A Survey of Field-based Testing Techniques

ANTONIA BERTOLINO, CNR-ISTI, Italy
PIETRO BRAIONE, University of Milano-Bicocca, Italy
GUGLIELMO DE ANGELIS, CNR-IASI, Italy
LUCA GAZZOLA, University of Milano-Bicocca, Italy
FITSUM KIFETEW, Fondazione Bruno Kessler (FBK), Italy
LEONARDO MARIANI, University of Milano-Bicocca, Italy
MATTEO ORRÙ, University of Milano-Bicocca, Italy
MAURO PEZZÈ, Università della Svizzera Italiana and Schaffhausen Institute of Technology, Switzerland
ROBERTO PIETRANTUONO, University of Naples Federico II, Italy
STEFANO RUSSO, University of Naples Federico II, Italy
PAOLO TONELLA, Università della Svizzera italiana, Switzerland

Contemporary software systems are highly-configurable and dynamic systems that interact with diverse execution environments. These systems raise many new validation challenges: Indeed in-house testing cannot completely verify such systems, and verification activities often continue at operational time in the field.

Field testing techniques are 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 software systems, but have not been systematically surveyed yet.

In this paper we present the first systematic survey of field testing. We survey a body of 79 papers, and propose a categorization of the many approaches based on both the environment and the system where field testing is performed. We discuss the surveyed studies, considering four research questions that address *how* software is tested in the field, *what* is tested in the field, which are the *requirements* to run tests in the field, and how field tests are *managed* in the field. The results and findings reported in this paper provide a roadmap for the researchers interested in the area, and propose a set of challenging research directions.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: software testing, field testing, in-vivo testing, ex-vivo testing.

ACM Reference Format:
Antonia Bertolino, Pietro Braione, Guglielmo De Angelis, Luca Gazzola, Fitsum Kifetew, Leonardo Mariani, Matteo Orrù, Mauro Pezzè, Roberto Pietrantuono, Stefano Russo, and Paolo Tonella. 0000. A Survey of Field-based Testing Techniques. *ACM Comput. Surv.* 0, 0, Article 0 (0000), 35 pages. https://doi.org/0000001.0000001

Authors' addresses: Antonia Bertolino, CNR-ISTI, Pisa, Italy, antonia.bertolino@isti.cnr.it; Pietro Braione, University of Milano-Bicocca, Milano, Italy, pietro.braione@unimib.it; Guglielmo De Angelis, CNR-IASI, Roma, Italy, guglielmo.deangelis@iasi.cnr.it; Luca Gazzola, University of Milano-Bicocca, Milano, Italy, luca.gazzola@disco.unimib.it; Fitsum Kifetew, Fondazione Bruno Kessler (FBK), Trento, Italy, kifetew@fbk.eu; Leonardo Mariani, University of Milano-Bicocca, Milano, Italy, leonardo.mariani@unimib.it; Matteo Orrù, University of Milano-Bicocca, Milano, Italy, matteo.orru@unimib.it; Mauro Pezzè, Università della Svizzera Italiana and Schaffhausen Institute of Technology, Lugano and Schaffhausen, Switzerland, mauro.pezze@usi.ch; Roberto Pietrantuono, University of Naples Federico II, Napoli, Italy, roberto.pietrantuono@unina.it; Stefano Russo, University of Naples Federico II, Napoli, Italy, sterusso@unina.it; Paolo Tonella, Università della Svizzera italiana, Lugano, Switzerland, paolo.tonella@usi.ch.

0360-0300/0000/0-ART0 $15.00
https://doi.org/0000001.0000001

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 [72].

Testing activities are common activities in 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, testing activities 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. [33] 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]. Test cases can be executed directly in the field on the same instance of the software used in production (*online field testing*), on a separated instance still running in production (*offline field testing*), or in-house with data collected from the field (*ex-vivo testing*).

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] 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 field failures in the nineties, with few studies addressing autonomic systems [44, 84], and real-time issues [83]. It attracted steady interest in the early years of the first decade of this century, with a sudden burst of results from 2007 on, leading to a relevant set of approaches not well contextualized yet.

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 79 distinct relevant studies since 1989. We discuss the characteristics of the different 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 testing in the field, 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 and indicates relevant open research directions.

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

[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.

2 METHODOLOGY

In this section, we present the methodology that we followed to identify the relevant papers.

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. We selected an initial set of papers 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 1238 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:

- *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 proposing techniques only weakly related to field testing, that is, techniques that deal with other issues and not on field testing per-se, such as runtime verification, cloud solutions for Testing-as-a-Service that offer scalable testing capabilities usually not including field testing capabilities, and adaptive testing that uses the outcome of a test to define the next test case to be executed, not necessarily performing this process in the field,
- *Exclusion Criterion 3*: studies about continuous experimentation [75], usability studies [43, 81] and performance testing based on operational profiles [77], which have been the subject of other surveys, and evolved independently from the notion of field testing,
- *Exclusion Criterion 4*: secondary or tertiary studies (e.g., systematic literature reviews and surveys),
- *Exclusion Criterion 5*: studies not available as full-text.

In a first pass, the authors of this paper 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 70 possibly relevant studies. We pruned this set with the inclusion and exclusion criteria, and added 31 new papers to our set of papers.

The process produced a set of **79 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

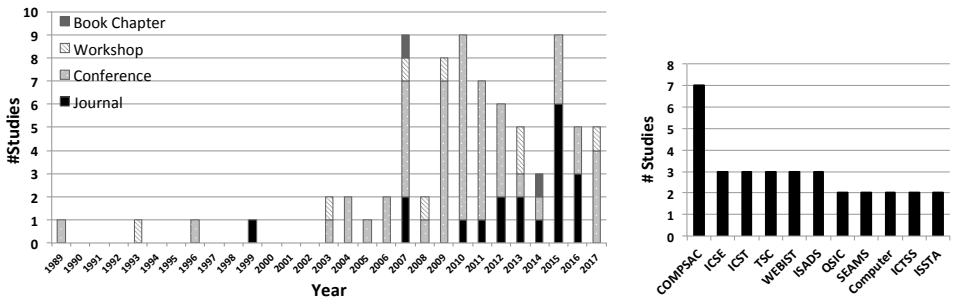
We identified several dimensions for analyzing the selected papers, based on the research questions presented in Section 4. For each paper, we checked if the presented study could be classified according to each identified dimension, and when possible we produced such a classification. We describe the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

We tuned the analysis process by assigning a small set of papers (14/79) to multiple authors, to obtain redundant classifications for each paper, and then discuss 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 rely 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

Figure 1a plots the selected studies by year and publication type. The publication years range from 1989 to 2017. 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 (50/79), followed by journals (19/79) and few workshop papers (8/79) and book chapters (2/79).

Figure 1b 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 79 studies. It is worth to note the presence of venues like TSC (*Transactions on Service Computing*) and WEBIST



(a) Field testing studies per year and venue type

(b) Field testing studies by venues

Fig. 1

(*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 is summarized in a spreadsheet that we make available for the interested researchers as supplemental online material.

3 FIELD TESTING

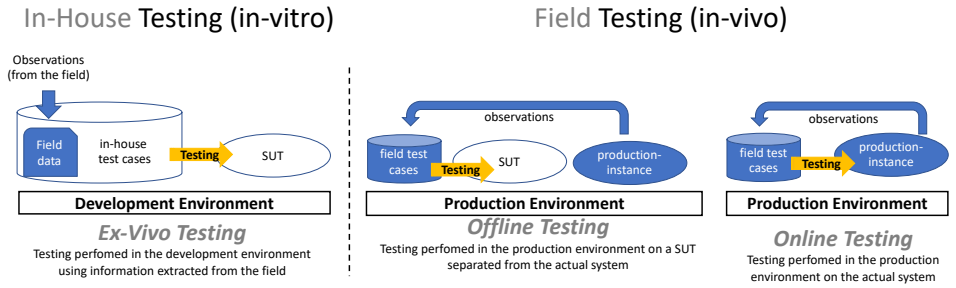


Fig. 2. Classes of field-testing approaches

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 environments specifically arranged to support the testing activities themselves, field testing refers to activities in 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. The same application can be deployed and used in multiple production environments, for instance an app can be installed in millions of different devices with different settings.

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 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. Figure 2 illustrates the main kinds of field testing, distinguishing between (left-side) testing activities performed in the field (aka in-vivo) and (right-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 a Software Under Test (SUT) and a set of test cases that are executed against the SUT within the development environment. 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. 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 ex-vivo testing techniques extensively use field data. Figure 2 represents the case of ex-vivo testing as the only kind of in-house testing activity relevant to this study.

Definition 5 (Field Testing (in-vivo)). *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, by opportunistically copying components and processes, thus limiting the degree of intrusiveness of field tests on the operational software system. Obtaining distinct instances of the software under test 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.

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.

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

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. We identify three dimensions: *test selection* to identify 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*	Section 5.1.1 *Mutation of Field Executions* [24, 62, 66, 67] *Test Suite Adaptation* [32, 41, 42]	Section 5.1.2 *Profile Data* [29]		Section 5.1.3 *Field Triggers* [60]
offline	*functional*	Section 5.2.1 *IO Data Pattern* [69] *Metamorphic Relations* [9, 64]			Section 5.2.3 *Built-in Tests* [46, 47, 63, 78, 79, 83] *Test planning and management* [36, 37, 65] *Adaptation and Reconfiguration* [52, 54]
	non-functional	Section 5.2.2 *Security Specifications* [22, 23]			
online	*functional*	Section 5.3.1 *Choreographies and Service-based specifications* [2, 5, 6, 11, 20, 82] *Finite-State Models* [17, 18, 28, 59, 80] *Metamorphic Relations* [19] *Graph grammars* [71]	Section 5.3.2 *Event Interface* [85]	Section 5.3.3 *Fault Injection* [86]	Section 5.3.4 *Test planning and management* [1, 4, 13, 36, 37, 49, 50] *Adaptation and Reconfiguration* [27, 38, 45, 52, 54] *Isolation* [16]
	non-functional	Section 5.4.1 *Stochastic Models of User Behavior* [70, 74] *Security Specifications* [10–12, 40] *Usability Models* [57] *Δ-grammars for QoS* [71] Timed-automata [59, 80]		Section 5.4.2 *Fault Injection* [3, 86] *Operational Profile* [61]	Section 5.4.3 *Adaptation and Reconfiguration* [27, 56, 58]

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

Table 2. Platforms for field testing

Offline Testing Platforms	Online Testing Platforms
Section 5.5	Section 5.6
Evolution	*Built-In and Pre-Existing Test Cases*
[48, 51, 53, 55]	[25, 26, 35, 44, 69, 84]
Shadow Instances for V&V	*SOA Online Testing*
[34, 39]	[68]
SOA and Component-Based Testing	*Evolution*
[8, 15, 69, 87–89]	[21, 31, 48, 51, 53]

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 [22, 23]. Online testing approaches address both functional and non-functional requirements.

From the *test case generation* viewpoint, we cluster techniques in four categories [72] depending on the information used to generate the test cases: *(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. Most available approaches either generate test cases from specifications, to suitably cover certain behaviors, or define strategies to execute test cases that are already available. Only few approaches derive test cases from structural information and fault-models.

This section thoroughly discusses field testing techniques, by referring to Table 1, as indicated by the references in the table itself.

5.1 Ex-Vivo Functional Testing

Ex-vivo testing techniques exploit information from the field to improve the in-house testing process. Research on ex-vivo testing focuses on functional test cases generated from specifications, code structure or fault-models.

5.1.1 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.

Mutation of field executions has been investigated by both Neves et al. [24, 66, 67] in the context of autonomous vehicles software and Morán et al. [62] in the context of MapReduce distributed data processing applications. Neves et al. [24, 66] 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 [62] 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* [76].

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. [32] 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 functions* 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. [41, 42] 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.

5.1.2 Structure-Based Approaches. Structure-based approaches generate in-house test cases by exploitging structural targets derived from the observation of field executions. Elbaum and Hardojo [29] 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.

5.1.3 Pre-existing Test Cases. Some approaches generate new test cases by modifying the available ones with information extracted from the field. Mei et al. [60] 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 reassessment of test priorities that may preempt an ongoing regression testing session, and opportunistically *trigger* a nested session aimed at covering the changed objectives.

5.2 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. The few offline testing techniques proposed so far either implement specification-based approaches or reuse existing test suites.

5.2.1 Specification-Based Approaches for Functional Requirements. Some approaches use specifications to generate functional test cases that can be executed offline. Nishijima et al. [69] 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.

Murphy et al. [9, 64] 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.

5.2.2 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 [22, 23]. 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.

5.2.3 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 Both Wang *et al.* [83] and Murphy *et al.* [63] enhance object-oriented classes with built-in test cases that are executed in test-mode. Suliman et al. [79] propose built-in tests to assess the confidence and the reliability of software components when running in the field, while Piel *et al.* define built-in tests to validate the component integration during the system evolution.

King et al. [46, 47] and Stevens et al. [78] 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. [65] define a strategy to identify untested application states at run-time, to prevent executing redundant test cases. Gonzalez-Sanchez *et al.* [36, 37] 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.* [52, 54] propose a technique to validate *architectural reconfigurations* with test cases that they select with a dependency analysis of the components involved in the reconfiguration, and that they execute online before completing the reconfiguration.

5.3 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.

5.3.1 Specification-Based Approaches. Specification-based approaches exploit different kinds of specifications to generate field test cases to be executed in the production environment. Different 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 to 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 [5, 6, 82]. 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.* [20] propose an approach to dynamically test the reconfiguration of composite web services exploiting a stochastic *usage model* of the services.

Bertolino *et al.* [11] 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 exploit *finite state models* to generate online test cases. Dranidis *et al.* derive test cases from Stream X-Machines (SXMs), a kind of finites tate machines that model both control- and the data-flow of a software system [28]. 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 [59]. They define an algorithm for generating and executing test cases implemented in the WSCCT WS-BPEL Compositions Conformance Testing tool. Similarly, Cao *et al.* [18] convert BPEL specifications into timed extended finite state machines that they use to derive test cases. 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 [80]. 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 [17]. 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. Metamorphic testing approaches generate new test inputs and oracles from observed executions, by using metamorphic relations. Chan et al. [19] 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.* [71] generate online test cases from Δ-grammars specifications of the functional services.

5.3.2 Structure-Based Approaches. Structure-based approaches exploit structural information to derive test cases to be executed online. Chunyang et al. [85] propose a method for white-box testing of service compositions with minimal exposure of internal information about the implementations of 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 define the informations that services shall expose depending on the desired coverage information, branch, path, data-flow coverage, and generates test cases with random and constraint-based techniques.

5.3.3 Fault-Based Approaches. Fault-based approaches generate test cases from fault models. Zhang's approach [86] 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.

5.3.4 Pre-existing Test Cases. Several field-testing techniques focus on execution and maintenance of available test cases.

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. [49, 50] 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.* [4, 13] 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.* [36, 37] define metrics to estimate the software testability, as we already discussed for offline techniques.

Adaptation and Reconfiguration. Field testing techniques are used to validate reconfigurations and adaptations either before or immediately after changes are finalized. Gu *et al.* [38] use online testing to validate the recovery actions synthesized for handling Java exceptions in a sandboxed environment before exploiting them. King *et al.* [45] and Piel *et al.* [73] test adaptations and evolution among interconnected components running availabel test cases. Lahami *et al.* propose a technique to validate architectural reconfigurations through online testing [52, 54]. The approach leverages dependency analysis of the affected components to identify the test cases to be executed, while

delaying the requests that the component under test receives until the testing process is completed. Di Penta *et al.* [27] 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. Bobba *et al.* [16] investigate the use of transactional memory to isolate the testing process from the system in operation.

5.4 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.

5.4.1 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.* [74] 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 a.l* combine test cases from stochastic models with passive monitoring to improve QoS attributes [70].

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 [10], and general role-based access policies [12]. De Angelis *et al.* generate test cases from UML-based specifications. Bertolino *et al.*'s PLASTIC framework [11] addresses non-functional properties of networked services, with test cases derived from Symbolic Transition System (STS) specifications.

Hui *et al.* [40] 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 military domain using a controller that can run test sessions online, engaging the users [57].

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 [71].

Timed Automata. Both Vain *et al.* [80] and Maaley *et al.* [59] generate online test cases from timed automata specifications to sample both the functional and timing behavior of the SUT.

5.4.2 Fault-Based Approaches. Fault-based approaches exploit fault-models to derive test suites.

Fault Injection. Fault-injection approaches aim to assess the capability of software systems to react to unexpected and faulty situations. Zhang's approach [86] that we already discuss in Section 5.3.3 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 *Post-monkey* approach alter the communication between components by both introducing random delays of message delivery and randomly injecting invalid messages.

Operational Profile. Metzger *et al.* [61] propose an online pro-active testing approach for service-based applications that aims to predict failures with a konw confidence. The approach monitors the applications for failures, and generates test cases with uses search-based strategies.

5.4.3 Pre-existing Test Cases. Some online testing approaches support the adaptation and reconfiguration of the software. Di Penta *et al.* [27] rely on available test cases to assess the non-regression of

the non-functional characteristics of web services. Lee and Na [56] 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. [58] 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 of the service under test under specific load conditions.

5.5 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 [55] 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 runtime testing of distributed (real-time) systems [48, 51]. Lahami *et al.*'s framework manages test execution in response to reconfiguration events. The framework identifies the components involved in the reconfiguration with dependency analysis, and executes the available TTCN-3 [48] test cases. The framework executes the test cases according to a resource-aware test plan generated at runtime [53].

Shadow Instances for V&V. Some platforms maintain shadow instances of the operative software, for verification and validation activities. In particular, Hosek and Cadar [39] 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. [34] 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, which operators substitute with 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 [88, 89] 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 [87] 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 is used that leverages coverage reports from previous testing sessions to infer changes on services externals, without information about their internals.

Bhanu *et al.* [15] propose a general architecture for offline testing of embedded software in the context of safety critical system. Nishijima *et al* [69] 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.

5.6 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.* [44, 69, 84] propose an architecture to support online testing with test cases both built in the autonomous component and generated with dynamically activated dedicated modules. Deussen *et al.* [25, 26] 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.* [35] 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.* [68] 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.* [21] propose *JAAF+T*, an extension of the *Java self-Adaptive Agent Framework* (JAAF), a framework for self-adaptive agents construction. 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 [48, 51], as already discussed in Section 5.5.

Fredericks *et al.* [31] present Proteus, a framework for online testing of self-adaptive systems that addresses the evolution of online test suites, by adjusting test parameters, making sure that test suites remain relevant to changes at runtime.

5.7 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 approach 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.

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

Table 3. Place *where* tests are generated

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 lead to the execution of some field test cases.

Most studies of field-testing strategies propose proactive strategies: 63 papers, that is 88% 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; 4 papers, 5%, support both strategies. For instance, Kawano *et al.*'s approach [44] both proactively activates test cases when modules change, and reactively responds to failures.

Figure 3 classifies the strategies according to the triggers they react to. The taxonomy identifies two main kinds of triggers, *IT Operations* 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

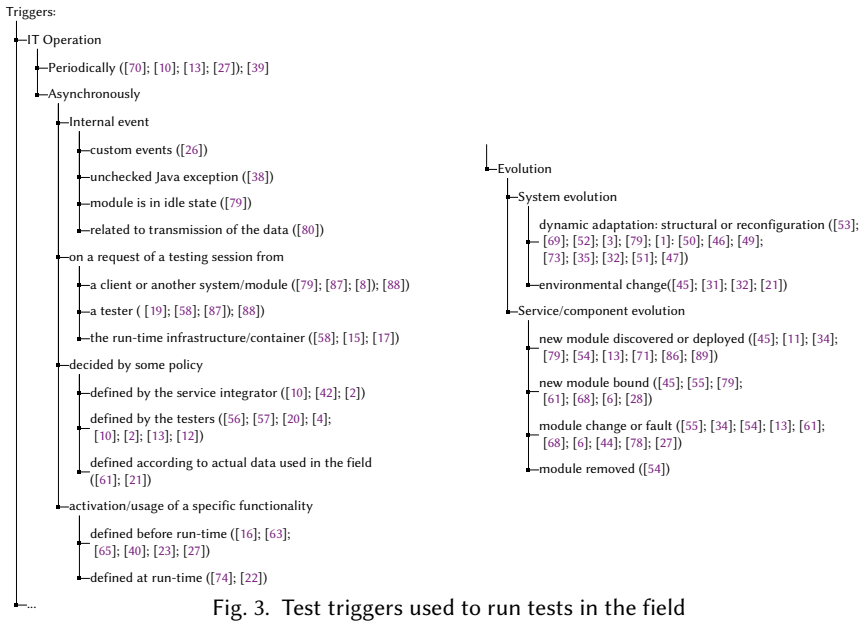


Fig. 3. Test triggers used to run tests in the field

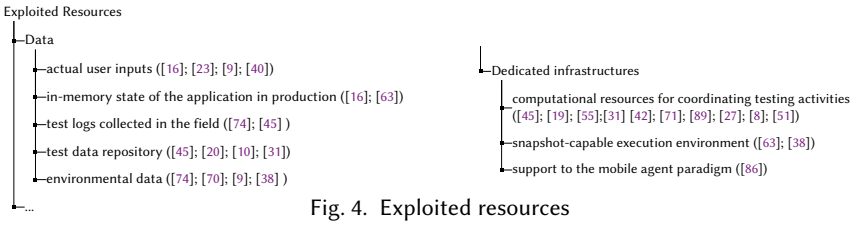


Fig. 4. Exploited resources

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.

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.

Figure 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.* [16] exploit *user inputs* to detect failures, Dai *et al.* [23] exploit *user inputs* to detect security vulnerabilities introduced by changes in the configurations. Bell *et al.* [9] mutate user inputs and observed outputs, leveraging metamorphic testing or weak-mutation strategies, to produce new test cases, Hui *et al.* [40] mutate inputs and outputs to reveal security vulnerabilities.

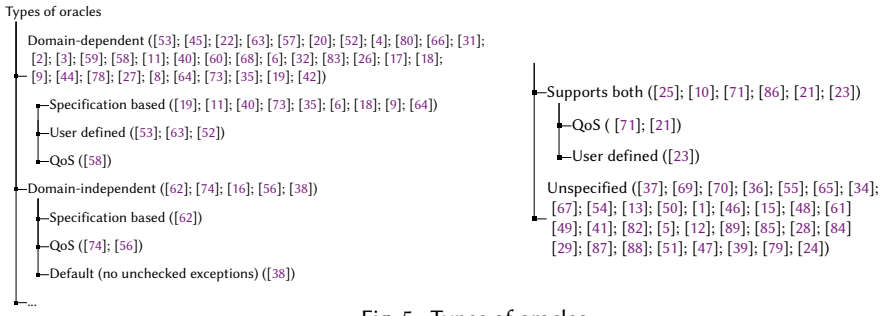


Fig. 5. Types of oracles

Other approaches rely on the *in-memory state* of the application in production. Bobba *et al.* [16] and Murphy *et al.* [63] 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.* [74] use such information to predict adaptations, while King *et al.* [45] 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 [20], authentication/authorization cookies [10], or more generally mutable test artefacts so as to keep them consistent with the current operating conditions [31]. Some approaches take advantage of information on both the *environment* and the execution context. For instance, Gu *et al.* [38] 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 [38, 63] or *agent-based paradigm* to migrate components between hosts [86].

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 result 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. Figure 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 (33 papers ≈ 41%) 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 [11], SUT models or specifications [6], BPEL specifications [18]. Several approaches refer to metamorphic relations [9, 19, 40, 62, 64]. Some approaches rely on user-defined oracles [23, 52, 53].

Some domain-independent oracles rely on QoS attributes of the SUT, such as general performance metrics, like response time [21, 58, 74], or execution duration and availability [71]. Few studies exploit the default oracle, that is, the detection of crashes or unchecked exceptions [38].

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-testing 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.

Target	Number of approaches	%
New feature	23	34%
Regression	36	54%
Changed feature	27	40%
Changed environment	18	27%

Table 4. Targets of testing approaches

Most approaches deal with regression testing (36 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 [46], and component-based systems [53]. 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 [60].

Field-testing has been also significantly exploited to test the impact of environment evolution. In fact, it is hard to exercise in-house every possible environment and every 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.

Granularity	Number of papers	%
Unit	45	58%
Subsystem	29	37%
System	24	31%
System of Systems	2	3%

Table 5. Granularity of testing approaches

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

Category	Number of papers	%
Remote	59	75%
Embedded	10	13%
Desktop	10	13%
Mobile	1	1%

Table 6. Approaches per application type

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.

Field testing approaches that target embedded applications consider interactions with hardware and environment, and focus primarily on ex-vivo testing [24, 66, 67] that can be safely performed in-house. Only few approaches deal with online testing for systems with strong fault tolerance and assurance requirements [3, 15, 34, 44, 80, 84].

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.

- *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

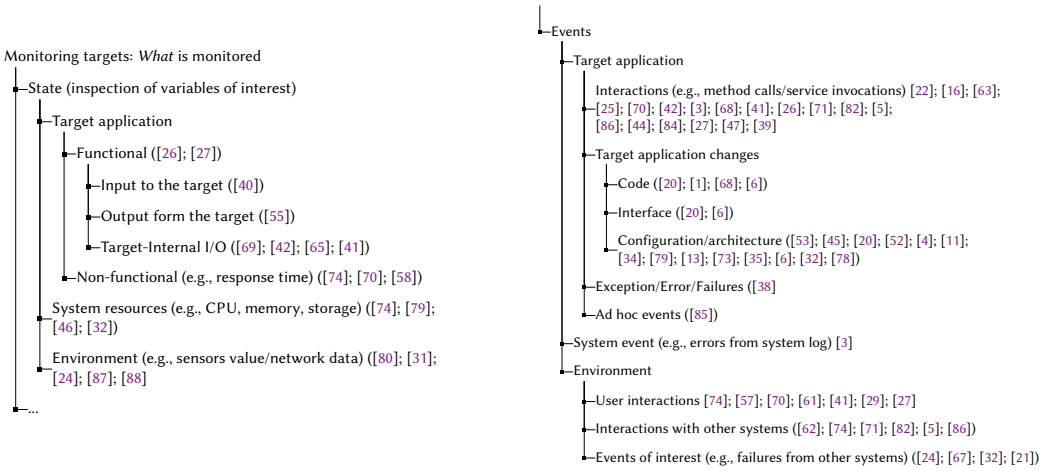


Fig. 6. Field testing approaches by monitoring targets

events and states of both the SUT and environment, data that are needed to trigger the testing process, generate and select test cases and selecting 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.

Figure 6 groups field-testing approached 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 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 79 (70%) 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 [40] 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

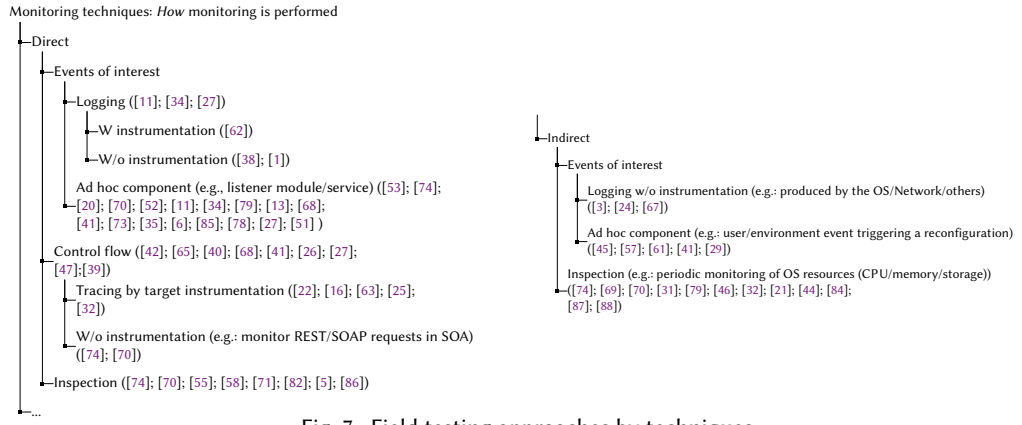


Fig. 7. Field testing approaches by techniques

untrusted values. Lee *et al.*'s approach [55] 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 with the output of the component that is replaced 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.*'approach [41] 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 [65] 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 [58].

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 [46]. De Olivera Neves *et al.*'s approach monitors the environment with sensors to trigger environment-dependent field test cases [24].

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 [38]) or domain-specific *ad hoc* events (e.g., events specifically defined for coverage assessment and published in the event interface [85]). Approaches can also exploit *system events*, such as data present in the logs, to assess the expected vs observed behavior during testing [3]. Some approaches monitor interactions of the target application with the external *environment*, by capturing *interactions* with either *users* or *other applications/services* such as in the case of service compositions.

Figure 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 79 (67%) of the studies explicitly indicate the monitoring technique, that is, *how* monitoring is performed. We distinguish between direct and indirect monitoring. Direct monitoring looks at events *directly* produced in the target application, indirect monitoring looks at information produced in the system or environment.

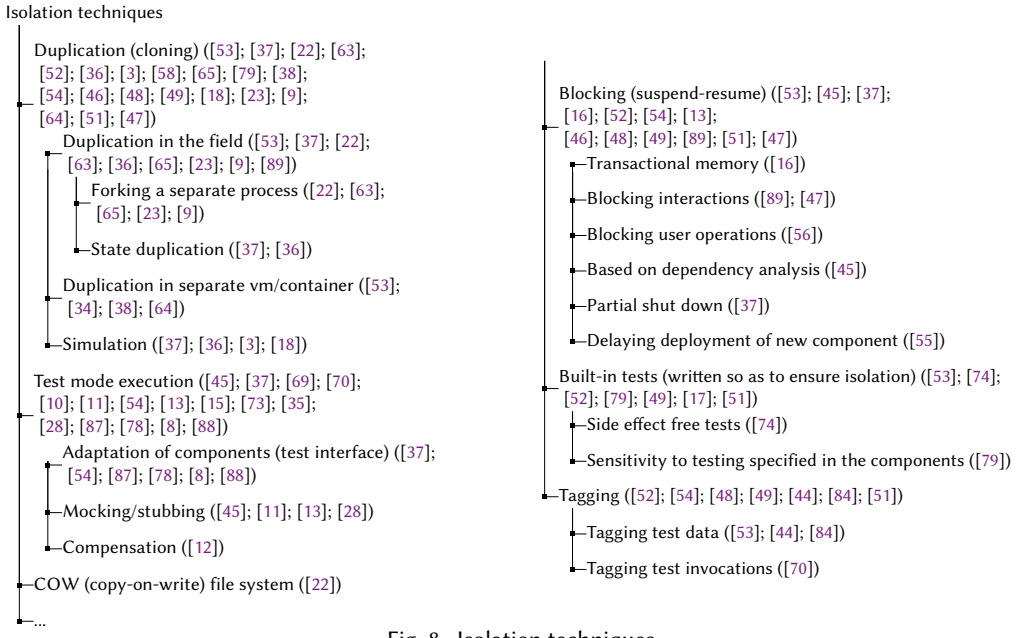


Fig. 8. Isolation techniques

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 component 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 [20].

Some approaches monitor the system by tracing the *control flow* of the execution, either with or without (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, 24]. Other approaches propose *ad hoc components* for monitoring users' or environment's events that trigger reconfiguration [45], [57]. Yet other approaches rely on periodic inspection of attributes of the system and the environment, such as system resource consumption [46] and network data [44], [84].

8.2 Isolation

Field testing shall not interfere with normal operations with undesired side effects. Field testing approaches prevent side effects on the execution flow of the SUT, by either executing field tests in sand-boxed environments or rolling-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 that we discuss 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 in the field with the expected one without interfering [25].

Many approaches that target service-oriented applications either assume side-effect-free SUTs or suitable *compensation mechanisms* [12], 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 that allow test developers 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 test cases are with no side effects by design [74]. 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.

Figure 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 the in-vivo testing from the production processes, by either executing the field test cases in a *separated virtual machine* that duplicates the production 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) with *stubs and mocks*, or (iii) by 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 [22] 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 test cases by design (*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 components by design, 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, that is, *sensitivity to field testing is specified for each component*. It is 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*.

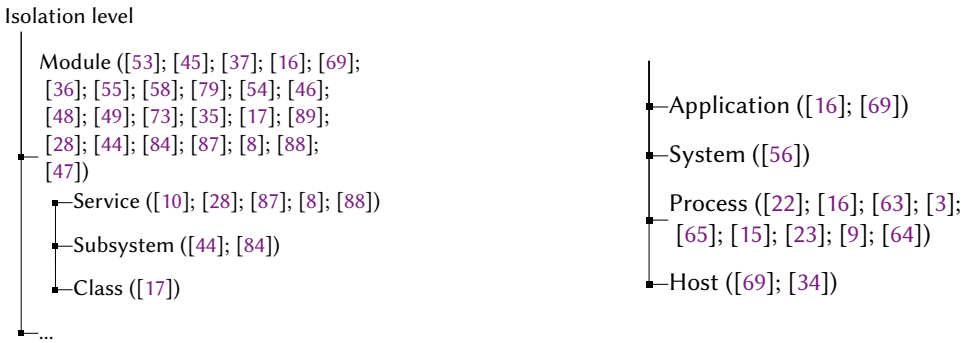


Fig. 9. Granularity of the computational unit being isolated during field testing

Few approaches define and explicit *isolation polices*. Lahami *et al.* [53] propose multiple policies, and assume that *developers choose* among the alternatives (a subset of those in Figure 8) once for all, before deploying the test suite.

Gonzalez-Sanchez *et al.* [37] 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.* [48] 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 Figure 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 at low granularity levels.

8.3 Privacy and Security

Privacy and security issues are still largely unaddressed. Only 8 studies out of 79 (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.* [85] propose a method for white-box testing 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.* [87] 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 ontology of software testing STOWS [88].

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.* [27] 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 [86] 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.* [20] exploit secure communication to assure the privacy of test execution logs and reports. Bertolino *et al.* [12] propose a method to test service

compositions under the governance of a service federation. The online testing component triggers service requests that are indistinguishable from regular requests, thanks to an assertion signed by an identity provider, which grants a regular role to the tester component. Tests are selected and executed proactively, so as to ensure the trustworthiness of the federation.

Finally, both Hummer *et al.* [42] and Murphy *et al.* [63] 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 the importance of the issues: 30% of the studies do not explicitly indicate what they monitor, and 33% 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 amount of studies monitor several information sources (9 out of the 53 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 form possible side effects on the state of the execution and the environment, and from the interference with non functional properties, and 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 propsoed 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 the importance of the issue, especially in some applicative domains, like web and service-based applications. Only one of the studies reported in this survey indicates security and privacy as a main focus [10].

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. Figure 10 groups field testing approaches according to the proposed test selection strategies.

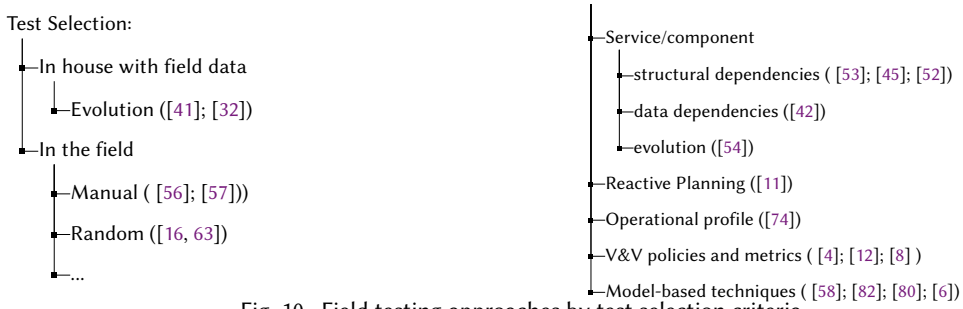


Fig. 10. Field testing approaches by test selection criteria

Few approaches select test cases in-house with data collected from the field. Frederiks *et al.* and Hummer *et al.* select ex-vivo test cases when observing changes of the system or its environment observed in the field [32, 41].

Lee *et al.* and Loustarinen *et al.* [56, 57] rely on *manual* test case selection based on domain knowledge, which is accurate, but slow and error-prone. Many approaches propose automatic procedures. Bobba *et al.* and Murphy *et al.* [16, 63] rely on *random* selection, simple and fast, but with no guarantee of accuracy and effectiveness of the selected test cases. Some approaches drive test selection with *information about the service or component* to be tested: King *et al.* and Lahamani *et al.* select test cases based on the structural dependencies to be tested [52, 53], Hummer *et al.* according to data dependencies [42], Lahami *et al.* according to information about the evolution of the unit under test[54].

Bertolino *at 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 [11]. Their auditing approach selects test cases at run-time guided by both the behavioural specification of the SUT (i.e., STS [30]) and the output returned by the SUT at each interaction. Some approaches select test cases based on the *operational profile*. De Angelis *at al.* refer to modules that have been exercised less by users [74].

Other approaches select test cases according to policies and metrics relevant to the *V&V software process*. De Angelis *et al.* [4, 12] consider test selection 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 that 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: Vain *et al.* rely on Markov models [80], Wang *et al.* on Petri Nets [82] to capture the behavior of the SUT and select the test cases that are likely to reveal failures. Bai *et al.* and Ma *et al.* use reinforcement learning and the 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 [6, 58].

Only few approaches address test prioritization, that is, a strategy for scheduling the order of execution of field test cases. Mei *et al.* [60] 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 likely to better exercise the change. Bobba *et al.* [16] select field test cases with a random strategy, and Vein *et al.* [80] with a probabilistic strategy. Few other approaches barely refer to some prioritization strategies without providing details [27, 61, 74].

9.2 Field Testing Governance

In general, governance is the act of 'governing' or 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 refer 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 [14].

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 system under test 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.

Test orchestration refers to the strategy to combine methods and tools into a coherent and seamless process that can be successfully integrated with normal production activities.

Figure 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.* [45] 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.* [58] 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.* [20] 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.* and Bertolino *et al.* [4, 13] 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. The TT4RT framework for runtime testing by Lahami *et al.* [48, 49] 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 [11] 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 [88, 89] (originally outlined in a preliminary work [87]) is orchestrated around the provision of 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 [31] 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.

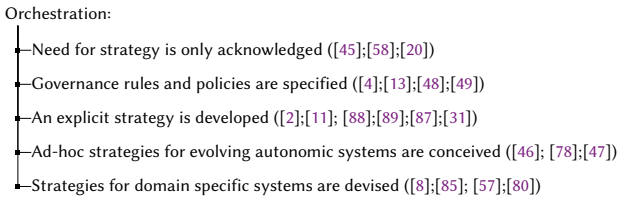


Fig. 11. Field testing approaches by orchestration strategy

King *et al.* [46, 78] and Stevens *et al.* [47] 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 [85] 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.* [57] use field testing for remote usability testing, by requiring to natively integrate a dedicated API within the user interface. Vain *et al.* [80] propose testing in operation for mission-critical systems, with an orchestration strategy based on a model-based conformance testing approach.

User awareness refers to the degree to which end users of the SUT are involved within field testing execution, and the possible cooperation that is required on their side.

Only few approaches explicitly address if and how the users are explicitly aware and involved into field testing. Niebuhr *et al.*'s approach [68] specifies the test cases executed at runtime by the Service Users. Luostarinen *et al.* [57] foresee that users actively participate to field testing by driving the requested usability tests.

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.* [52], 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.* [64] 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: Only a small set of studies investigate test case prioritization. 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 where 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 79 studies discuss orchestration, and among those only 6 develop an explicit strategy. Only 5 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 MAIN FINDINGS AND CONCLUSIONS

Field testing techniques address the complexity, unpredictability, evolvability and size of modern software systems, challenges that in-house testing activities cannot manage satisfactory. This paper surveys the state of the art in field testing following four research questions that face a multitude of aspects, including test generation, execution and evolution. Table 7 summarizes the main findings for each research questions. Our survey identifies several open 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. More advanced scenarios, such as the opportunistic generation of test cases in the field based on the characteristics of the underlying production environment, are also possible. 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 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.

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 mean 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

Table 7. Summary of the findings

ACKNOWLEDGMENTS

This paper describes research work undertaken in the context of the Italian MIUR PRIN 2015 Project: GAUSS.

REFERENCES

[1] M. Akour, A. Jaidev, and T. M. King. 2011. Towards Change Propagating Test Models in Autonomic and Adaptive Systems. In *Proceedings of the International Conference on the Engineering of Computer-Based Systems (ECBS)*.

[2] M. Ali, F. De Angelis, D. Fanì, A. Bertolino, G. De Angelis, and A. Polini. 2014. An Extensible Framework for Online Testing of Choreographed Services. *Computer* 47, 2 (Feb. 2014), 23–29. https://doi.org/10.1109/MC.2013.407

[3] K. Alnawasreh, P. Pelliccione, Z. Hao, M. Ränge, and A. Bertolino. 2017. Online Robustness Testing of Distributed Embedded Systems: An Industrial Approach. In *IEEE/ACM 39th International Conference on Software Engineering: SEIP*

Track. 133–142. https://doi.org/10.1109/ICSE-SEIP.2017.17

[4] Guglielmo De Angelis, Antonia Bertolino, and Andrea Polini. 2012. Validation and Verification Policies for Governance of Service Choreographies. In *Proceedings of the 8th International Conference on Web Information Systems and Technologies (WEBIST)*.

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

[6] Xiaoying Bai, Dezheng Xu, Guilan Dai, Wei-Tek Tsai, and Yinong Chen. 2007. Dynamic reconfigurable testing of service-oriented architecture. In *31st Annual International Computer Software and Applications Conference*, Vol. 1.

[7] Luciano Baresi and Carlo Ghezzi. 2010. The disappearing boundary between development-time and run-time. In *Proceedings of the Workshop on Future of Software Engineering Research (FoSER)*.

[8] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. 2011. Bringing white-box testing to Service Oriented Architectures through a Service Oriented Approach. *Journal of Systems and Software* 84, 4 (2011), 655–668. https://doi.org/10.1016/j.jss.2010.10.024 The Ninth International Conference on Quality Software.

[9] Jonathan Bell, Christian Murphy, and Gail Kaiser. 2015. Metamorphic Runtime Checking of Applications Without Test Oracles. *CrossTalk, The Journal of Defense Software Engineering* 28, 2 (2015), 9–13.

[10] Antonia Bertolino, Guglielmo De Angelis, and Andrea Polini. 2011. (role)CAST: A Framework for On-line Service Testing. In *Proceedings of the 7th International Conference on Web Information Systems and Technologies (WEBIST)*.

[11] Antonia Bertolino, Guglielmo De Angelis, Lars Frantzen, and Andrea Polini. 2007. The PLASTIC framework and tools for testing service-oriented applications. In *Software Engineering, International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*. Springer, 106–139. https://doi.org/10.1007/978-3-540-95888-8_5

[12] Antonia Bertolino, Guglielmo De Angelis, Sampo Kellomaki, and Andrea Polini. 2012. Enhancing Service Federation Trustworthiness through Online Testing. *IEEE Computer* 45, 1 (2012), 66–72.

[13] Antonia Bertolino, Guglielmo De Angelis, and Andrea Polini. 2013. Governance Policies for Verification and Validation of Service Choreographies. In *Web Information Systems and Technologies (Selected Papers) (LNBIP)*, J. Cordeiro and K.H. Krempels (Eds.), Vol. 140. Springer, 86–102.

[14] Antonia Bertolino and Andrea Polini. 2009. SOA Test Governance: Enabling Service Integration Testing Across Organization and Technology Borders. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '09)*. IEEE Computer Society, Washington, DC, USA, 277–286. https://doi.org/10.1109/ICSTW.2009.39

[15] Smt. J. Sasi Bhanu, A Vinaya Babur, and P. Trimurthy. 2015. A Comprehensive Architecture for Dynamic Evolution of Online Testing of an Embedded System. *Journal of Theoretical and Applied Information Technology* 80, 1 (2015), 160–172. http://www.jatit.org/volumes/Vol80No1/17Vol80No1.pdf

[16] Jayaram Bobba, Weiwei Xiong, Luke Yen, Mark D Hill, and David A Wood. 2009. StealthTest: Low overhead online software testing using transactional memory. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 146–155.

[17] Daniel Brenner, Colin Atkinson, Barbara Paech, Rainer Malaka, Matthias Merdes, and Dima Suliman. 2006. Reducing Verification Effort in Component-Based Software Engineering through Built-In Testing. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*. 175–184. https://doi.org/10.1109/EDOC.2006.44

[18] Tien-Dung Cao, Patrick Félix, Richard Castanet, and Ismail Berrada. 2010. Online Testing Framework for Web Services. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST)*.

[19] W. K. Chan, S. C. Cheung, and Karl R. P. H. Leung. 2007. A Metamorphic Testing Approach for Online Testing of Service-Oriented Software Applications. *International Journal of Web Services Research* 4 (2007), 61–81. https://doi.org/10.4018/jwsr.2007040103

[20] M. B. Cooray, J. H. Hamlyn-Harris, and R. G. Merkel. 2015. Dynamic Test Reconfigurationfor Composite Web Services. *IEEE Transactions on Services Computing* 8, 4 (July 2015), 576–585. https://doi.org/10.1109/TSC.2014.2312953

[21] Andrew Diniz da Costa, Camila Nunes, Viviane Torres da Silva, Baldoino Fonseca, and Carlos JP de Lucena. 2010. JAAF+T: a framework to implement self-adaptive agents that apply self-test. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 928–935.

[22] Huning Dai, Christian Murphy, and Gail Kaiser. 2010. Configuration fuzzing for software vulnerability detection. In *2010 International Conference on Availability, Reliability and Security*. IEEE, 525–530.

[23] Huning Dai, Christian Murphy, and Gail E. Kaiser. 2010. CONFU: Configuration Fuzzing Testing Framework for Software Vulnerability Detection. *International Journal of System of Systems Engineering* 1, 3 (2010), 41–55.

[24] Vânia de Oliveira Neves, Márcio Eduardo Delamaro, and Paulo Cesar Masiero. 2014. An environment to support structural testing of autonomous vehicles. In *2014 Brazilian Symposium on Computing Systems Engineering*. IEEE, 19–24. https://doi.org/10.1109/SBESC.2014.27

[25] Peter H. Deussen, George Din, and Ina Schieferdecker. 2002. An On-Line Test Platform for Component-Based Systems. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW '02)*.

[26] P. H. Deussen, G. Din, and I. Schieferdecker. 2003. A TTCN-3 based online test and validation platform for Internet services. In *The Sixth International Symposium on Autonomous Decentralized Systems, 2003. ISADS 2003.* 177–184. https://doi.org/10.1109/ISADS.2003.1193946

[27] Massimiliano Di Penta, Marcello Bruno, Gianpiero Esposito, Valentina Mazza, and Gerardo Canfora. 2007. *Web Services Regression Testing.* Springer Berlin Heidelberg, Berlin, Heidelberg, 205–234. https://doi.org/10.1007/978-3-540-72912-9_8

[28] Dimitris Dranidis, Andreas Metzger, and Dimitrios Kourtesis. 2010. Enabling Proactive Adaptation through Just-in-Time Testing of Conversational Services. In *Proceedings of the Third European Conference ServiceWave.* 63–75.

[29] Sebastian Elbaum and Madeline Hardojo. 2004. An Empirical Study of Profiling Strategies for Released Software and Their Impact on Testing Activities. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04).* ACM, New York, NY, USA, 65–75. https://doi.org/10.1145/1007512.1007522

[30] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. 2004. Test Generation Based on Symbolic Specifications. In *Proc. of 4th International Workshop Formal Approaches to Software Testing (FATES), Revised Selected Papers (LNCS)*, Vol. 3395. Springer, 1–15. https://doi.org/10.1007/978-3-540-31848-4_1

[31] Erik M. Fredericks and Betty H. C. Cheng. 2015. 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 '15).* IEEE Press, Piscataway, NJ, USA, 157–168. https://doi.org/10.1109/SEAMS.2015.15

[32] Erik M. Fredericks, Byron DeVries, and Betty H. C. Cheng. 2014. 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).* ACM, New York, NY, USA, 17–26. https://doi.org/10.1145/2593929.2593937

[33] Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. 2017. An Exploratory Study of Field Failures. In *International Symposium on Software Reliability Engineering (ISSRE).* https://doi.org/10.1109/ISSRE.2017.10

[34] Okehee Goh and Yann-Hang Lee. 2007. Schedulable online testing framework for real-time embedded applications in VM. In *International Conference on Embedded and Ubiquitous Computing (EUC 2007).* Springer, 730–741. https://doi.org/10.1007/978-3-540-77092-3_63

[35] Alberto González, Eric Piel, and Hans-Gerhard Gross. 2008. Architecture support for runtime integration and verification of component-based systems of systems. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering.* IEEE Press, I–41.

[36] A. Gonzalez-Sanchez, E. Piel, and H. Gross. 2009. RiTMO: A Method for Runtime Testability Measurement and Optimisation. In *Proceedings of the 9th International Conference on Quality Software.* 377–382. https://doi.org/10.1109/QSIC.2009.56

[37] A. Gonzalez-Sanchez, E. Piel, H. G. Gross, and A. J. C. van Gemund. 2010. Runtime Testability in Dynamic High-Availability Component-Based Systems. In *2010 Second International Conference on Advances in System Testing and Validation Lifecycle.* 37–42. https://doi.org/10.1109/VALID.2010.13

[38] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. 2016. Automatic runtime recovery via error handler synthesis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16).* 684–695. https://doi.org/10.1145/2970276.2970360

[39] Petr Hosek and Cristian Cadar. 2015. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15).* 339–353. https://doi.org/10.1145/2694344.2694390

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

[41] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. 2011. Test Coverage of Data-Centric Dynamic Compositions in Service-Based Systems. In *Proc. of the 4th IEEE International Conference on Software Testing, Verification and Validation, (ICST).* IEEE Computer Society, Berlin, Germany, 40–49. https://doi.org/10.1109/ICST.2011.55

[42] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. 2013. Testing of data-centric and event-based dynamic service compositions. *Software Testing, Verification and Reliability* 23, 6 (2013), 465–497. https://doi.org/10.1002/stvr.1493

[43] M. Y. Ivory and M. A. Hearst. 2001. The state of the art in automating usability evaluation of user interfaces. *Comput. Surveys* 4, 33 (2001).

[44] K. Kawano, M. Orimo, and K. Mori. 1989. Autonomous decentralized system test technique. In *Proceedings of the 13th Annual International Computer Software Applications Conference.* 52–57. https://doi.org/10.1109/CMPSAC.1989.65053

[45] Tariq M. King, Andrew A. Allen, Rodolfo Cruz, and Peter J. Clarke. 2011. Safe Runtime Validation of Behavioral Adaptations in Autonomic Software. In *Autonomic and Trusted Computing (Lecture Notes in Computer Science).* Springer, Berlin, Heidelberg, 31–46. https://doi.org/10.1007/978-3-642-23496-5_3

[46] Tariq M. King, Djuradj Babich, Jonatan Alava, Peter J. Clarke, and Ronald Stevens. 2007. Towards Self-Testing in Autonomic Computing Systems. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*.

[47] Tariq M. King, Alain E. Ramirez, Rodolfo Cruz, and Peter J. Clarke. 2007. An Integrated Self-Testing Framework for Autonomic Computing Systems. *Journal of Computers* 2, 9 (2007), 37–49. https://doi.org/10.4304/jcp.2.9.37-49

[48] Mariam Lahami, Fairouz Fakhfakh, Moez Krichen, and Mohamed Jmaiel. 2012. Towards a TTCN-3 Test System for Runtime Testing of Adaptable and Distributed Systems. In *Proc. of Testing Software and Systems - 24th IFIP WG 6.1 International Conference, (ICTSS) (LNCS)*, Vol. 7641. Springer, Aalborg, Denmark, 71–86. https://doi.org/10.1007/978-3-642-34691-0_7

[49] Mariam Lahami and Moez Krichen. 2013. Test Isolation Policy for Safe Runtime Validation of Evolvable Software Systems. In *Proc. of the Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE Computer Society, Hammamet, Tunisia, 377–382. https://doi.org/10.1109/WETICE.2013.62

[50] Mariam Lahami, Moez Krichen, Mariam Bouchakwa, and Mohamed Jmaiel. 2012. Using Knapsack Problem Model to Design a Resource Aware Test Architecture for Adaptable and Distributed Systems. In *Proceedings of the 24th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS '12)*. 103–118. https://doi.org/10.1007/978-3-642-34691-0_9

[51] Mariam Lahami, Moez Krichen, and Mohamed Jmaiel. 2012. A Distributed Test Architecture for Adaptable and Distributed Real-Time Systems. *Journal of New technologies of Information (RNTI), CAL 2012* RNTI-L-6 (2012), 73–92. https://editions-rnti.fr/?inprocid=1001804

[52] Mariam Lahami, Moez Krichen, and Mohamed Jmaiel. 2013. Runtime Testing Framework for Improving Quality in Dynamic Service-based Systems. In *Proceedings of the International Workshop on Quality Assurance for Service-based Applications (QASBA 2013)*. ACM, New York, NY, USA, 17–24. https://doi.org/10.1145/2489300.2489335

[53] Mariam Lahami, Moez Krichen, and Mohamed Jmaïel. 2015. Runtime Testing Approach of Structural Adaptations for Dynamic and Distributed Systems. *International Journal of Computer Applications in Technology (IJCAT)* 51, 4 (July 2015), 259–272. https://doi.org/10.1504/IJCAT.2015.070489

[54] Mariam Lahami, Moez Krichen, and Mohamed Jmaiel. 2016. Safe and efficient runtime testing framework applied in dynamic and distributed systems. *Science of Computer Programming* 122 (2016), 1–28. https://doi.org/10.1016/j.scico.2016.02.002

[55] Kihwal Lee and Lui Sha. 2005. 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)*. 160–165. https://doi.org/10.1109/RTCSA.2005.8

[56] Seoung-Hyeon Lee and Jung-Chan Na. 2017. Development of Test Agents for Automated Dynamic Testing of UNIWAY. In *Advanced Multimedia and Ubiquitous Engineering*, James J. (Jong Hyuk) Park, Shu-Ching Chen, and Kim-Kwang Raymond Choo (Eds.). Springer Singapore, Singapore, 683–688.

[57] R. Luostarinen, R. Järvinen, J. Määttä, and J. Manner. 2016. A model for usability testing in challenging environments. In *Proceedings of the International Conference on Military Communications and Information Systems (ICMCIS)*. 1–6. https://doi.org/10.1109/ICMCIS.2016.7496557

[58] Bo Ma, Bin Chen, Xiaoying Bai, and Junfei Huang. 2010. Design of bdi agent for adaptive performance testing of web services. In *2010 10th International Conference on Quality Software*. IEEE, 435–440.

[59] A. J. Maâlej, M. Krichen, and M. Jmaïel. 2012. Model-Based Conformance Testing of WS-BPEL Compositions. In *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. 452–457. https://doi.org/10.1109/COMPSACW.2012.86

[60] L. Mei, W. K. Chan, T. H. Tse, B. Jiang, and K. Zhai. 2015. Preemptive Regression Testing of Workflow-Based Web Services. *IEEE Transactions on Services Computing* 8, 5 (Sep. 2015), 740–754. https://doi.org/10.1109/TSC.2014.2322621

[61] Andreas Metzger, Osama Sammodi, Klaus Pohl, and Mark Rzepka. 2010. Towards pro-active adaptation with confidence: augmenting service monitoring with online testing. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ACM, Cape Town, South Africa, 20–28. https://doi.org/10.1145/1808984.1808987

[62] J. Morán, A. Bertolino, C. de la Riva, and J. Tuya. 2017. Towards Ex Vivo Testing of MapReduce Applications. In *International Conference on Software Quality, Reliability and Security (QRS)*. https://doi.org/10.1109/QRS.2017.17

[63] Christian Murphy, Gail E. Kaiser, Ian Vo, and Matt Chu. 2009. Quality Assurance of Software Applications Using the In Vivo Testing Approach. In *International Conference on Software Testing Verification and Validation (ICST '09)*.

[64] Christian Murphy, Kuang Shen, and Gail Kaiser. 2009. Automatic System Testing of Programs Without Test Oracles. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/1572272.1572295

[65] Christian Murphy, Moses Vaughan, Waseem Ilahi, and Gail Kaiser. 2010. Automatic detection of previously-unseen application states for deployment environment testing and analysis. In *Proceedings of the 5th Workshop on Automation of Software Test*. ACM, 16–23. https://doi.org/10.1145/1808266.1808269

[66] Vânia de Oliveira Neves, Márcio Eduardo Delamaro, and Paulo Cesar Masiero. 2016. Combination and mutation strategies to support test data generation in the context of autonomous vehicles. *International Journal of Embedded Systems* 8, 5/6 (2016), 464–482. https://doi.org/10.1504/IJES.2016.080388
[67] Vânia de Oliveira Neves, Márcio Eduardo Delamaro, and Paulo Cesar Masiero. 2017. Automated structural software testing of autonomous vehicles. *CibSE Proceedings-SET-Software Engineering Track* (2017).
[68] Dirk Niebuhr, Andreas Rausch, Cornel Klein, Juergen Reichmann, and Reiner N. Schmid. 2009. Achieving Dependable Component Bindings in Dynamic Adaptive Systems - A Runtime Testing Approach. In *Proc. of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems, (SASO)*. https://doi.org/10.1109/SASO.2009.40
[69] Eiji Nishijima, Hiroshi Yamamoto, Katsumi Kawano, Kazunori Fujiwara, and Keiji Oshima. 1996. On-line Testing for Application Software of Widely Distributed System. In *Proceedings of the Symposium on Reliable Distributed Systems*.
[70] Marc Oriol, Xavier Franch, and Jordi Marco. 2015. Monitoring the service-based system lifecycle with SALMon. *Expert Systems with Applications* 42, 19 (2015), 6507–6521. https://doi.org/10.1016/j.eswa.2015.03.027
[71] Youngki Park, Woosung Jung, Byungjeong Lee, and Chisu Wu. 2009. Automatic discovery of web services based on dynamic black-box testing. In *Annual International Computer Software and Applications Conference*, Vol. 1. IEEE.
[72] M. Pezzè and M. Young. 2008. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons Inc.
[73] Éric Piel and Alberto Gonzalez-Sanchez. 2009. Data-flow integration testing adapted to runtime evolution in component-based systems. In *Proceedings of the ESEC/FSE workshop on Software integration and evolution@ runtime*.
[74] O. Sammodi, A. Metzger, X. Franch, M. Oriol, J. Marco, and K. Pohl. 2011. Usage-Based Online Testing for Proactive Adaptation of Service-Based Applications. In *2011 IEEE 35th Annual Computer Software and Applications Conference*. 582–587. https://doi.org/10.1109/COMPSAC.2011.81
[75] G. Schermann, J. Cito, and P. Leitner. 2018. Continuous Experimentation: Challenges, Implementation Techniques, and Current Research. *IEEE Software* 2, 35 (2018), 26–31.
[76] S. Segura, G. Fraser, A. B. Sanchez, and A. R. Cortes. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering (TSE)* 42, 9 (2016).
[77] C. S. Smidts, C. Mutha, M. Rodriguez, and M. J. Gerber. 2014. Software testing with an operational profile: OP definition. *Comput. Surveys* 46, 3 (2014).
[78] Ronald Stevens, Brittany Parsons, and Tariq M. King. 2007. A Self-testing Autonomic Container. In *Proceedings of the Annual Southeast Regional Conference (ACM-SE 45)*. https://doi.org/10.1145/1233341.1233343
[79] Dima Suliman, Barbara Paech, Lars Borner, Colin Atkinson, Daniel Brenner, Matthias Merdes, and Rainer Malaka. 2006. The MORABIT approach to runtime component testing. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Vol. 2. IEEE, 171–176. https://doi.org/10.1109/COMPSAC.2006.169
[80] Jüri Vain, Leonidas Tsiopoulos, Vyacheslav Kharchenko, Apneet Kaur, Maksim Jenihhin, and Jaan Raik. 2017. 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 (WS TheRMIT 2017)*.
[81] A. Vermeeren, E.-L.-C. Law, V. Roto, M. Obrist, J. Hoonhout, and K. Mattila. 2010. User Experience Evaluation Methods: Current State and Development Needs. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries*. 521–530.
[82] Yongbo Wang, Xiaoying Bai, Juanzi Li, and Ruobo Huang. 2007. Ontology-based test case generation for testing web services. In *Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*. IEEE, 43–50.
[83] Yingxu Wang, Graham King, Dilip Patel, Shushma Patel, and Alec Dorling. 1999. On coping with real-time software dynamic inconsistency by built-in tests. *Annals of Software Engineering* 7, 1 (01 Oct 1999), 283–296. https://doi.org/10.1023/A:1018990322378
[84] Hiroshi Yamamoto, Akira Yoshizawa, Katsumi Kawano, Kinji Mori, Keiji Oshima, and Tsunao Kawamura. 1993. On-line Software Test Techniques Based on Autonomous Decentralized System. In *Proccedings of the 4th Workshop on Future Trends of Distributed Computing Systems (FTDCS)*. 291–296. https://doi.org/10.1109/FTDCS.1993.344143
[85] Chunyang Ye and Hans-Arno Jacobsen. 2013. Whitening SOA Testing via Event Exposure. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1444–1465.
[86] Jia Zhang. 2004. An approach to facilitate reliability testing of web services components. In *15th International Symposium on Software Reliability Engineering*. IEEE, 210–218.
[87] H. Zhu. 2006. A Framework for Service-Oriented Testing of Web Services. In *Proceedings of the Annual International Computer Software and Applications Conference*, Vol. 2. https://doi.org/10.1109/COMPSAC.2006.95
[88] Hong Zhu and Yufeng Zhang. 2012. Collaborative Testing of Web Services. *IEEE Transactions on Services Computing* 5 (2012), 116–130.
[89] Hong Zhu and Yufeng Zhang. 2014. A Test Automation Framework for Collaborative Testing of Web Service Dynamic Compositions. In *Advanced Web Services*. 171–197.

Received XXX 201?; revised XXX 201?; accepted XXX 201?