unexpected behavior.

Some approaches explicitly make an effort to leave the user completely unaffected by execution of field tests using isolation solutions such as sand-boxing, as we discuss in Section 8.2.

9.3 RQ4: Findings

We conclude this section summarizing the key characteristics of field testing approaches with respect to test selection, test prioritization, and test governance.

- *Test selection*

Test selection has been mostly investigated for in-vivo testing: The vast majority of approaches for selecting test cases focuses on in-vivo testing, where reducing the number of test cases to execute is extremely important, since the production environments typically offer limited resources for the execution of the test cases. Although less investigated, test selection techniques for ex-vivo testing can still be useful, when the ex-vivo test suites are particularly large.

- *Test case prioritization*

Test case prioritization is poorly investigated in field testing: This is quite surprising, since running tests in the field can be both expensive and risky. Thus, optimizing test execution to anticipate the discovery of failures could be particularly relevant for large field test suites. On the other hand, most of the studies consider test

suites of limited size for which prioritization is not likely to have a large impact. This might explain the outcome of our survey. We expect work on test prioritization to increase in the future.

- *Test governance*

The importance of governance is undervalued: Field testing poses many challenges, and a proper governance framework becomes necessary for making field testing possible, specifically including a proper orchestration strategy and minimizing the impact on the end users of the production system. However, only 20 out of the 80 studies discuss orchestration, and among those only 6 develop an explicit strategy. Only 6 approaches explicitly discuss users' involvement.
Orchestration rules and policies are needed: In field testing, testers need to refer to appropriate rules and policies that establish when, how and by whom a selected set of tests can be executed. Some of the studies assume that such rules and policies exist, and the studies focus on the technical challenges. Only few studies propose some rules and policies, even considering domain-specific contexts. However, most studies completely overlook the orchestration dimension.
Impact on users in production: The execution of field tests may directly or indirectly impact users' experience with the system under test. Only very few studies mention this issue, and take different research directions: either the users are made aware of field testing and is expected to actively participate, or an explicit effort is made not to affect users in any way.

10 Discussion

Table 7 summarizes the main findings for each research question. RQ1 deals with the "how is software tested" dimension of field testing. Our survey shows that most approaches are based on *functional* testing, that is, test cases for field testing are derived from the functional requirements and in particular from formal or semi-formal specifications (e.g., finite state models). However, approaches for fully automated generation of field tests are still missing. Non functional aspects, such as quality of service attributes, have been investigated widely only within the service-oriented domain.

The main trigger for in-vivo testing is the in-field observation of a new or anomalous change in the execution environment or in the application under test itself (e.g., in case of a software update or re-configuration). Field test cases make use of data and computational resources available in the field, although such usage is not always analyzed in detail, despite its potentially negative impact on the end user experience. Since the in-vivo execution state is possibly unknown, developers of field test cases often rely on weak forms of oracles, such as metamorphic relations, that do not require a detailed analysis of all possible execution conditions and configurations.

RQ2 deals with the "what is tested" dimension of field testing. Field testing has been investigated at all granularity levels. There are examples of field testing approaches targeting individual units, such as functions or services, the integration of units, as well as the entire system. However, the types of systems being field tested are not evenly spread among all domains. Server applications that provide long running services to their users or to other applications have been extensively considered as case studies, although the scalability of field testing to larger federations of cooperating systems is still an unexplored research area. While combinatorial testing addresses the problem of the huge configuration space of software product lines, such testing approaches remain mostly confined to in-house testing and how to test those systems in the field is still a quite unexplored research area.

RQ3 deals with the "what is required" dimension of field testing. The most critical aspect of field testing is probably the creation of mechanisms to ensure isolation of field test executions. While some form of a-priori test case design for in-vivo execution is unavoidable, there are major opportunities to develop a general purpose infrastructure that provides isolation services to in-vivo tests. The solutions available as research prototypes span from duplication to test mode execution, blocking and tagging, but no comprehensive and comparative assessment was carried out to establish their performance footprint, as well as their impact on privacy and security.

RQ4 deals with the "how is field testing managed" dimension. The main finding for this research question is that field testing policies are largely missing and under-studied. While proper governance of the field testing activities would be needed to ensure an even distribution of the testing load across application instances and to control that field testing is activated only when it provides a global added value with respect to the previously executed test cases, no such a comprehensive framework is available in the state of the art. Existing works focus only on some of the aspects of the field test management dimension, such as the selection of which test cases to execute or their prioritization.

Table 7: Summary of the findings

RQ1: Approaches, Fault Types, Test Case Generation and Platforms	
Field testing mostly focuses on validating functional requirements, while non-functional aspects are under-investigated. The research on validating non-functional requirements in the field mostly focuses on QoS attributes. Specification-based test case generation is quite popular, but automatic synthesis of valuable test cases that can be safely executed in the field is still an open issue.	Section 5
RQ1: Source, Strategy, Triggers, Resources and Oracles	
While in-house and field test generation are widely studied, ex-vivo testing is clearly under-explored.	Section 6.1
Field testing is mostly exploited to anticipate failures and is commonly activated as a reaction to events from the SUT or on its environment.	Section 6.2
Data gathered from the production environment are the most relevant resource for in-vivo testing, however gathering such data requires additional engineering or infrastructures, not well addressed yet.	Section 6.3
Specification-based oracles are often exploited to reveal domain-dependent failures, but the oracle problem is sill largely overlooked. Suitable oracles are either not available or not usable.	Section 6.4
RQ2: Test Target, Granularity and Type of Tested Applications	
Field testing addresses a variety of target objectives: overall it is an effective means to augment the validation activities that cannot be realistically completed in-house.	Section 7.1
Field testing applies to all levels from unit to system testing. However only few studies focus on scenarios foreseeing an opportunistic cooperation among complex systems. Research on field testing for Systems of Systems deserves more attention.	Section 7.2
Long-running server applications are extensively investigated: some of the key challenges of in-vivo testing can be addressed by acting on additional dedicated resources. Despite the evolving configurations sensed by environment and the huge diversity of the devices are ideal scenarios advocating for in-vivo testing, mobile applications have not been frequently considered, probably due to the complexity of the mobile environment.	Section 7.3
RQ3: Monitoring, Isolation, Privacy and Security	
Monitoring is often overlooked: most studies do not discuss in detail which data are monitored and how, and rely on custom solutions tailored to the context. However, some studies investigate combined approaches that exploit several information sources.	Section 8.1
Isolating the SUT from the side effects due to field testing is still an open problem. The "Design for isolation" principle is often advocated, but isolation is usually not considered as part of the contributions. Automated isolation techniques mostly contribute to specific aspects. In general, performance studies estimating the overhead due to the adopted isolation mechanisms are rare.	Section 8.2
Security and privacy are largely unaddressed. Further investigation on approaches that specifically guarantee privacy and security in field testing is needed.	Section 8.3
RQ4: Field Test Selection, Prioritization, and Governance	
Test selection and prioritization are mostly investigated for in-vivo testing. Although less investigated, test selection techniques for ex-vivo testing are still useful, when the size of ex-vivo test suites increases.	Section 9.1
The importance of a comprehensive governance framework is undervalued. Orchestration rules and policies are needed to establish when, how and by whom a selected set of tests can be launched. The explicit management of the impact on users in production has to be better considered and investigated.	Section 9.2

While we have designed our research questions to be as independent and "orthogonal" as possible, there are interactions among them. The absence of a well established framework for field testing governance affects the assessment of the isolation strategies. In fact, the performance degradation due to isolated in-vivo test execution depends on the overall governance of the field testing process, which eventually determines the frequency of field testing experienced by each running application instance. In turn, the effectiveness of test case generation, selection and prioritization depends on the effectiveness of the triggers that activate field testing. In fact, additional coverage can be achieved and additional faults can be exposed only if the right application state triggers the execution of the right test cases. Finally, the chosen oracle is a cross-cutting factor that determines the effectiveness of all other in-field activities, since a weak oracle may not expose any fault despite the timely activation of the appropriate in-vivo tests.

10.1 Open Challenges for Researchers

Our survey identifies several open research challenges:

• **Generating and implementing field test cases**: Generating and implementing test cases designed for the field is still an open challenge. Field test cases shall adapt to the production environment, that is, they shall exercise the software system in the context of the production environment. Field test cases must offer a huge degree of openness and must deal with a high degree of uncertainty, since the field is not entirely known during development. Test cases with these characteristics have been mostly studied in the domain of service-based applications to test the interaction with dynamically discovered services. Consolidating the field testing practices to effectively address a wider range of situations is an important research direction. While existing approaches for test generation (especially specification-based ones) have been employed also for field testing, more advanced scenarios, such as the opportunistic generation of test cases in the field based on the characteristics of the underlying production environment, could also be conceived. These strategies are still underdeveloped and significant effort is still needed to move test case generation

approaches from the development to the production environment.

• **Isolation Strategies**: Field test cases must be non-intrusive, that is, they should not interfere with the processes running in production and their data. There are many strategies to guarantee the isolation of the test cases, such as duplicating processes and components, enabling test modes, and selectively blocking executions. However these strategies might be difficult and expensive to apply, depending on the domain and the test cases that must be executed. We need solutions that can be conveniently applied and adapted to specific contexts, including approaches to design software ready to be field tested, and software components with built-in tests.

• **Oracle Definition**: Oracles for field testing need to adapt to the unknown execution conditions that can emerge in the field. Oracles checking abstract properties might miss relevant test failures, while oracles accurately checking the result produced by a field test might be extremely hard or even impossible to write. When specifications are available, they can be exploited to generate effective field oracles. However, defining oracles that can be effectively used in the field jointly with field test cases is largely an open research challenge.

• **Security and Privacy**: Executing test cases in the field challenges security and privacy. Indeed, the testing infrastructure can be potentially exploited to attack the software system, and the data mined from the field by field tests, for instance failure reports, may accidentally violate users' privacy. Security and privacy aspects have been under-investigated so far, and must be urgently addressed to move field testing to production.

• **Orchestrating and Governing Test Cases**: Field testing requires a disciplined approach, due to the potential impact on production: hence, testers need to refer to appropriate rules and policies that establish when, how, by whom, and in which order, a selected set of tests can be executed. Some of the studies just assume that such rules and policies are in place, thus bypassing the problem. The few studies that propose some rules and policies often consider domain-specific contexts. Research is needed to find appropriate governance strategies and establish the technical and organizational conditions under which such strategies can be actualized. On the other hand, the execution of field tests may directly or indirectly impact users' experience with the system in production, and it remains an open challenge how to mitigate or recover such impact after field test execution.

• **Challenging Domains**: Although field testing has been experimented in multiple domains, so far, field testing has been studied limitedly or not studied at all in other domains. In particular, the safety critical domain is extremely challenging for field testing, due to the consequences that an imperfectly isolated testing environment may have. Surprisingly, the mobile computing domain also received little attention, despite the huge variety of devices and environments mobile applications can interact with. This is likely due to the security constraints and limited resources present in mobile devices and their operating systems. Although challenging, these domains can greatly benefit from the field testing technology, and we expect more research in the future.

10.2 Guidelines for Practitioners

While answering the research questions along which we structured this survey, we made some observations that are of particular interest to practitioners in the industry.

• **Specifications**: Specifications are intensively used both for test generation as well as for identification of suitable oracles. However, it is also the case that specifications are not typically found in real systems, and when available are not specified formally, hence they are not suitable for automated processing. Practitioners in the industry, in particular when complex systems of systems are involved, could adopt development methodologies that make use of (formal) specifications, such as model driven development. This enables the effective application of automated in-vivo testing techniques and tools, in addition to addressing the general problem of test oracle definition.

• **Isolation**: One of the important findings of the survey is that isolation is critical for effective in-vivo testing. However, in many cases, achieving isolation during in-vivo test execution is far from trivial and remains still an open research problem. Practitioners could address it by adopting design and development practices that are intrinsically compatible with isolation, making sure that side-effects are kept at a minimum. When a system is planned to be tested in-vivo, developers should introduce isolation by design into the system.

• **Security and privacy**: Findings of the survey show that security and privacy concerns are mostly left unaddressed during in-vivo testing. While this could be partly attributed to research efforts trying to reduce the complexity of the problem by scoping out security/privacy concerns, it is also due to the fact that explicit policies outlining security and privacy related aspects of the systems under test are lacking. Practitioners adopting in-vivo testing should specify and document explicitly the security and privacy guidelines accompanying the system. This would support a compatibility check between the information access needed for in-vivo testing and the information restrictions enforced by such security and privacy guidelines.

• **Governance**: The survey findings show that the issue of comprehensive in-vivo testing governance is not addressed by the overwhelming majority of the works surveyed. While the research should do better to incorporate governance, this is also partly attributable to the lack of clearly defined governance frameworks from the systems under test. Practitioners should adopt a comprehensive governance framework that clearly defines the in-vivo process in the life-cycle of development and testing of the system. In this way, the in-vivo testing phase would play a specific role in the overall governance, which in turn determines the frequency of in-vivo test execution, its goals and the overall test data collection process. This is a core governance concern in highly distributed settings, with a multitude of diverse, geographically scattered users.

11 Threats to validity

The process of selecting and reviewing the articles is subject to the risk of bias. We hereafter report the potential validity threats and the actions we took to mitigate them.

Study identification/sampling: Selection of the primary studies can end up with a non-representative sample with respect to the investigated topic, for instance some relevant papers might have been included or excluded incorrectly. The first step of our search was by a keyword-based strategy on the *Sciverse Scopus DB*. Although it is a single source, *Scopus* is one of the largest general purpose DBs for peer-reviewed literature and indexes journals/proceedings of the most common publishers including *Elsevier, Springer, Wiley, IEEE, ACM*. As general purpose DB, we preferred *Scopus* to *Google Scholar*, as the latter includes non-published literature (e.g., pre-print), papers that are not scientific articles or not peer-reviewed (e.g., technical reports, theses and other grey literature). The search string was kept generic so as to cover as much relevant studies as possible; although this conservative approach has lead to a wide set, requiring an intensive manual filtering, it has mitigated the risk of missing relevant studies. Moreover, to complement the search, we ran a forward and backward snowballing step.

All the steps, from the initial filtering (from 1,238 to 434 studies) to the application of inclusion/exclusion criteria (from 434 to 48 studies) and to snowballing (from 48 to 80) were conducted, independently, by the four research units which the authors belong to: each unit analysed a subset of the papers selected randomly from the whole set at each stage, discussing both within the unit and between the units with (plenary and one-to-one) online meetings, so as to agree on the papers to include/exclude and, at the same time, iteratively refine inclusion/exclusion criteria.

Another threat regards the **quality** of selected studies. To mitigate it, studies are searched among those indexed by Scopus, which considers only peer-reviewed studies (a well-established requirement for high quality publications), excludes grey literature, and filters out several low-quality conferences and journals. Inclusion/exclusion criteria are applied on each of the 434 studies to ensure to keep only studies pertinent to field testing. This indeed mitigates but does not eliminate the threat of low-quality studies.

Data extraction and classification, according to the defined dimensions, could also be biased by a subjective interpretation. To reduce such a risk, the extraction and classification were done according to a data collection form with 9 dimensions and 20 sub-dimensions defined to answer the four research questions, in order to record data and categorize the studies unambiguously. To validate the form, we used a *test-set* strategy [84, 56], in which subsets of 14 papers were assigned to each unit, with each paper reviewed by at least two units. The units classified the papers independently, and then discussed in plenary meetings so as to ensure that all the reviewers had the same understanding of the dimensions and of their alignment with the four research questions.

After the dimensions validation, the rest of the papers were assigned to the four units and classified independently. The results were discussed in several inter-units and in six plenary meetings. The process, involving all the authors in multiple iterations, also mitigates the interpretative validity threat, due to the researcher bias in interpreting the data, classifying the papers and map them to the final findings [84]. The

experience and expertise of many of the authors in the field of software testing is an additional mitigation factor. Though, since this step involves human judgment, the threat cannot be eliminated.

In general, the best practices for literature reviews by Kitchenham et al. for studies selection and analysis have been followed as documented above [56], and we published the final categorization on the ACM Digital Library as supplemental online material of this manuscript, thus making our analysis easy to be replicated by other researchers.

The final findings derived from the collected evidences are with references to a wide variety of applications and domains. However, some findings could change if applied to applications/domains outside what observed. For instance, the evidence collected for test prioritization in field testing (Section 9.3) is mainly based on applications with small/medium test suites, hence our findings could change if larger test suite are considered.

12 Conclusions

Field testing techniques address the complexity, unpredictability, evolvability and size of modern software systems – challenges that in-house testing activities cannot manage satisfactorily due to the huge configuration space under which those systems can operate. This paper surveyed the state of the art in field testing following four research questions that deal with the multiple dimensions of the field testing activities: (1) how field tests are generated; (2) what field tests are generated; (3) what is required to execute the field tests; and (4) how field testing is managed.

The area of field testing is a challenging one and the existing research has just scraped the surface. Among the directions for future research, the most promising ones include the automated synthesis of test cases with high added-value (e.g., coverage increase) associated with in-field execution; the creation of oracles that target those faults that escape in-house testing and are better exposed in the field; the execution of in-field test cases on end-user devices with limited computation, memory and energy resources, such as mobile phones; approaches to ensure privacy of the exchanged data and security of the application under test when in-field tests are executed; overall governance of the distributed execution of field tests across all available installations and users.

ACKNOWLEDGMENTS

This paper describes research work undertaken in the context of the Italian MIUR PRIN 2015 Project: GAUSS. This work was partially supported by the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703).

References

[1] Mohammed Akour, Akanksha Jaidev, and Tariq M. King. Towards change propagating test models in autonomic and adaptive systems. In *Proceedings of the 18th International Conference on the Engineering of Computer-Based Systems (ECBS 2011)*, pages 89–96, Las Vegas, USA, April 2011. IEEE.

[2] Midhat Ali, Francesco De Angelis, Daniele Fanì, Antonia Bertolino, Guglielmo De Angelis, and Andrea Polini. An Extensible Framework for Online Testing of Choreographed Services. *Computer*, 47(2):23–29, February 2014.

[3] Khaled Alnawasreh, Patrizio Pelliccione, Zhenxiao Hao, Mårten Range, and Antonia Bertolino. Online Robustness Testing of Distributed Embedded Systems: An Industrial Approach. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering: Software Engineering In Practice Track (ICSE-SEIP 2017)*, pages 133–142, Buenos Aires, Argentina, May 2017. IEEE.

[4] Sultan Alyahya. Crowdsourced software testing: A systematic literature review. *Information and Software Technology*, 127, November 2020.

[5] Xiaoying Bai, Shufang Lee, Wei-Tek Tsai, and Yinong Chen. Ontology-based test modeling and partition testing of web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2008)*, pages 465–472, Beijing, China, September 2008. IEEE.

[6] Xiaoying Bai, Dezheng Xu, Guilan Dai, Wei-Tek Tsai, and Yinong Chen. Dynamic reconfigurable testing of service-oriented architecture. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 368–378, Beijing, China, July 2007. IEEE.

[7] Luciano Baresi and Carlo Ghezzi. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010)*, pages 17–22, Santa Fe, USA, November 2010. ACM.

[8] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Bringing white-box testing to service oriented architectures through a service oriented approach. *Journal of Systems and Software*, 84(4):655–668, April 2011.

[9] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[10] Jonathan Bell, Christian Murphy, and Gail Kaiser. Metamorphic runtime checking of applications without test oracles. *CrossTalk, The Journal of Defense Software Engineering*, 28(2):9–13, 2015.

[11] Antonia Bertolino, Guglielmo De Angelis, Micael Gallego, Boni García, Francisco Gortázar, Francesca Lonetti, and Eda Marchetti. A systematic review on cloud testing. *ACM Computing Surveys*, 52(5), September 2019.

[12] Antonia Bertolino, Guglielmo De Angelis, Lars Frantzen, and Andrea Polini. The PLASTIC framework and tools for testing service-oriented applications. In Andrea De Lucia and Filomena Ferrucci, editors, *Software Engineering, International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, volume 5413 of *Lecture Notes in Computer Science*, pages 106–139. Springer, 2007.

[13] Antonia Bertolino, Guglielmo De Angelis, Sampo Kellomaki, and Andrea Polini. Enhancing service federation trustworthiness through online testing. *Computer*, 45(1):66–72, 2012.

[14] Antonia Bertolino, Guglielmo De Angelis, and Andrea Polini. Governance policies for verification and validation of service choreographies. In J. Cordeiro and K.H. Krempels, editors, *Proceedings of the 8th International Conference on Web Information Systems and Technologies (WEBIST 2012), Revised Selected Papers*, volume 140 of *Lecture Notes in Business Information Processing*, pages 86–102, Porto, Portugal, April 2012. Springer.

[15] Antonia Bertolino and Andrea Polini. Soa test governance: Enabling service integration testing across organization and technology borders. In *Proceedings of the International Workshop on Web Testing (WebTest)*, pages 277–286, Denver, USA, April 2009. IEEE.

[16] Smt. J. Sasi Bhanu, A. Vinaya Babu, and P. Trimurthy. A comprehensive architecture for dynamic evolution of online testing of an embedded system. *Journal of Theoretical and Applied Information Technology*, 80(1):160–172, October 2015.

[17] Jayaram Bobba, Weiwei Xiong, Luke Yen, Mark D. Hill, and David A. Wood. Stealthtest: Low overhead online software testing using transactional memory. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT 2009)*, pages 146–155, Raleigh, USA, September 2009. IEEE.

[18] Daniel Brenner, Colin Atkinson, Barbara Paech, Rainer Malaka, Matthias Merdes, and Dima Suliman. Reducing Verification Effort in Component-Based Software Engineering through Built-In Testing. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006)*, pages 175–184, Hong Kong, China, October 2006. IEEE.

[19] Kai-Yuan Cai, Yong-Chao Li, and Ke Liu. Optimal and adaptive testing for software reliability assessment. *Information and Software Technology*, 46(15):989–1000, December 2004.

[20] Tien-Dung Cao, Patrick Félix, Richard Castanet, and Ismail Berrada. Online testing framework for web services. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 363–372, Paris, France, April 2010. IEEE.

[21] W. K. Chan, S. C. Cheung, and Karl R. P. H. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):61–81, 2007.

[22] Mark B. Cooray, James H. Hamlyn-Harris, and Robert G. Merkel. Dynamic test reconfiguration for composite web services. *IEEE Transactions on Services Computing*, 8(4):576–585, 2015.

[23] Andrew Diniz da Costa, Camila Nunes, Viviane Torres da Silva, Baldoino Fonseca, and Carlos J. P. de Lucena. Jaaf+t: A framework to implement self-adaptive agents that apply self-test. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010)*, pages 928–935, Sierre Switzerland, March 2010. ACM.

[24] Huning Dai, Christian Murphy, and Gail Kaiser. Configuration fuzzing for software vulnerability detection. In *Proceedings of the Fifth International Conference on Availability, Reliability and Security (ARES 2010)*, pages 525–530, Krakow, Poland, February 2010. IEEE.

[25] Huning Dai, Christian Murphy, and Gail E. Kaiser. CONFU: configuration fuzzing testing framework for software vulnerability detection. *International Journal of Secure Software Engineering*, 1(3):41–55, 2010.

[26] Guglielmo De Angelis, Antonia Bertolino, and Andrea Polini. (role)cast: A framework for on-line service testing. In *Proceedings of the 7th International Conference on Web Information Systems and Technologies (WEBIST 2011)*, pages 13–18, Noordwijkerhout, The Netherlands, May 2011. SciTePress.

[27] Guglielmo De Angelis, Antonia Bertolino, and Andrea Polini. Validation and verification policies for governance of service choreographies. In *Proceedings of the 8th International Conference on Web Information Systems and Technologies (WEBIST 2012)*, pages 58–70, Porto, Portugal, April 2012. SciTePress.

[28] Vânia de Oliveira Neves, Márcio Eduardo Delamaro, and Paulo Cesar Masiero. An environment to support structural testing of autonomous vehicles. In *Proceedings of the IV Brazilian Symposium on Computing Systems Engineering (SBESC 2014)*, pages 19–24, Manaus, Brazil, November 2014. IEEE.

[29] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. Search-based crash reproduction using behavioural model seeding. *Softw. Test. Verification Reliab.*, 30(3), May 2020.

[30] Peter H. Deussen, George Din, and Ina Schieferdecker. An on-line test platform for component-based systems. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW '02)*, pages 96–103, Greenbelt, USA, December 2002. IEEE.

[31] Peter H. Deussen, George Din, and Ina Schieferdecker. A TTCN-3 based online test and validation platform for Internet services. In *Proceedings of the Sixth International Symposium on Autonomous Decentralized Systems (ISADS 2003)*, pages 177–184, Pisa, Italy, April 2003. IEEE.

[32] Massimiliano Di Penta, Marcello Bruno, Gianpiero Esposito, Valentina Mazza, and Gerardo Canfora. *Web Services Regression Testing*, pages 205–234. Springer, 2007.

[33] Dimitris Dranidis, Andreas Metzger, and Dimitrios Kourtesis. Enabling proactive adaptation through just-in-time testing of conversational services. In Elisabetta Di Nitto and Ramin Yahyapour, editors, *Proceedings of the Third European Conference, ServiceWave 2010*, volume 6481 of *Lecture Notes in Computer Science*, pages 63–75, Ghent, Belgium, December 2010. Springer.

[34] Sebastian Elbaum and Madeline Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 65–75, Boston, USA, July 2004. ACM.

[35] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, January 2017.

[36] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. In Jens Grabowski and Brian Nielsen, editors, *Proceedings of 4th International Workshop on Formal Approaches to Software Testing (FATES 2004), Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15, Linz, Austria, September 2004. Springer.

[37] Erik M. Fredericks and Betty H. C. Cheng. Automated Generation of Adaptive Test Plans for Self-adaptive Systems. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*, pages 157–167, Florence, Italy, August 2015. IEEE.

[38] Erik M. Fredericks, Byron DeVries, and Betty H. C. Cheng. Towards Run-time Adaptation of Test Cases for Self-adaptive Systems in the Face of Uncertainty. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014)*, pages 17–26, Hyderabad, India, June 2014. ACM.

[39] Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. An Exploratory Study of Field Failures. In *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE 2017)*, pages 67–77, Toulouse, France, October 2017. IEEE.

[40] Okehee Goh and Yann-Hang Lee. Schedulable online testing framework for real-time embedded applications in vm. In Tei-Wei Kuo, Edwin Sha, Minyi Guo, Laurence T. Yang, and Zili Shao, editors, *Proceedings of the International Conference on Embedded and Ubiquitous Computing (EUC 2007)*, volume 4808 of *Lecture Notes in Computer Science*, pages 730–741, Taipei, Taiwan, December 2007. Springer.

[41] Alberto González, Eric Piel, and Hans-Gerhard Gross. Architecture support for runtime integration and verification of component-based systems of systems. In *Proceedings of the International Workshop on Automated engineeRing of Autonomous and run-tiMe evolving Systems (ARAMIS 2008)*, pages 41–48, L'Aquila, Italy, September 2008. IEEE.

[42] Alberto Gonzalez-Sanchez, Eric Piel, and Hans-Gerhard Gross. Ritmo: A method for runtime testability measurement and optimisation. In *Proceedings of the 9th International Conference on Quality Software (QSIC 2009)*, pages 377–382, Jeju, South Korea, August 2009. IEEE.

[43] Alberto Gonzalez-Sanchez, Eric Piel, Hans-Gerhard Gross, and Arjan J. C. van Gemund. Runtime Testability in Dynamic High-Availability Component-Based Systems. In *Proceedings of the Second International Conference on Advances in System Testing and Validation Lifecycle (VALID 2010)*, pages 37–42, Nice, France, August 2010. IEEE.

[44] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. Automatic runtime recovery via error handler synthesis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, pages 684–695, Singapore, Singapore, August 2016. ACM.

[45] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, volume 50 of *ACM SIGPLAN Notices*, pages 339–353, Istanbul, Turkey, March 2015. ACM.

[46] Zhan-Wei Hui, Song Huang, and Meng-Yu Ji. A runtime-testing method for integer overflow detection based on metamorphic relations. *Journal of Intelligent & Fuzzy Systems*, 31(4):2349–2361, 2016.

[47] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Test coverage of data-centric dynamic compositions in service-based systems. In *Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, pages 40–49, Berlin, Germany, March 2011. IEEE.

[48] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Testing of data-centric and event-based dynamic service compositions. *Software Testing, Verification and Reliability*, 23(6):465–497, September 2013.

[49] Melody Y. Ivory and Marti A. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4):470–516, December 2001.

[50] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pages 474–484, Zurich, Switzerland, June 2012. IEEE.

[51] Katsumi Kawano, Masayuki Orimo, and Kinji Mori. Autonomous decentralized system test technique. In *Proceedings of the 13th Annual International Computer Software & Applications Conference (COMPSAC 89)*, pages 52–57, Orlando, USA, September 1989. IEEE.

[52] Fitsum Meshesha Kifetew, Wei Jin, Roberto Tiella, Alessandro Orso, and Paolo Tonella. Reproducing field failures for programs with complex grammar-based input. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, (ICST 2014)*, pages 163–172, Cleveland, USA, March 2014. IEEE.

[53] Tariq M. King, Andrew A. Allen, Rodolfo Cruz, and Peter J. Clarke. Safe Runtime Validation of Behavioral Adaptations in Autonomic Software. In Jose M. Alcaraz Calero, Laurence T. Yang, Félix Gómez Mármol, and Luis Javier García Villalba, editors, *Proceedings of the 8th International Conference on Autonomic and Trusted Computing (ATC 2011)*, volume 6906 of *Lecture Notes in Computer Science*, pages 31–46, Banff, Canada, September 2011. Springer.

[54] Tariq M. King, Djuradj Babich, Jonatan Alava, Peter J. Clarke, and Ronald Stevens. Towards self-testing in autonomic computing systems. In *Proceedings of the Eigth International Symposium on Autonomous Decentralized Systems (ISADS 2007)*, pages 51–58, Sedona, USA, March 2007. IEEE.

[55] Tariq M. King, Alain E. Ramirez, Rodolfo Cruz, and Peter J. Clarke. An integrated self-testing framework for autonomic computing systems. *Journal of Computers*, 2(9):37–49, 2007.

[56] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, July 2007.

[57] Ron Kohavi and Roger Longbotham. Online controlled experiments and a/b testing. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of machine learning and data mining*, pages 922–929. Springer, 2017.

[58] Mariam Lahami, Fairouz Fakhfakh, Moez Krichen, and Mohamed Jmaiel. Towards a TTCN-3 test system for runtime testing of adaptable and distributed systems. In Brian Nielsen and Carsten Weise, editors, *Proceedings of the 24th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2012)*, volume 7641 of *Lecture Notes in Computer Science*, pages 71–86, Aalborg, Denmark, November 2012. Springer.

[59] Mariam Lahami and Moez Krichen. Test isolation policy for safe runtime validation of evolvable software systems. In *Proceedings of the Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2013)*, pages 377–382, Hammamet, Tunisia, June 2013. IEEE.

[60] Mariam Lahami, Moez Krichen, Mariam Bouchakwa, and Mohamed Jmaiel. Using knapsack problem model to design a resource aware test architecture for adaptable and distributed systems. In Brian Nielsen and Carsten Weise, editors, *Proceedings of the 24th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2012)*, volume 7641 of *Lecture Notes in Computer Science*, pages 103–118, Aalborg, Denmark, November 2012. Springer.

[61] Mariam Lahami, Moez Krichen, and Mohamed Jmaiel. A distributed test architecture for adaptable and distributed real-time systems. *Revue des Nouvelles Technologies de l'Information*, RNTI-L-6:73–92, 2012.

[62] Mariam Lahami, Moez Krichen, and Mohamed Jmaiel. Runtime testing framework for improving quality in dynamic service-based systems. In *Proceedings of the 2nd International Workshop on Quality Assurance for Service-based Applications (QASBA 2013)*, pages 17–24, Lugano, Switzerland, July 2013. ACM.

[63] Mariam Lahami, Moez Krichen, and Mohamed Jmaïel. Runtime Testing Approach of Structural Adaptations for Dynamic and Distributed Systems. *International Journal of Computer Applications in Technology*, 51(4):259–272, July 2015.

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

[65] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit. Lessons learned at 208K: Towards debugging millions of cores. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*, pages 1–9, Austin, USA, November 2008. IEEE.

[66] Kihwal Lee and Lui Sha. A dependable online testing and upgrade architecture for real-time embedded systems. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005)*, pages 160–165, Hong Kong, China, August 2005. IEEE.

[67] Seoung-Hyeon Lee and Jung-Chan Na. Development of test agents for automated dynamic testing of uniway. In James J. (Jong Hyuk) Park, Shu-Ching Chen, and Kim-Kwang Raymond Choo, editors, *Proceedings of the International Conference on Advanced Multimedia and Ubiquitous Engineering (MUE/-FutureTech 2017)*, volume 448 of *Lecture Notes in Electrical Engineering*, pages 683–688, Seoul, South Korea, May 2017. Springer.

[68] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[69] Riku Luostarinen, Risto Järvinen, Juho Määttä, and Jukka Manner. A model for usability testing in challenging environments. In *Proceedings of the International Conference on Military Communications and Information Systems (ICMCIS 2016)*, pages 1–6, Brussels, Belgium, May 2016. IEEE.

[70] Bo Ma, Bin Chen, Xiaoying Bai, and Junfei Huang. Design of bdi agent for adaptive performance testing of web services. In *Proceedings of the 10th International Conference on Quality Software (QSIC 2010)*, pages 435–440, Zhangjiajie, China, July 2010. IEEE.

[71] Afef Jmal Maâlej, Moez Krichen, and Mohamed Jmaïel. Model-Based Conformance Testing of WS-BPEL Compositions. In *Proceedings of the Fourth IEEE International Workshop on Software Testing Automation (STA 2012)*, pages 452–457, Izmir, Turkey, July 2012. IEEE.

[72] Lijun Mei, W. K. Chan, T. H. Tse, Bo Jiang, and Ke Zhai. Preemptive regression testing of workflow-based web services. *IEEE Transactions on Services Computing*, 8(5):740–754, September 2015.

[73] Andreas Metzger, Osama Sammodi, Klaus Pohl, and Mark Rzepka. Towards pro-active adaptation with confidence: augmenting service monitoring with online testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10)*, pages 20–28, Cape Town, South Africa, May 2010. ACM.

[74] Jesús Morán, Antonia Bertolino, Claudio de la Riva, and Javier Tuya. Towards Ex Vivo Testing of MapReduce Applications. In *Proceedings of the 2017 International Conference on Software Quality, Reliability and Security (QRS 2017)*, pages 73–80, Prague, Czech Republic, July 2017. IEEE.

[75] Christian Murphy, Gail E. Kaiser, Ian Vo, and Matt Chu. Quality assurance of software applications using the in vivo testing approach. In *Proceedings of the Second International Conference on Software Testing Verification and Validation (ICST 2009)*, pages 111–120, Denver, USA, April 2009. IEEE.

[76] Christian Murphy, Kuang Shen, and Gail Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 189–200, Chicago, USA, July 2009. ACM.

[77] Christian Murphy, Moses Vaughan, Waseem Ilahi, and Gail Kaiser. Automatic detection of previously-unseen application states for deployment environment testing and analysis. In *Proceedings of the 5th Workshop on Automation of Software Test (AST 2010)*, pages 16–23, Cape Town, South Africa, May 2010. ACM.

[78] Vânia de Oliveira Neves, Márcio Eduardo Delamaro, and Paulo Cesar 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.

[79] Vânia de Oliveira Neves, Márcio Eduardo Delamaro, and Paulo Cesar Masiero. Automated structural software testing of autonomous vehicles. In *Proceedings of the 20th Iberoamerican Conference on Software Engineering (CibSE 2017)*, volume 1, pages 15–28, Buenos Aires, Argentina, May 2017. Curran Associates.

[80] Dirk Niebuhr, Andreas Rausch, Cornel Klein, Juergen Reichmann, and Reiner N. Schmid. Achieving dependable component bindings in dynamic adaptive systems - A runtime testing approach. In *Proceedings of the Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2009)*, pages 186–197, San Francisco, California, USA, September 2009. IEEE.

[81] Eiji Nishijima, Hiroshi Yamamoto, Katsumi Kawano, Kazunori Fujiwara, and Keiji Oshima. On-line testing for application software of widely distributed system. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS '96)*, pages 54–63, Niagara-on-the-Lake, Canada, October 1996. IEEE.

[82] Marc Oriol, Xavier Franch, and Jordi Marco. Monitoring the service-based system lifecycle with salmon. *Expert Systems with Applications*, 42(19):6507–6521, November 2015.

[83] Youngki Park, Woosung Jung, Byungjeong Lee, and Chisu Wu. Automatic discovery of web services based on dynamic black-box testing. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*, volume 1, pages 107–114, Seattle, USA, July 2009. IEEE.

[84] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, August 2015.

[85] Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[86] Éric Piel and Alberto Gonzalez-Sanchez. Data-flow integration testing adapted to runtime evolution in component-based systems. In *Proceedings of the 2009 ESEC/FSE workshop on Software Integration and Evolution @ Runtime (SINTER'09)*, pages 3–10, Amsterdam, The Netherlands, August 2009. ACM.

[87] Adam Porter, Cemal Yilmaz, Atif M. Memon, Douglas C. Schmidt, and Bala Natarajan. Skoll: A process and infrastructure for distributed continuous quality assurance. *IEEE Transactions on Software Engineering*, 33(8):510–525, 2007.

[88] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. Reconstructing core dumps. In *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation, (ICST 2013)*, pages 114–123, Luxembourg, Luxembourg, March 2013. IEEE.

[89] Osama Sammodi, Andreas Metzger, Xavier Franch, Marc Oriol, Jordi Marco, and Klaus Pohl. Usage-Based Online Testing for Proactive Adaptation of Service-Based Applications. In *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2011)*, pages 582–587, Munich, Germany, July 2011. IEEE.

[90] Gerald Schermann, Jürgen Cito, and Philipp Leitner. Continuous experimentation: Challenges, implementation techniques, and current research. *IEEE Software*, 35(2):26–31, 2018.

[91] Carol Smidts, Chetan Mutha, Manuel Rodriguez, and Matthew J. Gerber. Software testing with an operational profile: Op definition. *ACM Computing Surveys*, 46(3), February 2014.

[92] Mozhan Soltani, Pouria Derakhshanfar, Xavier Devroey, and Arie van Deursen. A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering*, 25(1):96–138, January 2020.

[93] Ronald Stevens, Brittany Parsons, and Tariq M. King. A self-testing autonomic container. In *Proceedings of the 45th Annual Southeast Regional Conference (ACM-SE 45)*, pages 1–6, Winston-Salem, USA, March 2007. ACM.

[94] Dima Suliman, Barbara Paech, Lars Borner, Colin Atkinson, Daniel Brenner, Matthias Merdes, and Rainer Malaka. The MORABIT approach to runtime component testing. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*, volume 2, pages 171–176, Chicago, USA, September 2006. IEEE.

[95] Jüri Vain, Leonidas Tsiopoulos, Vyacheslav Kharchenko, Apneet Kaur, Maksim Jenihhin, and Jaan Raik. Multi-fragment Markov model guided online test generation for MPSoC. In *Proceedings of the 3rd International Workshop on Theory of Reliability and Markov Modeling for Information Technologies (TheRMIT 2017)*, pages 594–607, Kyiv, Ukraine, May 2017. CEUR-WS.org.

[96] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, volume 30 of *ACM SIGSOFT Software Engineering Notes*, page 273–282, Lisbon, Portugal, September 2005. ACM.

[97] Arnold P. O. S. Vermeeren, Effie Lai-Chong Law, Virpi Roto, Marianna Obrist, Jettie Hoonhout, and Kaisa Väänänen-Vainio-Mattila. User experience evaluation methods: Current state and development needs. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries (NordiCHI 2010)*, pages 521–530, Reykjavik, Iceland, October 2010. ACM.

[98] Yingxu Wang, Graham King, Dilip Patel, Shushma Patel, and Alec Dorling. On coping with real-time software dynamic inconsistency by built-in tests. *Annals of Software Engineering*, 7:283–296, Oct 1999.

[99] Yongbo Wang, Xiaoying Bai, Juanzi Li, and Ruobo Huang. Ontology-based test case generation for testing web services. In *Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems (ISADS 2007)*, pages 43–50, Sedona, USA, March 2007. IEEE.

[100] Hiroshi Yamamoto, Akira Yoshizawa, Katsumi Kawano, Kinji Mori, Keiji Oshima, and Tsunao Kawamura. On-line software test techniques based on autonomous decentralized system. In *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems (FTDCS)*, pages 291–296, Lisbon, Portugal, September 1993. IEEE.

[101] Chunyang Ye and Hans-Arno Jacobsen. Whitening SOA testing via event exposure. *IEEE Transactions on Software Engineering*, 39(10):1444–1465, 2013.

[102] Jia Zhang. An approach to facilitate reliability testing of web services components. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 210–218, Saint-Malo, France, November 2004. IEEE.

[103] Hong Zhu. A framework for service-oriented testing of web services. In *Proceedings of the Third International Workshop on Quality Assurance and Testing Web-Based Applications (QATWBA'06)*, volume 2, pages 145–150, Chicago, USA, September 2006. IEEE.

[104] Hong Zhu and Yufeng Zhang. Collaborative testing of web services. *IEEE Transactions on Services Computing*, 5(1):116–130, 2012.

[105] Hong Zhu and Yufeng Zhang. A test automation framework for collaborative testing of web service dynamic compositions. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Advanced Web Services*, pages 171–197. Springer, 2014.

Online Supplemental Material: A Survey of Field-based Testing Techniques

This online appendix details the presentation of the approaches referred in Section 5 and it proposes a deeper analysis about their classification within the framework proposed in the current survey.

A RQ1.1: Approaches, Fault Types, Test Case Generation and Platforms

Table 8: Testing techniques

Testing approach		Test Generation Strategy: *Specification-based*	*Structure-Based*	*Fault-Based*	*Pre-existing*
ex-vivo	*functional*	Section A.1.1 *Mutation of Field Executions* [74, 78, 28, 79] *Test Suite Adaptation* [48, 47, 38]	Section A.1.2 *Profile Data* [34]		Section A.1.3 *Field Triggers* [72]
offline	*functional*	Section A.2.1 *IO Data Pattern* [81] *Metamorphic Relations* [10, 76]			Section A.2.3 *Built-in Tests* [94, 98, 75, 54, 55, 93] *Test planning and management* [43, 42, 77] *Adaptation and Reconfiguration* [62, 64]
	non-functional	Section A.2.2 *Security Specifications* [25, 24]			
online	*functional*	Section A.3.1 *Choreographies and Service-based specifications* [22, 2, 12, 6, 99, 5] *Finite-State Models* [71, 95, 18, 33, 20] *Metamorphic Relations* [21] *Graph grammars* [83]	Section A.3.2 *Event Interface* [101]	Section A.3.3 *Fault Injection* [102]	Section A.3.4 *Test planning and management* [60, 1, 59, 14, 27, 43, 42] *Adaptation and Reconfiguration* [44, 53, 64, 32, 62] *Isolation* [17]
	non-functional	Section A.4.1 *Stochastic Models of User Behavior* [89, 82] *Security Specifications* [12, 26, 13, 46] *Usability Models* [69] *Δ-grammars for QoS* [83] Timed-automata [71, 95]		Section A.4.2 *Fault Injection* [3, 102] *Operational Profile* [73]	Section A.4.3 *Adaptation and Reconfiguration* [70, 32, 67]

Table 9: Platforms for field testing

Offline Testing Platforms		Online Testing Platforms	
Section A.5 *Evolution* [66, 61, 58, 63] *Shadow Instances for V&V* [45, 40]	*SOA and Component-Based Testing* [105, 104, 8, 16, 81, 103]	Section A.6 *Built-In and Pre-Existing Test Cases* [81, 51, 100, 31, 30, 41] *SOA Online Testing* [80]	*Evolution* [23, 58, 61, 63, 37] *Distributed Quality Assurance* [87]

We analyze field testing techniques by considering the *field testing approach* and the *test case generation strategy* dimensions of the classification. We distinguish between approaches that address functional and non-functional faults (Table 8), and classify approaches based on the target platforms (frameworks and architectures) (Table 9).

The studies are not evenly distributed with respect to the *testing approaches*: We found few ex-vivo and offline field testing techniques, and many online field testing techniques. This indicates a strong interest towards techniques that test exactly the operational application.

The nature of the *requirements* that are tested is not evenly distributed within each testing approach. Most ex-vivo and offline field testing approaches focus on functional requirements. Only two approaches address offline testing of security requirements to assess the security of software systems in the operational environment as early as possible [25, 24]. Online testing approaches address both functional and non-functional requirements.

Concerning the problem of *test case generation*, in field testing a test case has a broader scope than just the test input data. The actual novel challenges in generating field tests descend from identifying and reproducing with each test case the relevant interactions with the field, whereas a-priori we do not expect that new, ad-hoc techniques need to be specified for generating the test inputs data. We cluster techniques in four categories [85] depending on the information used to generate the test cases, as is also done for

in-house testing: *(i) Specification-based* techniques derive test cases from formal or informal requirements specification; *(ii) Structure-based* techniques use structural information, mostly the code structure, to derive test cases; *(iii) Fault-based* techniques use models of potential faults (*fault models*) to derive test cases that address the faults represented in the fault model; *(iv)* Techniques working with *pre-existing* test cases exploit already available test cases. What we observed from our survey of literature is that most commonly black-box techniques are used. Namely either test cases are generated from specifications, to suitably cover certain behaviors, or strategies are defined to execute test cases that are already available, for instance from field monitoring. Only few approaches derive test cases from structural information and fault-models.

The available platforms, as shown in Table 9, offer a range of solutions for both offline and online testing targeting several environments (e.g., service and component-based systems) and scenarios (e.g., continuous quality assurance and evolution).

The classification in Tables 8 and 9 provides a roadmap to researchers, who can easily understand the positioning of the individual studies with respect to the existing literature along the identified dimensions (testing approach, test generation strategy) and along the respective sub-dimensions. It is also potentially useful to practitioners, when they face a specific field testing problem. Thanks to the presented classification, they can find relevant research papers and tools that address the kind of field testing approach they want to implement (ex-vivo, offline, online) and the kind of test generation strategy they intend to adopt (structural, fault oriented or pre-existing tests).

A.1 Ex-vivo Functional Testing

Ex-vivo testing techniques exploit information from the field to improve the in-house testing process. The main idea is that field information can help testers direct their effort towards the scenarios that matter the most in practice, and thus require a more extensive and thorough form of validation compared to other scenarios. Research on ex-vivo testing focuses on functional test cases generated from specifications, code structure or pre-existing tests.

A.1.1 Specification-Based Approaches

Ex-vivo testing uses specifications either to *mutate field executions* and turn them into new (in-house) test cases or to guide the *test suite adaptation* process. Note that executions collected from the field are normally modified, not only to make them fit the in-house context, but also to be exploited as seed for new test cases that resemble without being identical to the already observed field usages.

Mutation of field executions has been investigated by both Neves et al. [28, 78, 79] in the context of autonomous vehicles software and Morán et al. [74] in the context of MapReduce distributed data processing applications. Neves et al. [28, 78] use vehicles' sensor data to reveal the position of the vehicle while the software is running and exploit *mutations* to produce new test cases to improve the coverage of the execution space. Morán et al.'s approach [74] samples production data to execute offline the computations observed in the field, with the same data but under different mapper/reducer task configurations. If different executions produce different results, the approach reports a bug. The test cases are designed in house by developers, while the oracles are obtained from program executions based on *metamorphic relations*.

Test suite adaptation has been investigated in the context of software with highly dynamic behavior, such as self-adaptive systems and dynamic service compositions. Fredericks et al. [38] propose to adapt test cases according to the evolutions of the system and the environment observed in the field. They identify the test cases to be adapted with a *goal-based specification* of the system and use a *utility function* to estimate the correctness of the behavior of the adaptive system. The adaptation process guarantees the relevance of the tests while preventing test cases to pass under invalid conditions. Hummer et al. [47, 48] address the problem of finding a minimum number of test cases to validate a dynamic service composition, by considering data-flow relationships between services. They encode the problem with a *data-model*, and use the information collected from the field to drive the test generation process towards compositions not validated in the field.

A.1.2 Structure-Based Approaches

Structure-based approaches generate in-house test cases by exploiting structural targets derived from the observation of field executions. Elbaum and Hardojo [34] studied the impact of *profiling* techniques on testing, and exploit user sessions to semi-automatically derive ex-vivo test cases that cover all the entities exercised in the field.

A.1.3 Pre-existing Test Cases

Some approaches generate new test cases by modifying the available ones with information extracted from the field. Mei et al. [72] consider the problem of regression testing of Web services that evolve so quickly that changes of external services occur during the regression testing session itself. They introduce a dynamic re-assessment of test priorities that may preempt an ongoing regression testing session, and opportunistically *trigger* a nested session aimed at covering the changed objectives.

A.2 Offline Testing

Offline testing techniques verify the functional behavior of the software in the field, while acting on an instance of the SUT separated from the one that is operational, looking for a compromise between isolation and direct exploitation of the field. The few offline testing techniques proposed so far either implement specification-based approaches or reuse existing test suites.

A.2.1 Specification-Based Approaches for Functional Requirements

Some approaches use specifications to generate functional test cases that can be executed offline while taking field information into consideration. Nishijima et al. [81] propose an offline testing approach to verify the functional behavior of distributed autonomous (real-time and non-real-time) systems. Nishijima et al.'s solution exploits built-in tests and deploy a test mode of the SUT to safely run test cases. They derive test cases from *input-output data patterns* that specify the behavior of the components and their interactions. The test cases look for unexpected side-effects of the field on the exercised interactions.

Metamorphic testing approaches generate new test inputs and oracles from observed executions, by using metamorphic relations. Murphy et al. [10, 76] move metamorphic testing to the production environment by capturing executions at the level of both individual functions and applications to trigger additional executions that can exercise the SUT in unexpected situations. Murphy et al.'s approach executes a copy of the application in a sandbox, and checks the behavior of the application with oracles derived from metamorphic relations, interestingly expanding the scope and scale of classic metamorphic testing.

A.2.2 Specification-Based Approaches for Non-Functional Requirements

Some offline testing approaches address *security requirements*. Dai et al. propose fuzzing to generate test cases that can detect vulnerabilities due to configuration changes [24, 25]. Dai et al.'s approach executes a set of manually identified functions of the program in a sandboxed environment with fuzzed configurations. Failures are detected by security oracles that testers design as program invariants.

A.2.3 Pre-Existing Test Cases

Some software units such as components and services include *built-in test* suites that are opportunistically executed when activating a test-mode behavior on the SUT. When the tests are executed in the field typical issues may concern the validation of the impact in production and the status of the components involved in the computations on the correctness of the execution.

Both Wang et al. [98] and Murphy et al. [75] enhance object-oriented classes with built-in test cases that are executed in test-mode. Suliman et al. [94] propose built-in tests to assess the confidence and the reliability of software components when running in the field, while Piel et al. [86] define built-in tests to validate the component integration during the system evolution.

King et al. [54, 55] and Stevens et al. [93] study built-in test cases to dynamically validate the changes produced by autonomic software in the field, before the changes are finalized.

Several studies propose strategies for *test planning and managing* field tests. Murphy et al. [77] define a strategy to identify untested application states at run-time, to prevent executing redundant test cases. Gonzalez-Sanchez et al. [43, 42] propose a metric to empirically estimate the online testability of a software system. They propose a testability analysis to chose an appropriate field testing approach. Lahami et al. [62, 64] propose a technique to validate architectural reconfigurations: based on a dependency analysis of the components involved, they identify a set of test cases to be executed online before completing a reconfiguration, meanwhile requests that the component under test would receive are delayed until the online testing process is completed.

A.3 Online Functional Testing

Online functional testing techniques generate test cases to be executed in the production environment to validate the functional behavior of the software system.

A.3.1 Specification-Based Approaches

Specification-based approaches use different kinds of specifications to generate field test cases to be executed in the production environment. In particular, specification-based approaches use *choreographies* to generate field tests for service-based applications, *finite-state models* to generate field tests for stateful components, *metamorphic relations* to generate field tests that compare outputs across executions, and *graph grammars* to test web services.

Choreographies and service-based specifications Several techniques exploit *choreographies and service-based specifications* for testing service-based applications. Bai et al.'s approach models web service test with OWL-S, and generates test cases through partition testing based on ontologies [6, 99, 5]. Ali et al. propose a testing framework for service choreographies specified in BPMN [2]. The framework supports a model-based approach for automatically generating, storing and executing test cases, to trustworthiness rate service choreographies and individual subscribing services. Cooray et al. [22] propose an approach to dynamically test the reconfiguration of composite web services exploiting a stochastic *usage model* of the services.

Bertolino et al. [12] propose the PLASTIC framework for testing service-oriented applications. PLASTIC supports the validation of both functional and extra-functional properties of networked services, spanning over both in-house and field testing stages. The framework supports online test case generation from Symbolic Transition System (STS) specifications that capture both the structure of the interfaces and the messages allowed across interfaces.

Finite state models Other techniques generate online test cases from *finite state models*. Dranidis et al. derive test cases from Stream X-Machines (SXMs), a kind of finite state machines that model both the control-flow and the data-flow of a software system [33]. Dranidis et al. propose just-in-time testing of conversational web services to ensure that the invocation protocol works before executing a service composition. They assume services to be testable without side effects.

Maâlej et al. propose a technique for online conformance testing of BPEL compositions, based on Timed Automata [71]. They define an algorithm for generating and executing test cases implemented in the WSCCT WS-BPEL Compositions Conformance Testing tool. Similarly, Cao et al. [20] convert BPEL specifications into timed extended finite state machines that are used to derive test cases. The test cases are executed online on the web service composition under test, by simulating external services with service mocks.

Vain et al. map Multi-Fragment Markov Models (MFMMs) to Uppaal Probabilistic Timed Automata (UPTA), for online monitoring and model-based testing [95]. They map MFMMs specifications of reliability and security-related behaviors to UPTA fine-grain states and timing constraints, and use the UPTA models to generate conformance tests of the operation modes specified with MFMM.

Brenner et al. enhance the verification process of component-based systems with built-in tests [18]. They automate the testing process at component level, to enable runtime testing of large systems in the presence of configuration changes. The runtime testing leverages Markov chains to represent states and operations, along with the frequency of invocation.

Metamorphic Relations Chan et al. [21] define metamorphic services that encapsulate the SUT and generate online test cases from *metamorphic relations* provided by the testers, aiming to reveal regressions and harmful changes in the environment.

Graph grammars Park et al. [83] generate online test cases from Δ-grammars specifications of the functional services.

A.3.2 Structure-Based Approaches

Structure-based approaches leverage structural information to derive the test cases to be executed online. Chunyang et al. [101] propose a method for white-box testing of service compositions with minimal exposure of information about the internal implementations of the participating services. They expose an *event interface* that service consumers use to check the white-box coverage of the used service. Chunyang et al.'s approach defines the information that services shall expose depending on the desired coverage, namely branch, path, or data-flow coverage, and generates the test cases by random and constraint-based techniques.

A.3.3 Fault-Based Approaches

Fault-based approaches generate test cases from fault models. Zhang's approach [102] generates fault-based test cases for dynamically aggregated web services. The approach targets the problem of determining whether a discovered web service can be integrated into a system without disrupting the reliability of the system as a whole. It addresses the problem by testing both the discovered web service in isolation, to determine whether the service is reliable, and the web service communications, to determine whether the communications are robust. The approach assesses the reliability of the discovered service by *injecting faults* into correct inputs, and executing them to determine if the new service is vulnerable to corrupted inputs. It assesses the robustness of the communication by integrating the reliable service into the system, executing it with the injected faults, and monitoring the effects of fault propagation.

A.3.4 Pre-existing Test Cases

Several field testing techniques focus on the execution and maintenance of the available test cases, which can be either the same as the in-house test cases or designed *ad-hoc* for field testing.

Test planning and Management. Planning and management techniques focus on strategies to schedule test execution based on information available in the field. Lahami et al. [60, 59] define strategies to properly interleave the normal operation of the software with the execution of online test cases, to reduce the interference between these two processes. Bertolino et al. [27, 14] propose a governance framework for V&V activities in the context of service choreographies. Bertolino et al.'s framework includes policies for activating, rating, ranking, and enacting choreographies, and for selecting test cases (we further discuss test governance in Section 9.2). Akour et al. [1] define a model-driven approach to maintain test cases by removing the test cases that are no longer applicable due to changes that occurred in the system. Gonzalez-Sanchez et al. [43, 42] define metrics to estimate the software testability, as we already discussed for offline techniques (see Section A.2.3).

Adaptation and Reconfiguration. Field testing techniques are used to validate reconfigurations and adaptations either before or immediately after changes are finalized. Gu et al. [44] use online testing to validate the recovery actions synthesized for handling Java exceptions in a sandboxed environment before exploiting them. King et al. [53] and Piel et al. [86] test adaptation and evolution among interconnected components by running available test cases. The approach proposed by Lahami et al. [62, 64] and already presented in Section A.2.3 applies both offline and online functional testing techniques. Di Penta et al. [32] propose a technique for online regression testing of service changes based on a set of available unit test cases. The approach monitors component interactions to prevent the execution of redundant test cases.

Isolation. Some solutions specifically address isolation. For instance, Bobba et al. [17] investigate the use of transactional memory to isolate the testing process from the system in operation.

A.4 Online Non-Functional Testing

Online non-functional testing techniques generate test cases to be executed in the production environment to validate non-functional properties of the SUT.

A.4.1 Specification-Based Approaches

In this section, we discuss the approaches that derive test cases from specifications of non-functional requirements, for the different non-functional properties.

Stochastic Models of User Behavior. Sammodi et al. [89] capture the user behavior with stochastic models that can be used to generate test cases for assessing the Quality of Service (QoS) of service-based applications. Oriol et al. combine test cases from stochastic models with passive monitoring to improve QoS attributes [82].

Security Specifications. De Angelis et al. propose (Role)CAST, a framework for online testing federations of services in their execution context, while the server is engaged in serving operational requests. The framework focuses on authentication, authorization, identification features [26], and general role-based access policies [13]. De Angelis et al. generate test cases from UML-based specifications. Bertolino et al.'s PLASTIC framework [12], already presented in Section A.3.1, also addresses non-functional properties of networked services, with test cases derived from Symbolic Transition System (STS) specifications. Hui et al. [46] use online metamorphic testing to improve software security by exercising paths to security sensitive sinks with additional untrusted values obtained from metamorphic relations.

Usability Models. Luostarinen et al. propose a model-based approach for remotely testing a messaging platform in the military domain using a controller that can run test sessions online, engaging the users [69].

Graph Grammars for QoS. Park et al. select test cases from graph grammar specifications of QoS attributes to test if target services satisfy specific qualities [83].

Timed Automata. Both Vain et al. [95] and Maaley et al. [71] generate online test cases from timed automata specifications to sample the functional and the timing behaviors of the SUT.

A.4.2 Fault-Based Approaches

Fault-based approaches derive test suites based on fault-models.

Fault Injection. Fault-injection approaches aim to assess the capability of software systems to react to unexpected and faulty situations. Zhang's approach [102], which we already discussed in Section A.3.3, addresses both functional and nonfunctional quality attributes, with a focus on vulnerability and interoperability issues within a federation of web services.

Alnawasreh et al. [3] focus on the problem of testing the robustness of distributed embedded systems in the presence of spurious or incorrect communication via message passing. Their *Postmonkey* approach alters the communication between components by both introducing random delays of message delivery and randomly injecting invalid messages.

Operational Profile. Metzger et al. [73] propose an online pro-active testing approach for service-based applications that aims to predict failures with a known confidence. The approach monitors the applications for failures, and generates test cases with search-based strategies.

A.4.3 Pre-existing Test Cases

Some online testing approaches support the adaptation and reconfiguration of the software. Di Penta et al. [32] rely on available test cases to assess the non-regression of the non-functional characteristics of web services. Lee and Na [67] exploit available test *scripts* and data for automatically testing unidirectional communication system online. Test data are prepared at design time and the user, at runtime, manually selects the test to run. Ma et al. [70] propose a multi-agent framework for continuous performance testing of service-oriented software. Ma et al.'s *Test Coordinator* agent orchestrates multiple *Test Runner* agents. The Test Coordinator activates the Test Runners that execute test cases online to expose performance issues, such as slow response time under specific load conditions.

A.5 Platforms for Offline Testing

Platforms for offline field testing provide architectural solutions to execute test cases on separated instances of the software under test.

Evolution. Multiple platforms support offline testing activities in response to software evolution.

Lee and Sha [66] present an architecture for offline testing of upgrades in real-time embedded systems. Lee and Sha's solution is based on the Simplex architecture, which allows to execute multiple and diverse implementations of the same components. Lee and Sha's solution takes advantage of this capability to execute both the original and the upgraded component, and restores the original component in the presence misbehaviors.

Lahami et al. propose a testing framework for the evolution of dynamic distributed (real-time) systems [61, 58]. Lahami et al.'s framework manages test execution in response to reconfiguration events. By dependency analysis, it identifies the components involved in the reconfiguration and then executes the available TTCN-3 test cases [58]. The framework executes the test cases according to a resource-aware test plan generated at runtime [63].

Shadow Instances for V&V. Some platforms maintain shadow instances of the operative software, for verification and validation (V&V) activities. In particular, Hosek and Cadar [45] propose an architecture that replicates field executions on a shadow instance of the same application, which can be tested and analyzed without any risk of interfering with the original application. Hosek and Cadar's approach captures and replays executions at the system call level. Goh et al. [40] propose an online virtual execution environment to test a replica of an application. They test the replica in the virtual execution environment with states and input data monitored from the operative system, with which the operators substitute the replica after successfully completing the tests.

SOA and Component-Based Software. Several approaches propose multiple offline platforms to address the specific cases of service and component-based software systems.

Zho and Zhang [105, 104] propose a framework for field testing of web services. The framework adopts the SOA paradigm, and is centered on a broker service that facilitates the collaboration among different online testing services, such as test execution environment and test driver services. Zhu [103] proposes an ontology to mediate the information exchanged among the testing services. Zhu's approach executes the available test cases on an equivalent version of the operational SUT.

Bartolini et al. [8] define a framework to execute testing sessions on available Web Services, just before their deployment. Bartolini et al.'s approach requires an instrumented version of the deployed service that they use for testing sessions with the available test cases. An orchestrator leverages coverage reports from previous testing sessions to infer changes on services externals, without information about their internals.

Bhanu et al. [16] propose a general architecture for offline testing of embedded software in the context of safety critical system. Nishijima et al. [81] present an architecture for distributed autonomous systems and a testing mechanism composed of off-line and on-line testing techniques. Off-line tests check functional specifications, while on-line tests check the communication and other non-functional aspects of the live system.

A.6 Platforms for Online Testing

Online testing platforms support testing activities targeting the operational instance of the SUT.

Built-In and Pre-Existing Test Cases. Several architectural solutions offer features to maintain and execute tests in the form of either test cases built in software components or external test suites.

Kawano et al. [81, 51, 100] propose an architecture to support online testing with test cases both built in the autonomous components and generated with dynamically activated dedicated modules. Deussen et al. [31, 30] propose an architecture for validating systems online, by monitoring and controlling the target distributed system, and executing the available TTCN-3 test cases. González et al. [41] propose the ATLAS framework for integration testing of component-based systems. The framework requires components to expose both built-in interfaces for testing and acceptance interfaces for managing test cases.

SOA Online Testing. Niebuhr et al. [80] propose an integration infrastructure for verifying the correct binding of components at runtime. Service consumers specify both the components to be tested and the test cases to be executed. The integration infrastructure executes the test cases, and prunes the components to be integrated according to the test results.

Evolution. Several platforms focus on managing software evolution, most of which support the execution of validation activities in the field in response to changes.

Da Costa et al. [23] propose *JAAF+T*, an extension of the *Java self-Adaptive Agent Framework* (JAAF), a framework for the implementation of self-adaptive agents. JAAF+T extends the typical Monitor, Analyze, Plan and Execute (MAPE) loop with a *testing* activity that checks adapted behaviors before their execution. Lahami et al. propose a hybrid offline/online framework for runtime testing of distributed (real-time) systems [61, 58], as already discussed in Section A.5.

Fredericks et al. [37] present Proteus, a framework for online testing of self-adaptive systems that addresses the evolution of online test suites, by adjusting test parameters and making sure that test suites remain relevant to changes at runtime.

Distributed Quality Assurance. Distributed quality assurance platforms exploit the participation of multiple nodes to the same infrastructure to perform large-scale studies aimed at assessing the quality of a SUT. Interestingly, Skoll envisions the direct participation of the users with their machines as the nodes of this infrastructure [87]. In this way, quality assurance tasks, including testing tasks, can be distributed among participants taking their environments into consideration. This solution guarantees not only scalability, but also the capability to assess the SUT against many diverse real configurations and environments, effectively covering a range of situations otherwise infeasible to cover in house.

A.7 RQ1.1: Findings

We conclude this section summarizing the key findings with respect to the type of addressed faults, the test case generation strategies, and the platforms proposed so far:

• *Approaches privilege functional versus non-functional faults, which are still under-investigated:* The majority of approaches address functional faults. The few approaches that address efficiency, security, reliability and usability, shed some initial light on a largely unexplored domain that calls for non-intrusive testing techniques to properly address non-functional properties.

• *Quality of service is a relatively well-studied quality attribute among non-functional aspects:* Many of the approaches that deal with non-functional properties address the relevant problem of predicting the QoS of applications executed in different and heterogenous production environments.

• *Specification-based test case generation approaches are by far the most studied approaches:* Many approaches rely on specifications to generate field test cases, leaving open the hard problem of generating test cases in absence of specifications, as in the many cases of systems that evolve beyond the initial specifications to adapt to emerging execution conditions and configurations.

• *Automation is still limited:* Many approaches rely on relevant human contribution and already available test cases, and automation of field testing is still limited.