Emerging Software Testing Technologies

Francesca Lonetti[1] and Eda Marchetti[1]

[1]ISTI-CNR, 56124 Pisa, Italy
{francesca.lonetti, eda.marchetti}@isti.cnr.it

Abstract

Software testing encompasses a variety of activities along the software development process and may consume a large part of the effort required for producing software. It represents a key aspect to assess the adequate functional and non functional software behaviour aiming to prevent and remedy malfunctions. The increasing complexity and heterogeneity of software poses many challenges to the development of testing strategies and tools. In this chapter, we provide a comprehensive overview of emerging software testing technologies. Beyond the basic concepts of software testing, we address prominent test case generation approaches and focus on more relevant challenges of testing activity as well as its role in recent development processes. An emphasis is also given to testing solutions tailored to the specific needs of emerging application domains.

Keywords— Software testing, Testing objectives, Testing techniques and tools, Testing challenges, Software process, Testing needs and trends, Emerging application domains

1 Introduction

The testing phase is an important and critical part of software development, consuming even more than half of the effort required for producing deliverable software [6]. Unfortunately, often due to time or cost constraints, the testing is not developed in the proper manner or is even skipped. The testing activity in fact is not limited to the detection of "bugs" in the software, but it encompasses the entire development process.

The testing planning starts during the early stages of requirement analysis, and proceeds systematically, with continuous refinements during the course of software development until the completion of the coding phase, with the beginning of the test cases execution. This last step represents the biggest part of software cost that can be evaluated in terms of: the cost of designing a suitable set of test cases which can reveal the presence of bugs;

the cost of running those tests, which also requires a considerable amount of time; the cost of detecting them, i.e. the development of a proper "oracle" which can identify the manifestation of bugs as soon as possible; the cost of correcting them. All these activities have in common the same testing purpose: evaluating the product quality for increasing the software engineering confidence in the proper functioning of the software. However, it must be made clear that testing cannot show the absence of defects; it can only reveal that software defects are present (Dijkstra [40]).

In this chapter, we refer to the definition of the Software Testing introduced in [22]: "*Software Testing consists of the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behaviour*". As shown by this definition, testing deals with dynamic verification of system quality, which also involves the code execution, as will be better described in this Chapter. Generally, the techniques applicable for quality evaluation can be divided into two sets: static techniques, which do not involve code execution, and dynamic techniques, to which testing belongs to, which instead require running code. The static techniques are applicable all during the process development for different purposes such as to check the adherence of the code to the specification or to detect defects in code by its inspection or review. Instead, the latter approach more properly observes failures as they show up. In particular, dynamic analysis techniques involve the execution of the code and the analysis of its responses in order to determine its validity and detect errors. The behavioral properties of the program are also observed. Other examples of dynamic analysis include simulation, sizing and timing analysis, and prototyping, which may be applied throughout the lifecycle [6]. Here, we briefly present the static techniques (Section 2.1.1), preferring to concentrate on testing, which is the main topic of this Chapter.

Before continuing the presentation it is important to clarify the terminology relative to the terms "fault", "defect" and "failure" that we will use. Although their meanings are strictly related, there are some distinctions between them. As discussed in [6], a failure is the manifested inability of the program to perform the function required, i.e. a system malfunction evidenced by incorrect output, abnormal termination or unmet time and space constraints. The cause of a failure, i.e. the missing or incorrect code, is a fault. In particular, a fault may remain undetected until some stirring up event activates it. In this case it brings the program into an intermediate unstable state, called error, which if propagated to the output causes a failure. The process of failure manifestation is therefore *Fault-Error-Failure*, which can be iterated recursively: a fault can be caused by a failure in some other interacting system.

Testing reveals failures and a consequent analysis stage is needed to identify the faults that caused them. In particular, it is possible that many different failures can result from a single fault, and the same failure can be

caused by different faults.

The chapter is organized as follows: In Section 2 we present basic concepts of testing including types of test (static and dynamic), test levels and objectives characterizing the testing activity; in Section 3 we address most prominent test generation techniques and tools; Section 4 targets challenging aspects of software testing whereas Section 5 identifies the role of software testing in relevant software processes. Finally, Section 6 outlines needs and trends of software testing for emerging application domains and Section 7 draws discussion and conclusions.

2 Basic Concepts of Testing

The one term testing involves different concepts, refers to a full range of test techniques, even quite different from one other, and embraces a variety of aims. In this section we provide some basic concepts of software testing useful in the remaining of this Chapter.

2.1 Type of Tests

Since the 1980, the widespread use of modern technologies has led a large part of the software engineering to focus its attention on quality, usability, safety and other characteristic attributes of software applications. In particular, interest was captured both by the process for software development and by its results. Using their experience software engineering researchers have gradually arrived at the conviction that only the joint between a mature and well-established development process with specific techniques for the quantitative evaluation of the attributes of interest of the artifacts produced can guarantee high quality and reliable applications. Therefore, research has been split into two sets, with of course some natural intersections and points of contact: the former interested to the process (Software Process Improvement (SPI) [106], and the latter focused on the product.

Frameworks such as CMM [88], SPICE [43], RUP [66] (detailed in Section 5) are the products of the SPI research work belonging to the former set. They capture the good practices for the process assessment and are de facto references used by thousands of organizations. Considering the latter set, generally the techniques applicable to the product can be divided into two groups: static techniques, which do not involve code execution, and the dynamic techniques, which instead require code running, to which testing belongs. In the remaining of this section more details about these two techniques are provided.

2.1.1 Static Techniques

Static techniques are based on the (manual or automated) examination of project documentation, of software models and code, and of other related information about requirements and design. Thus, static techniques can be employed all along software development, and their earlier usage is of course highly desirable.

Considering a generic development process (see Section 5 for more details), they can be applied [6]:

- at the requirements stage for checking language syntax, consistency and completeness as well as the adherence to established conventions;
- at the design phase for evaluating the implementation of requirements, and detecting inconsistencies (for instance between the inputs and outputs used by high level modules and those adopted by sub-modules);
- during the implementation phase for checking that the form adopted for the implemented products (e.g., code and related documentation) adheres to the established standards or conventions, and that interfaces and data types are correct.

Traditional static techniques include [27]: *Software inspection*, that is the step-by-step analysis of the documents (deliverables) produced, against a compiled checklist of common and historical defects; *Software reviews*, that is the process by which different aspects of the work product are presented to project personnel (managers, users, customer etc) and other interested stakeholders for comment or approval; *Code reading*, that is the desktop analysis of the produced code for discovering typing errors that do not violate style or syntax; *Algorithm analysis and tracing*, that is the process in which the complexity of algorithms employed and the worst-case, average-case and probabilistic analysis evaluations can be derived.

The processes implied by the above techniques are heavily manual, error-prone, and time consuming. To overcome these problems, researchers have proposed static analysis techniques relying on the use of formal methods [54]. The goal is to automate as much as possible the verification of the properties of the requirements and the design. Towards this goal, it is necessary to enforce a rigorous and unambiguous formal language for specifying the requirements and the software architecture.

In the middle between static and dynamic analysis techniques, is symbolic execution [34], which executes a program by replacing variables with symbolic values (see Section 3.4).

2.1.2 Dynamic Techniques

Dynamic techniques [22] obtain information of interest about a program by observing some executions. Standard dynamic analysis include testing

(on which we focus in the rest of the chapter) and profiling. Essentially, a program profile records the number of times some entities of interest occur during a set of controlled executions. These data can be used to derive measures of coverage or frequency. Other specific dynamic techniques also include simulation, sizing and timing analysis, and prototyping [6].

Testing properly said is based on the execution of the code on valued inputs. Of course, although the set of input values can be considered infinite, those that can be run effectively during testing are finite. It is in practice impossible, due to the limitations of the available budget and time, to exhaustively exercise every input of a specific set even when not infinite. In other words, by testing we observe some samples of the program behavior. A test strategy therefore must be adopted to find a trade-off between the number of chosen inputs and overall time and effort dedicated to testing purposes. Different techniques can be applied depending on the target and the effect that should be reached. We will describe test selection strategies in Section 3.

2.2 Objectives of testing

Software testing can be applied for different purposes, such as verifying that the functional specifications are implemented correctly, or that the system shows specific non-functional properties such as performance, reliability, usability. A (certainly non complete) list of relevant testing objectives includes [22, 6]:

- Acceptance/qualification testing: the final test action prior to deploying a software product. Its main goal is to verify that the software respects the customer requirement. Generally, it is run by or with the end-users to perform those functions and tasks the software was built for;

- Installation testing: the system is verified upon installation in the target environment. Installation testing can be viewed as system testing conducted once again according to hardware configuration requirements. Installation procedures may also be verified;

- Alpha testing: before releasing the system, it is deployed to some inhouse users for exploring the functions and business tasks. Generally, there is no test plan to follow, but the individual tester determines what to do;

- Beta Testing: the same as alpha testing but the system is deployed to external users. In this case the amount of detail, the data, and approach taken are entirely up to the individual testers. Each tester is responsible for creating their own environment, selecting their data, and determining what functions, features, or tasks to explore. Each

tester is also responsible for identifying their own criteria for whether to accept the system in its current state or not;

- Conformance Testing/Functional Testing: the test cases are aimed at validating that the observed behavior conforms to the specifications. In particular, it checks whether the implemented functions are as intended and provide the required services and methods. This test can be implemented and executed against different test targets, including units, integrated units, and systems;

- Non-Functional testing: it is specifically aimed at verifying non-functional properties of the system such as performance, reliability, usability, security and so on;

- Regression testing: it is the selective re-execution of test cases to verify that code modifications have not caused unintended effects and that the system or component still complies with requirements. In practice, the objective is to show that a system which previously passed the tests still does. Notice that a trade-off must be made between the assurance given by regression testing every time a change is made and the resources required to do that.

2.3 Test Levels

During the development lifecycle of a software product, testing is performed at different levels and can involve the whole system or parts of it. Depending on the process model adopted, then, software testing activities can be articulated in different phases, each one addressing specific needs relative to different portions of a system. Whichever the process adopted, we can at least distinguish in principle between unit, integration, system and regression test [6]. These are the testing stages of a traditional phased process (see Section 5). However, even considering different, more modern, process models, a distinction between these test levels remains useful to emphasize three logically different moments in the verification of a complex software system. None of these levels is more relevant than another, and more importantly a stage cannot supply for another, because each addresses different typologies of failures.

2.3.1 Unit Test

A unit is the smallest testable piece of software, which may consist of hundreds or even just a few lines of source code, and generally represents the result of the work of one programmer. The unit test purpose is to ensure that the unit satisfies its functional specification and/or that its implemented structure matches the intended design structure [6].

Unit tests can also be applied to check interfaces (parameters passed in correct order, number of parameters equal to number of arguments, parameter and argument matching), local data structure (improper typing, incorrect variable name, inconsistent data type) or boundary conditions.

2.3.2 Integration Test

Generally speaking, integration is the process by which software pieces or components are aggregated to create a larger component. Integration testing is specifically aimed at exposing the problems that can arise at this stage. Even though the single units are individually acceptable when tested in isolation, in fact, they could still result in incorrect or inconsistent behaviour when combined in order to build complex systems. For example, there could be an improper call or return sequence between two or more components [6].

Integration testing thus is aimed at verifying that each component interacts according to its specifications as defined during preliminary design. In particular, it mainly focuses on the communication interfaces among integrated components.

There are not many formalized approaches to integration testing in the literature, and practical methodologies rely essentially on good design sense and the testers intuition. Integration testing of traditional systems was done substantially in either a non-incremental or an incremental approach.

In a non-incremental approach the components are linked together and tested all at once ("big-bang" testing) [6]. In the incremental approach, we find the classical "top-down" strategy, in which the modules are integrated one at a time, from the main program down to the subordinated ones, or "bottom-up", in which the tests are constructed starting from the modules at the lowest hierarchical level and then are progressively linked together upwards, to construct the whole system. Usually in practice, a mixed approach is applied, as determined by external project factors (e.g., availability of modules, release policy, availability of testers and so on) [6].

In modern Object Oriented, distributed systems, approaches such as top-down or bottom-up integration and their practical derivatives, are no longer usable, as no "classical" hierarchy between components can be generally identified. Some other criteria for integration testing imply integrating the software components based on identified functional threads [6]. In this case, the test is focused on those classes used in reply to a particular input or system event (thread-based testing); or by testing together those classes that contribute to a particular use of the system. Finally, some authors have used the dependency structure between classes as a reference structure for guiding integration testing, i.e., their static dependencies, or even the dynamic relations of inheritance and polymorphism [6].

2.3.3 System Test

System test involves the whole system embedded in its actual hardware environment and is mainly aimed at verifying that the system behaves according to the user requirements. In particular, it attempts to reveal bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects (which are the subject of integration testing).

Summarizing, the primary goals of system testing can be [6]: i) discovering the failures that manifest themselves only at system level and hence were not detected during unit or integration testing; ii) increasing the confidence that the developed product correctly implements the required capabilities; iii) collecting information useful for deciding the release of the product.

System testing should therefore ensure that each system function works as expected, failures are exposed and analyzed, and additionally that interfaces for export and import routines behave as required.

Generally, system testing includes testing for performance, security, reliability, stress testing and recovery [6]. In particular, test and data collected applying system testing can be used for defining an operational profile necessary to support a statistical analysis of system reliability (see Section 3.3).

A further test level, called *Acceptance Test*, is often added to the above subdivision. This is however more an extension of system test, rather than a new level. It is in fact a test session conducted over the whole system, which mainly focuses on the usability requirements more than on the compliance of the implementation against some specification. The intent is hence to verify that the effort required from end-users to learn to use and fully exploit the system functionalities is acceptable.

2.3.4 Regression Test

Properly speaking, regression test is not a separate level of testing (it is listed among test objectives in Section 2.2), but may refer to the retesting of a unit, a combination of components or a whole system after modification, in order to ascertain that the change has not introduced new faults [6].

As software produced today is constantly in evolution, driven by market forces and technology advances, regression testing takes by far the predominant portion of testing effort in industry. Since both corrective and evolutive modifications may be performed quite often, to re-run after each change all previously executed test cases would be prohibitively expensive [52]. Therefore various types of techniques have been developed to reduce regression testing costs and to make it more effective.

Selective regression test techniques [6] help in selecting a (minimized) subset of the existing test cases by examining the modifications (for instance at code level, using control flow and data flow analysis). Other approaches

instead prioritize the test cases according to some specified criterion (for instance maximizing the fault detection power or the structural coverage), so that the test cases judged the most effective with regard to the adopted criterion can be taken first, up to the available budget (see Section 4.4.3 for more details).

3 Test cases Generation

Test cases generation is among the most important and intensive activities of software testing. A lot of research has been devoted in the last decades to automatic test case generation with the consequent development of different techniques and tools. In this section, we provide an overview of prominent test case generation approaches evidencing the main issues and challenges of each of them.

3.1 Search-based Testing

Search-Based Software Testing is one of the emerging methodologies for automated test generation. It is based on an heuristic, such as a Genetic Algorithm [115], to optimize search techniques and automate the test case definition. Such heuristic uses a problem-specific fitness function for on side guiding the search of solutions from a potentially infinite search space, and from the other limiting the required execution time. Search-Based Software Testing is not a recent proposal, indeed the first approach dates back to 1976 [82]. However, in the last years there is an huge increase of proposals due to their flexibility and applicability. Commonly, the application of search-based testing requires that [78]: the solutions for the problem should be encoded so that they can be manipulated by the heuristic algorithm; the fitness function needs to be defined for each specific problem and should be able to guide the search to promising areas of the search space by evaluating candidate solutions.

Due to their versatility, search-based approaches have been adopted in several areas such as [78]: functional testing, temporal testing, integration testing, regression testing, stress testing, mutation testing, test prioritization, interaction testing, state machine testing and exception testing.

Common weaknesses of search-based testing are related to the interaction with external environment, the definition of the fitness function and the automated oracle. In the former case, the main difficulty is related to the handling interactions with external environment or components with which the system under test could be dependent. In particular, issues could be risen in checking the existence of files or directories. In literature, possible solutions propose to include, in the file or database, test data that can be read back by the program under test, or to use mock objects.

In case of the fitness functions, due to their heuristics nature, there is the possibility they fail to give adequate guidance to the search. A common example is the so-called *flag* problem in structural test data generation [55]: a branch predicate consists of a boolean value (*the flag*), yielding only two branch distance values one for when the flag is true, and one for when it is false. The *Testability Transformation* [55] is the most adopted solution to deal with this problem. It consists in the generation of a temporary version of the program under test that can be used to generate the test data and then discarded.

About the automated oracle, notwithstanding the huge interest devoted to this topic, reducing the human activity in the evaluation of the testing results is still an issue. Indeed, available proposals for fitness functions are more focused on maximizing the (structural) coverage, while minimizing the testing effort. The seeding work of [79] tries to alleviate this problem proposing solutions based on knowledge management, such for instance the possibility of explicitly select the starting point of any search-based approach or use the program source of information for setting up the types of inputs that may be expected.

3.2 Model-based Testing

In recent years, model-based testing (MBT) has become increasingly successful thanks to the emergence of model-centric development paradigms, such as UML and MDA [67]. Models are used for capturing knowledge and specify a system with different levels of accuracy. The main goal of model-based testing is the automatic generation, execution and evaluation of test cases based on a formal model of the SUT. The work in [111] provides a taxonomy of characteristics, similarities and differences of MBT techniques, and classifies existing tools according to them.

The four main approaches known as model-based testing are [110]: i) Generation of test input data from a domain model; ii) Generation of test input data from an environment model; iii) Generation of test cases with oracle from a behavior model; iv) Generation of executable test scripts from abstract tests.

However, model-based testing suffers from some drawbacks that may prevent its use in some application areas, such as the availability of a system specification, which not always exists in practice. Moreover, when it exists, this specification should be complete enough to ensure some relevance of the derived test suite. Finally, this specification cannot include all the implementation details and is restricted to a given abstraction level. Therefore, to become executable, the derived test cases need to be refined into more concrete interaction sequences. Automating this process still remains an open problem.

Despite the continuous evolving of MBT field, as it could be observed in

the increasing number of MBT techniques published at the technical literature, there is still a gap between research related to MBT and its application in the software industry. The authors of [39] present an overview of MBT approaches supporting the selection of MBT techniques for software projects and the risk factors that may influence the use of these techniques in the industry.

3.3 Black-box vs. White-box Testing

In this section, alternative classifications of test techniques are provided: the first, called black-box testing, is based on the input/output behaviour of the system (Section 3.3.1); the second, called white-box testing, is based on the structure and internal data of the software under test (Section 3.3.2); the third called grey-box testing trying to mix the previous one (Section 3.3.3). Generally, the presented test techniques are not alternative approaches but can be used in combination as they use and provide different sources of information [63].

3.3.1 Black-box Testing

Black-box testing, also called functional testing, relies on the input/output behaviour of the system. In particular, the system is subjected to external inputs, so that the corresponding outputs are used to verify the conformance of the system to the specified behaviour, with no assumptions of what happens in between. Therefore, in this process we assume knowledge of the (formal or informal) specification of the system under test, which can be used to define a behavioral model of the system (a transaction flowgraph) [6]. A complete black-box test would consist of subjecting the program to all possible input streams and verifying the outcome produced, but as stated in Section 2 this is theoretically impossible. For this, different techniques can be applied such as:

- Testing from formal specifications: In this case it is required that specifications be stated in a formal language, with a precise syntax and semantics. The tests are hence derived automatically from the specification, which are also used for deriving inductive proofs for checking the correct outcome [54].

- Equivalence partitioning: The functional tests are derived from the specifications written in structured, semiformal language. The input domain is partitioned into equivalence classes so that elements in the same class behave similarly. In this context, the Category Partition is a well-known and quite intuitive method, which provides a systematic, formalized approach to partition testing [29].

- Boundary-values analysis: This is a complementary approach to equivalence partitioning, and concentrates on the errors occurring at boundaries of the input domain. The test cases are thus chosen near the extremes of the class [29].

- Random methods: they consists of generating random test cases based on a uniform distribution over the input domain. It is a low-cost technique because large sets of test patterns can be generated cheaply without requiring any preliminary analysis of software [84, 8].

- Operational profile: Test cases are produced by a random process meant to produce different test cases with the same probabilities with which they would arise in actual use of the software [73].

- Decision Tables: The decision tables are rules expressed in a structural way used to express the test experts' or design experts' knowledge. Decision Tables can be used when the outcome or the logic involved in the program is based on a set of decisions and rules which need to be followed [84].

- Cause-effect graphs: These are combinatorial logic networks that can be used to explore in systematic way the possible combinations of input conditions. By analyzing the specification, the relevant input conditions or causes, and the consequent transformations and output conditions, the effects are identified and modeled into graphs linking the effects to their causes [84].

- Combinatorial Testing: In combinatorial testing, test cases are designed to execute combinations of input parameters [84]. Because providing all combinations is usually not feasible in practice, due to their extremely large numbers, combinatorial approaches able to generate smaller test suites for which all combinations of the features are guaranteed, are preferred [8]. Among them, common approach is all-pair testing technique, focusing on all possible discrete combinations of each pair of input parameters. We refer to [85] for a complete overview of the most recent proposals and tools.

- State Transition Testing: This type of testing is useful for testing state machine and also for navigation of graphical user interface [84]. A wide variety of state transition systems exist, including finite state machines [68], I/O Automata [72], Labeled Transition Systems [109], UML state machines [30], Simulink/Stateflow [108], and PROMELA [12].

- Evidence based testing: In evidence-based software engineering (EBSE) the *best* solution for a practical problem should be identified based on evidence [62]. The process for solving practical problems based on a

rigorous research approach includes the following different steps that can be applied also to testing activities [64]: i) identify the evidence and formulate a question; ii) track down the best evidence to answer the question; iii) critically reflect on the evidence provided with respect to the problem and context that the evidence should help to solve. In software engineering, two approaches that allow to identify and aggregate evidence are systematic mapping studies and systematic reviews [100].

One of the points against the black-box testing is its dependence on the specification's correctness and the necessity of using a large amount of inputs in order to get good confidence of acceptable behaviour.

3.3.2 White-box Testing

The white-box testing, also called structural testing, requires complete access to the object's structure and internal data, which means the visibility of the source code. The tests are derived from the program's structure, which is also used to track which parts of the code have been executed during testing. For this, some of the commonly used techniques for test case selection are:

- Control flow-based criteria: these techniques use the control flow graph representation of a program in which nodes correspond to sequentially executed statements while edges represent the flow of control between statements. The aim of white box testing criteria is to cover as much as possible the control flow graph, limiting the number of selected test cases. In particular, they differentiate in: *statement coverage* which is based on executable statements, *branch coverage*, which focuses on the blocks and case statements that affect the control flow, *condition coverage* which relies on subexpressions independently of each other, *path coverage* which is based on the possible paths exercised through the code [84, 6].

- Data-Flow coverage: in data-flow testing, a data definition of a variable is a location where a value is stored in memory (definition) and a data use is a location where the value of the variable is accessed for computations (c-use) or for predicate uses (p-use). The data-flow testing goal is to generate tests that execute program subpaths from definition to use. Traditional data-flow analysis techniques work on control flow graphs annotated with specific information on data usage [84, 6].

3.3.3 Grey-box Testing

The grey-box testing tries to combine the two above mentioned proposals [63]. In this case, the tests are generated exploiting limited knowledge of

the internal working and the knowledge of input/output behaviour of the system under test. Important aspects of grey box testing techniques are the possibility of exploiting the interface definition and functional specification rather than source code and the focus on the user's point of view rather than designer one. On the other side, usually grey-box testing does not assure the coverage assessment as the access to source code is completely not available. Indeed, there is the possibility that many program paths remain untested or that the tests could be redundant. The other name of grey box testing is translucent testing.

3.4 Symbolic Execution

Symbolic execution is a popular program analysis technique, introduced in the '70s, that has found interest in recent years in the research community, with particular emphasis on applications to test generation [34]. The main idea behind symbolic execution is to use symbolic values, instead of actual data, as input values, and to execute the program by manipulating program expressions involving the symbolic values. The state of a symbolically executed program is represented by the symbolic values of program variables and by a boolean formula over the symbolic inputs representing constraints which the inputs must satisfy. As a result, the output values computed by a program are expressed as a function of the input symbolic values and a symbolic execution tree represents the execution paths. The tree nodes represent program states and they are connected by program transitions [87].

The main challenges of symbolic execution in processing real-world code are related to [13]:

- Environment interactions: interactions with the file system or the network through libraries and system calls could cause side-effects and affect the execution. Evaluating any possible interaction outcome is generally unfeasible since it could generate a large number of execution states. A typical strategy is to consider popular library and system routines and create models that can help the symbolic engine to analyze only significant outcomes.

- State space explosion and path selection: the existence of real-world looping programs might exponentially increase the number of execution states and prevent the symbolic execution engine to exhaustively explore all the possible states within a reasonable amount of time. In practice, heuristics are used to guide and prioritize states exploration. In addition, efficient mechanisms for evaluating multiple states in parallel are implemented in symbolic engines.

- Constraint solver limitations: Constraint solvers suffer from a number of limitations when they have to deal with non-linear constraints over

their elements. Symbolic execution engines normally rely on optimizations and algebraic simplifications.

However, to overcome the limitations of symbolic execution due to path explosion and the time spent in constraint solving, a standard approach consists into combining concrete and symbolic executions of the code under test. This approach is known as *concolic* testing [101] and tightly couples both concrete and symbolic executions that run simultaneously, and each gets feedback from the other. The basic idea is that a *concolic* execution engine uses the concrete execution to drive the symbolic execution, then after choosing an arbitrary input to begin with, it executes the program both concretely and symbolically by simultaneously updating concrete and symbolic stores as well as the path constraints.

Symbolic execution has been included in several recent tools offering the capability to systematically exploring many possible execution paths at the same time without necessarily requiring concrete inputs. The most common ones evidencing the growing impact of symbolic execution in practice are:

- Symbolic Java Path Finder (Symbolic JPF): It is a symbolic execution framework that implements a non-standard bytecode interpreter on top of the Java PathFinder model checking tool [86]. It allows to perform automated generation of test cases and to check properties of code during test case generation. The framework performs a non-standard bytecode interpretation and uses JPF to systematically generate and execute the symbolic execution tree of the code under analysis. It has been used for testing a prototype NASA flight software component helping discovering a serious bug.

- CUTE and jCUTE: CUTE (A Concolic Unit Testing Engine) and jCUTE (CUTE for Java) handle multi-threaded programs that manipulate dynamic data structures using pointer operations [102]. They combine concolic execution with dynamic partial order reduction to systematically generate both test inputs and thread schedules. Both tools have been applied to test several Java and C open-source software.

- KLEE: It is a symbolic execution tool, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs [33]. An important feature of KLEE is its ability to handle interactions with real world data read from the file system or over the network. Moreover, it is applied to a variety of areas, including wireless sensor networks, automated debugging, testing of binary device drivers, online gaming.

3.5 Non-functional Testing

An important aspect for software testing is the possibility of determine and evaluate properties of the product such as determine how fast the product responds to a request or how long it takes to do an action. Non-Functional testing is performed at all test levels and is focused on software attributes such performance, security, scalability and so on. Nowadays, there exist more than 150 types of non-functional testing. Here below, the most common non-functional testing types are briefly schematized:

- Performance Testing: it is specifically aimed at verifying that the system meets the specified performance requirements. It usually targets validation of response times, throughput, resource-utilization levels and other specific performance characteristics of the software under test. However, the term performance testing is a general one that many time is associated to various attributes or characteristics of non functional testing. We refer to [69] for a specific definition of this term.

- Load Testing: it focuses on assessing the behavior of a system under pressure in order to detect load-related problems where the term load refers to the rate at which different service requests are submitted to the system under test (SUT) [61]. Possible related problems can be either functional problems that appear only under load (e.g, such as deadlocks, racing, buffer overflows and memory leaks) or violations in non-functional quality-related requirements under load (e.g., reliability, stability and robustness). We refer to [61] for more details about the most recent proposals and tools.

- Stress Testing: it is the process of putting a system far beyond its capabilities to verify its robustness and/or to detect various load-related problems. Contrary to load testing in which the maximum allowable load is generated, in stress testing, the load generated is more than what the system is expected to handle [61].

- Volume Testing: it is the process in which internal program or system limitations, such for instance storage requirements, are tried. It involves the ability of the systems to exchange data and information [69].

- Failover Testing: it validates a system's ability to be able to allocate extra resource when it encounters heavy load or unexpected failure so to continue normal operations [69].

- Security Testing: it assesses whether security properties related to confidentiality, integrity, availability, authentication, authorization, and non-repudiation are correctly implemented. Security testing demonstrates either the conformance with the security properties or addresses

known vulnerabilities by means of malicious, non-expected input data set [44]. The emerging security testing techniques integrate on different well known approaches such as: model-based testing, code-based testing and static analysis, penetration testing and dynamic analysis as well as regression testing. We refer to [44] for a complete overview of the most recent testing techniques and tools.

- Reliability testing: testing is used as a means to improve reliability; in such a case, the test cases must be randomly generated according to the operational profile, i.e., they should sample more densely the most frequently used functionalities [36].

- Compatibility Testing: it is the process for verifying whether the software can collaborate with different hardware and software facilities or with different versions or releases of the same hardware or software [69].

- Usability Testing: it evaluates the ease of using and learning the system and the user documentation, as well as the effectiveness of system functioning in supporting user tasks, and, finally, the ability to recover from user errors. This testing is particularly important when testing GUI [50].

- Scalability Testing: it is focused on verifying the ability of the software to increase and scale up on any of its non-functionality requirements such as load, number of transactions, volume of data and so on [36].

4 Test Challenges

Beyond decades of research activities, techniques and actors in software testing, there are still many aspects that remain more challenging due to the complexity, pervasiveness and criticality of continual software development. In this section, we overview several outstanding research challenges that need to be addressed to advance the state of the art in software testing and represent the directions to be followed for coping with the rapid advances of the software market.

4.1 Oracle Problem

An important component of testing is the oracle. Indeed, a test is meaningful only if it is possible to decide about its outcome. The difficulties inherent to this task, often oversimplified, had been early articulated in [92, 15]. Ideally, an oracle is any (human or mechanical) agent that decides whether the program behaved correctly on a given test. The oracle is specified to output a reject verdict if it observes a failure (or even an error, for smarter

oracles), and approve otherwise. Not always the oracle can reach a decision: in these cases the test output is classified as inconclusive.

In a scenario in which a limited number of test cases is executed, the oracle can be the tester himself/herself, who can either inspect a posterior the test log, or even decide a priori, during test planning, the conditions that make a test successful and code these conditions into the employed test driver [79].

Oracle can be derived by the analysis of textual (natural language) documentation describing the functionalities expected from the SUT to varying degrees. The partial and ambiguous nature of this documentation has often forced the manual oracle decisions. However, as overviewed in [15], some proposals try either to construct a formal specification exploiting user and developer documentations and source code comments; or to restrict the natural language to a semi-formal one enabling automatic processing.

When the tests cases are automatically derived, or also when their number is quite high, in the order of thousands, or millions, a manual log inspection or codification of the test results is not thinkable. Automated oracles must then be implemented.

In this case, a possible solution is to use pseudo-oracle [37], i.e. an alternative version of the program produced independently, e.g. by a different programming team or written in an entirely different programming language [15]. Alternatively, regression testing results can be used, i.e. the test outcome is compared with earlier version executions (which however in turn had to be judged passed or failed).

However, most of the available proposals for oracle definition are derived from a specification of the expected behavior [92]. According to [15] approaches based on formal specification can be classified into four categories:

- Model-based specification languages: the purpose is to exploit the models and a syntax that defines desired behavior in terms of its effects on the model so to derive the expected oracle. Different proposals discuss how to use the abstract specification to define high level test oracles [30, 90]. However, the models or the documents describing the specification could be very abstract, quite far from concrete execution output and consequently, oracle definition could result quite problematic;

- State transition systems: this kind of approaches focuses on the formal modeling of the system thought state transition systems. In particular, they focus on the reaction of a system to stimuli, i.e. *transitions*, and abstract a property of the states. A rigorous empirical evaluation of test oracle construction techniques using state transition systems is provided in [83];

- Assertions and contracts: if assertions could be embedded into the program so to provide run-time checking capability, conditions are instead

expressly specified to be used as test oracles. As consequence, the produced execution traces could be logged and analyzed so to derive the oracle verdicts [71];

- Algebraic specifications: the purpose is to define equations over program operations that hold when the program is correct. The software is represented in terms of equational axioms that specify the required properties of the operations. Typically, these languages employ first-order logic to prove properties of the specification, like the correctness of refinements. Abstract data types (ADT), which combine data and operations over that data, are well-suited to algebraic specification [92].

The formal specifications have the advantage that they can be used both for test case derivation and oracle specification as well. However, the gap between the abstract level of specifications and the concrete level of executed tests only allows for partial oracles implementations, i.e., only necessary (but not sufficient) conditions for correctness can be derived [92].

In view of these considerations, it should be evident that the *oracle* might not always judge correctly. So the notion of coverage of an oracle is introduced to measure its accuracy. It could be measured for instance by the probability that the oracle rejects a test (on an input chosen at random from a given probability distribution of inputs), given that it should reject it [15], whereby a perfect oracle exhibits a 100% coverage, while a less than perfect oracle may yield different measures of accuracy.

4.2 Full Automation

An important challenge of decades of research in software testing has been to improve the degree of attainable automation, either by developing advanced techniques for generating the test inputs or, beyond test generation, by finding innovative support procedures to automate the testing process [23]. Due to this, the market for tester support tools is in a huge expansion and recent forecasts show that it shall grow up to $34 billion worldwide by 2017, which opens relevant business opportunities for new innovative testing platforms.

Test automation tools promise to increase the number of tests they run and the frequency at which they run them by many orders of magnitude. Test automation can be incredibly effective, giving more coverage and new visibility into the software under test. However, 100% automatic testing remains still a dream [23]. Applicability of test automation is still limited and its adaptation to testing contains practical difficulties in usability.

Practitioners frequently report disastrous failures in the attempt to reduce costs by automating software tests, particularly at the level of system testing [89]. This is due to a gross underestimation of the money and time necessary to automate tests. Automating tests takes time due both to selection, building, installation and integration of the testing tools and to

planning and implementing automated tests. Often this effort is equivalent to that of manual tests. Automated tests may generate a lot of results that can take much more staff involvement and costs for analysis and isolation of discovered faults than manual tests [57].

The trade-off between automated and manual testing is discussed in literature. The authors of [94] present a cost model that uses the concept of opportunity cost to balance benefits, objectives and risks mitigation of automated and manual testing. The work in [53] presents an extensive review of experiences of test automation detailing the used tools, the application domain, the lifecycle development process, the project dimension as well as the successful or failure results. The challenge would be the full development of a powerful integrated test environment which by itself can automatically take care of possibly instrumenting the deployed software deriving drivers and stubs as well as the most suitable test cases, executing them and finally handling the obtained results providing a test report.

An attempt in this direction is represented by *perpetual testing*, also known as *continuous testing* [98], which uses machine resources to continuously run tests in the background, providing rapid feedback about test failures as source code is edited, reducing in this way wasted time by 92-98%.

The principle of continuous testing is today a key feature of DevOps which extends agile principles to entire software delivery pipeline [113]. The main idea is to move the testing process to early in software lifecycle but also allow the tests to be carried out on production like system executing the test suite on every software build generated automatically without any user intervention.

Finally, existing techniques aim to reduce automation costs by automating test automation. The main approach consists into a sequence of natural language test steps enabling a sequence of procedure calls with accompanying parameters that can drive testing without human intervention. This technique has been proven effective in reducing the cost of test automation by automating over 82% of the steps contained in a test suite [107].

4.3 Scalability of testing

The complexity of software systems is not only caused by their functionalities, but it is also due to the complexity of the platform and environment in which they run. Mobile devices, distributed, wireless networked and virtualised environment, large scale clusters, and mobile clouds are just some examples. Additionally, in settings like Software as a service (SaaS) [32], Big Data applications [119], Web of Systems [59], and Cyber-physical Systems [10], there is from one hand, the need of computation able to scale up as well as scale out; and from the other hand, an increasing demand of automated testing of these large scale software systems, as stated in Section 4.2.

Recent studies focus on scalability of mutation testing which requires much time and computational resources to execute the overall test suite against all mutants. An empirical study of scalability of selective mutation testing is proposed in [117]. The authors of [117] show how the program size and the total number of non-equivalent mutants affect the effectiveness of selective mutation testing based on the subsets of selected mutants. In particular, as either the program size or the total number of non-equivalent mutants increases, the number of selected mutants increases slowly, while the proportion of selected mutants decreases, evidencing a good scalability of selective mutation testing.

4.4 Test Effectiveness

Evaluating the program under test, measuring the efficacy of a testing technique or judging whether testing is sufficient and can be stopped, are important aspects of the software testing, which in turn would require having measures of the effectiveness. The following sections provide details about the different aspects of effectiveness.

4.4.1 Measuring the Software

For evaluating the program under test different measurements and approaches can be applied as reported in [45]:

- Linguistic measures: these are based on proprieties of the program or of the specification text. This category includes for instance the measurement of: Sources Lines of Code (LOC), statements, number of unique operands or operators, and function points;
- Structural measures: these are based on structural relations between objects in the program and comprise control flow or data flow complexity. These can include measurements relative to the structuring of program modules, e.g., in terms of the frequency with which modules call each other;
- Hybrid measures: these may result from the combination of structural and linguistic properties;
- Fault density: this is a widely used measure in industrial contexts and foresees the counting of the discovered faults and their classification by their type. For each fault class, fault density is measured by the ratio between the number of faults found and the size of the program.

4.4.2 Measuring the Testing Technique

For evaluating the testing approach, sometimes, coverage/thoroughness measures can be adopted. In this case, adequacy criteria evaluate the testing

approach through the percentage of exercised set of elements identified in the program or in the specification by testing. However, the widespread adopted methods focus on the evaluation of the fault-detection effectiveness of the considered testing technique. With different degrees of formalization, these proposals measure how much the test cases are able to revealing categories of likely or pre-defined faults. Among them, mutation analysis [60] is the standard technique to assess the effectiveness of a testing approach. In mutation testing, a mutant is a slightly modified version of the program under test, differing from it by a small, syntactic change. Every test case exercises both the original and all the generated mutants: if a test case is successful in identifying the differences between the program and a mutant, the latter is said to be killed. The underlying assumption of mutation testing and the coupling effect, is that, by looking for simple syntactic faults, more complex, but real, faults will be found. For the technique to be effective, a high number of mutants must be automatically derived in a systematic way.

Mutation Testing can be adopted at any level of testing (unit, integration level, system level) and can be applied both for white-box, black-box or gray-box testing. In literature, there are plenty of proposals and tools for many programming languages. We refer to [60, 6] for an overview of the most recent proposals about Fortran programs, Ada programs, C programs, Java programs, C# programs, and AspectJ programs. Mutation Testing has also been used for interfaces testing [38] or more in general during the design level [60, 20] for mutating Finite State Machines, Statecharts, Petri Nets and Network protocols. Recently Mutation testing has been applied to Web Services [6] or to Security Policies [76, 26].

4.4.3 Test cases Selection & Prioritization

In software testing, to improve fault detection rate at a given test execution time, common solutions rely on the application of proper strategies for test cases selection or prioritization. Test case selection aims to reduce the cardinality of the test suites while keeping the same effectiveness in terms of coverage or fault detection rate; test case prioritization aims at defining a test execution order according to some criteria (e.g., coverage, fault detection rate), so that those tests that have a higher priority are executed before the ones having a lower priority.

Many proposals address test cases selection for regression systems, aiming to speed up testing with a focus on detecting change-related faults [103]. The main idea is to select and run from the full test suite those tests that are relevant to the changes made to different software revisions, guaranteeing that the outcome of the tests that are not selected will not be affected by the changes. The effectiveness of regression test selection is measured by the ratio of the number of tests selected to run over the total number of tests [103].

Some recent proposals try to automate test cases selection techniques using information retrieval [74], others exploit the tracking of dynamic dependencies of tests on Java and JUnit files without require integration with version-control systems [51].

Test case prioritization techniques [104] schedule test cases in an order to increase::

- the rate of fault detection, namely, the likelihood of revealing faults earlier in a run of regression tests;
- the likelihood of revealing regression errors related to specific code changes earlier in the regression testing process [99];
- the coverage of code under test;
- the confidence in the reliability of the system under test.

In practice, the test case prioritization problem may be intractable, so test case prioritization techniques rely typically on heuristics. A review of such prioritization techniques is presented in [96]. In [97], several test prioritization techniques are used to increase the fault detection rate of test suites. More recent results [41] still confirm the effectiveness of test case prioritization based on fault detection rate and show the flexibility of the approach for application in different contexts. However, as demonstrated in [42], no prioritization metric is the best one for any system: indeed, the performance of the prioritization approach varies according to the considered application and could depend on the evaluated test suites [24]. Another proposal addresses time-constrained test prioritization in the context of integer programming [118]. An approach that is currently considered very promising is based on the notion of test similarity [114]: the intuition behind similarity-based prioritization is that when resources are limited and only a subset of test cases within a large test suite can be executed, then it is convenient to start from those that are the most dissimilar according to a predefined distance function. In the context of testing of access control systems, some prioritization criteria based on similarity of access control requests are presented in [24].

5 Testing Process

During recent years, software testing has increased its role in the process of development. It is no longer focused on the defects detection after code completion, but it is now an integrated and significant activity performed during the whole software life cycle. Its critical nature and the importance for the overall quality of the final products led adopting the good practice of starting its management at the early stages of software development during

the requirements analysis and proceeding with its organization systematically and continuously during the entire development process up to the code level.

Without referring to any specific software development process, usually the basic development phases can be summarized as in the following [6]:

- Requirements Analysis: services, constraints, goals, and features of the overall system are established and organized. The list of requirements specifies the guidelines that the project must adhere to. At the end of this phase a set of documented, actionable, measurable, testable, and traceable requirements should be defined to a level of detail sufficient for system design;

- System and Software Design: this phase targets both hardware and software requirements and establishes the overall system architecture. The representation of the system is usually provided in terms of functionalities easily transformable into one or more executable applications/programs;

- Implementation: the established design is implemented in order to satisfy the list of requirements identified in the requirement analysis and definition phase. The implemented units, or subsystems are integrated into a complete system to be properly tested;

- Validation and Testing: quality attributes as well as requirement assessment are calculated and validated. Different test techniques can be included in the process development, each one targeting a different aspect of the proposed system.

Considering in particular the Validation and Testing phase, even if its management can depend strictly on the development process adopted for delivering the software products, the main phases can be resumed in [6]:

- Planning: as for any other process activity, the testing must be planned and scheduled. Thus, the time and effort needed for performing and completing software testing must be established in advance during the early stages of development. This also includes the specification of the personnel involved, the tasks they must to perform and the facilities and equipments they may use;

- Test cases generation: according to the test plan constraints a set(s) of the test cases must be generated by using a (several) test strategy (ies);

- Test cases execution: the test cases execution may involve testing engineers, outside personnel or even customers. It is important to document every action performed in order to allow the experiments' duplication and meaningful and truthful evaluations of the results obtained;

- Test results analysis: the collected testing results must be evaluated to determine whether the test was successful (the system performs as expected, or there are no major unexpected outcomes) and used for deriving measures and values of interest;

- Problem reporting: a test log documents the testing activity performed. This should contain for example the date in which a test was conducted, the data of the people who performed the test, the information about the system configuration and any other relevant data. Anomalies or unexpected behaviours should be also reported;

- Post-closure activities: the information relative to failures or defects discovered during testing execution are used for evaluating the performance and the effectiveness of the developed testing strategy(ies) and determining whether the process development adopted needs some improvements.

In the remaining of this Section some of the most important development processes are briefly introduced and the role of testing inside them discussed.

5.1 Sequential Models

One of the first models to be designed was the waterfall model (W-model), also known as the cascade model, where the software development lifecycle was constructed from a sequential set of stages [6]. It starts with a requirements analysis at the top level and finishes with the testing activity, so that defects are discovered close to the releasing, with a deep impact on the overall development costs. Usually, the waterfall model is the recommended framework for creating software which provides back-end functionality, meaning a software whose main scope is to provide a service for other applications.

The V-model, is the successive variation of the W-model, where a V shape folded at the coding level is included. It was important to demonstrate that testing activities should be planned and designed as early as possible in the lifecycle. Sometimes the V-model can be used in large projects which can include several subsystems and 3rd party systems.

The most common alternatives of the sequential approaches are evolutionary ones which are based on iterations. In this case, the system is not defined in advance but evolves through the evaluation of the achieved intermediate results.

5.2 Iterative Models

During the last ten years, part of software research has been dedicated to the improvement of the development process (Software Process Improvement (SPI) initiatives). In this context, the CMM model (Capability Maturity

Model) [88], developed at the Software Engineering Institute (SEI) is a de facto reference used by thousands of organizations together with the SPICE framework [43]. We report below a brief description of the commonly used process assessment models referring to [43] for further details.

As summarized in [88], the CMM model is a framework that describes the elements required for an effective software process. In particular, it focuses on an evolutionary improvement path from an ad hoc, immature process to a mature, disciplined process. It presents sets of recommended practices in a number of key process areas that have been shown to enhance software development and maintenance capability. The CMM guides developers in gaining control of their development and maintenance processes, and evolving toward a culture of software engineering and management excellence. CMM is not a framework that advocates magical and revolutionary new ideas, but it is in fact a tailored compilation of the best practices in software testing. Specifically, it improves quality of software delivered, it increases the customer satisfaction; it helps in achieving targeted cost savings; it also ensures stability and consistent high performance.

Other methods for managing the program improvement are the IDEAL framework [77] defined at the SEI, and the Rational Unified Process (RUP) [66].

The IDEAL method is an integrated approach for SPI defined by the SEI which identifies five phases: i) Initiating, which specifies the business goals and objectives that will be realized or supported; ii) Diagnosing, which identifies the organization's current state with respect to a related standard or reference model; iii) Establishing, which develops plans to implement the chosen approach; iv) Acting, which brings together everything available to create a best guess solution specific to organizational needs and puts the solution in place; v) Leveraging, which summarizes lessons learned regarding processes used to implement IDEAL.

The Rational Unified Process (RUP), which is a detailed refinement of the Unified Process (UP) [66], presents itself as a web-enabled software engineering process useful for: improving team productivity, delivering of software best practices to all team members, guiding the user in applying UML during the process development, and providing an extensive set of guidelines, templates, and examples. It is in particular a customizable framework, adaptable to the different organization exigencies, supported by tools (tightly integrated with Rational tools) which automates a large part of the process development. A central role of this process is represented by the RUP Best Practices, which are mainly guidelines for a well-established process development. RUP identifies six best practices which are: Develop Software Iteratively, Manage Requirements, Use Component-Based Architectures, Visually Model Software, Verify Software Quality, Control Changes to Software.

The RUP structure is characterized by: a static structure that describes the process (who is doing what, how and when); dynamic structure that

details how the process rolls out over time; an architecture-centric process that defines and details the architecture; a Use-Case Driven Process which specifies how use cases are used throughout the development cycle.

The RUP encourages testing early by offering a number of mechanisms to integrate testing more closely with the software development effort. In particular, RUP targets to: i) making Test a distinct discipline; ii) using an iterative development approach; iii) continuously verifying quality and letting use cases drive development; iv) scheduling implementation based on risk; v) managing changes strategically; vi) using the right-sized process.

5.3 Agile development process

The term *agile software development* was created by the Agile Manifesto [46]. Agile software development is basically an iterative approach that focuses on incremental specification, design, and implementation, while requiring full integration of testing and development. Agile development process has been originated by a precedent development practice, the Rapid Application Development (RAD) methodology [28]. RAD uses a large number of iterations, with every iteration being a complete development cycle. At the end of each iteration, a complete executable product is released. The iteration product is a subset of the complete desired product and it will be increased from iteration to iteration until the final product is released. Natural evolution of RAD are therefore the nowadays known agile software development approaches.

In the agile environment, testing is a frequent activity as small amounts of code are tested immediately upon being written. According to the Manifesto for Agile Software Development, agile main points are: i) individuals and interactions over processes and tools; ii) working software over comprehensive documentation; iii) customer collaboration over contract negotiation, and iv) responding to change over following a plan.

The intent is to produce high quality software in a cost effective and timely manner, and in the meantime, meet the changing needs of end users. In an agile environment, testing is often included within development in the form of Test Driven Development (TDD) [46], which is a programming technique that promotes code development by repeating short cycles. It combines Test-first development with refactoring and methods involving four steps: i) write a test for an unimplemented functionality or behaviour; ii) supply the minimal amount of code to pass the test; iii) refactor the code; iv) check that all tests are still passing after the changes were done.

By developing only enough code to pass the tests and then using the refactoring as an improvement method for the design quality, TDD leads to better code quality and improves the confidence in the code as well as increasing the productivity.

Recent applications of agile software development are the XP (eXtreme

Programming) and Scrum [65] development methods. In eXtreme Programming (XP), after a short planning stage, development goes through analysis, design, and implementation stages quickly (from one to four weeks). XP success relies on skilled and well prepared software developers that are able to improve development quality and productivity.

The Scrum is more focused on the delivering of objected-oriented software [65]. Origin of term Scrum came from the popular sport Rugby, because rugby strategies have been used the first time to describe its hyper productive development processes that are: i) a holistic team approach; ii) constant interaction among team member; iii) unchanging core team members.

Basic adjectives characterizing Scrum implementations are: transparency (visibility), inspection, and adaptation. Transparency or visibility means that any aspect of the process that affects the outcome must be visible and known to everybody involved in the process. Inspection requires that various aspects of the process must be inspected frequently enough so that unacceptable variances in the process can be detected. Adaptation requires that the inspector should adjust the process if one or more aspects of the process are in an unacceptable range.

6 Domain Specific Testing

Recent years have witnessed the emergence of domain-specific approaches in software testing. These approaches are tailored to the specific needs of the domain and leverage domain knowledge to adapt and customize well-known testing solutions. Specifically, in the last decades software development is driven by emerging trends such as the widespread diffusion of mobile technology, cloud infrastructures adoption as well as big data analysis and software as a service paradigm that point out new constraints and challenges for the testing activity. In the following of this section, we give a brief overview of different aspects and solutions for software testing applied into several domain-specific environments.

6.1 Cloud-based Testing

Cloud computing is henceforth an accepted alternative for deploying applications and services. Businesses desire to achieve higher-level operational performance and flexibility while keeping the development and deployment cost as lower as possible. Meanwhile, cloud computing has an high impact in software testing with the diffusion of a new testing paradigm known as Testing as a Service (TaaS). TaaS in a cloud infrastructure is a new business and service model, which provides static or dynamic on-demand testing services in the cloud and delivers them as a service to customers.

The main advantages of cloud-based testing deal with [47, 35]:

- Reducing the testing costs and time. Cloud computing offers unlimited storage as well as virtualized resources and shared cloud infrastructures that can help to eliminate required computer resources and licensed software costs as well as to reduce the execution time of large test suites in a cost-effective manner. This allows large IT companies to support many production lines which require diverse computing resources and test tools;

- Performing on-demand test services to conduct large-scale performance and scalability online validation;

- Performing testing of dynamic, complex, distributed applications such as mobile and web applications, leveraging multiple operating systems and updates, multiple browser platforms and versions, different types of hardware and a large number of concurrent users.

The authors of [35] provide a systematic survey of cloud based testing techniques including model based testing, performance testing, symbolic execution, fault injection testing, random testing, privacy aware testing and others.

There are three main different forms of testing as service (TaaS) in a cloud environment. Each of them has different focuses and objectives [48, 47]:

- Testing on clouds. In this form, software applications are deployed and executed on a cloud, and validated using the provided test services given by TaaS vendors. The major objective here is to take the advantage of large-scale test simulations and elastic computing resources on a cloud;

- Testing over clouds. In this form, software applications are deployed and validated based on different clouds (such as private clouds, public or hybrid clouds). A typical software system crossing multiple clouds is structured with components, service software, and servers deployed crossing over several clouds;

- Testing of a cloud. It validates the quality of a cloud infrastructure according to the specified capabilities and service requirements.

Besides the many advantages in using cloud-based systems, there are yet many realistic problems of cloud service testing that need to be solved [48, 47]:

- On-demand test environment set up. There is a lack of supporting solutions to assist engineers to set up a required test environment in a cloud using a cost-effective way. To overcome these limitations, solutions aiming to empower the cloud applications with self-configuration, self-healing, and self-protection capabilities are provided;

- The heterogeneity and lack of standards in test tools and their connectivity and interoperability to support testing services;
- Assuring and assessing user privacy and security of cloud based applications inside a third-party cloud infrastructure.

Finally, the authors of [11] provide a survey of typical tools for cloud testing. The main requirements for these tools include multi-layer testing, SLA-based testing, large scale simulation, and on-demand test environment.

6.2 SOA Testing

As described in Section 3.3, traditional testing approaches are divided into three major classes: black-box, white-box and grey-box testing. Considering the specific domain of SOA testing while black-box approaches keep the same target of traditional ones, white-box and grey-box proposals assume a slightly different meaning. Indeed, in SOA white-box testing two different points of view can be identified: coverage measured at the level of a service composition (orchestration or choreography) and coverage of a single service internals. Generally, validation of service orchestrations is based on the Business Process Execution Language description [1] considered as an extended control flow diagram. Classical techniques of white-box coverage (e.g., control flow or dataflow) can be used to guide test generation or to assess test coverage so as to take into consideration the peculiarities of the Business Process Execution Language. Other proposals are instead based on formal specification of the workflows, e.g., Petri Nets and Finite State Processes used for verifying specific service properties [17]. Considering a service choreography, existing research focuses, among others, on service modeling, process flow modeling, violation detection of properties such as atomicity and resource constraints, and XML-based test derivation.

If, on the one side, there are several approaches for structural testing of service compositions, there are few proposals for deriving structural coverage measures of the invoked services. The reason for this is that independent web services usually provide just an interface, enough to invoke them and develop some general (black-box) tests, but insufficient for a tester to develop an adequate understanding of the integration quality between the application and independent web services.

In the rest of this section, a survey of some proposed approaches and tools for supporting SOA testing is presented. Additionally, since black, white and grey box testing, albeit successfully executed, do not prevent security weaknesses, an overview of this important aspect is presented in Section 6.2.4.

[1] WSBPEL available at: `https://www.oasis-open.org/committees/tc_home.php?wgabbrev=wsbpel`

6.2.1 SOA Black-box Testing

As a common guideline, SOA black-box testing relies on the functionality provided by the Web Services. Commonly, the use of the (formal) specification or the XML Schema datatype [2] available allows for the generation of test cases for boundary-value analysis, equivalence class testing or random testing [17]. The derived test suite could have different purposes, such as to prove the conformance to a user-provided specification, to show the fault detection ability assessed on fault models, or to verify the interoperability by exploiting the invocation syntax defined in an associated WSDL (WS Description Language) document [3]. In this last case the formalized WSDL description of service operations and of their input and output parameters can be taken as a reference for black box testing at the service interface [19, 91]. More details about SOA black-box testing are provided in [31, 17]. Despite the different proposals, test automation is still an open issue in this context [56]. In the specific case of web services, the use of XML-based syntax of WSDL documents could support fully automated WS test generation by means of traditional syntax-based testing approaches.

6.2.2 SOA White-box Testing

Most often white-box testing of SOA applications is devoted to the validation of Web Service Compositions (WSC). Validation of WSCs can be usually addressed performing structural coverage testing of a WSC specification. As overviewed in [31, 17] the WSC specification can be abstracted as an extended control flow diagram or transformed into formal specification so to apply structural coverage criteria (e.g. transition coverage). Similar complexity explosion problems may be encountered in such methods, since the amount of states and transitions of the target model can be very high. Alternative proposals adopt either model-checking techniques for conformance testing by generating test cases from counterexamples, or use the formal model of the WSC and the associated properties to verify properties such as reachability. However, the complexity of the involved models and model-checking algorithms in some cases could make these approaches hardly applicable to real-world WSCs. In [18, 58] an overview of the SOA testing specifically based on data-related models is provided.

6.2.3 SOA Grey-box Testing

In the context of Service Oriented Architectures (SOAs), where independent web services can be composed with other services to provide richer functionality, interoperability testing becomes a major challenge. Independent web

[2] XML Schema description: `https://www.w3.org/XML/Schema#dev`

[3] WSDL description: `https://www.w3.org/TR/wsdl20-primer/`

services usually provide just an interface, enough to invoke them and develop some general functional (black-box) tests, but insufficient for a tester to develop an adequate understanding of the integration quality between the application and the independent web services. To address this lack, grey box proposals, trying to "whitening" the SOA testing, can be considered [16, 116]. In such grey-box proposals, the main idea is to try to derive data during the service execution so that produced test traces can be collected and analyzed against the specification paths. Many times, the collection of service execution traces is associated with the possibility to introducing an "Observer" stakeholder into the ESOA (Extended SOA) framework. In ESOA, the goal is either to monitor code coverage [16], or to monitor ESOA services (passive testing) against a state-model [21].

6.2.4 SOA Security Testing

Security aspects are highly critical in designing and developing web services. It is possible to distinguish at least two kinds of strategies for addressing protective measures of the communication among web services: security at the transport level and security at the message level. Enforcing the security at the transport level means that the authenticity, integrity, and confidentiality of the message (e.g., the SOAP message) are completely delegated to the lower-level protocols that transport the message itself from the sender to the receiver. Such protocols use public key techniques to authenticate both the end points and agree to a symmetric key, which is then used to encrypt packets over the (transport) connection. Since SOAP messages may carry vital business information, their integrity and confidentiality need to be preserved, and exchanging SOAP messages in a meaningful and secured manner remains a challenging part of system integration. Unfortunately, those messages are prone to attacks based on an on-the-fly modification of SOAP messages (XML rewriting attacks or XML injection) that can lead to several consequences such as unauthorized access, disclosure of information, or identity theft. Message-level security within SOAP and web services is addressed in many standards such as WS-Security [4], which provides mechanisms to ensure end-to-end security and allows to protect some sensitive parts of a SOAP message by means of XML Encryption and XML Signature [44].

The activity of fault detection is an important aspect of (web) service security. Indeed, most breaches are caused when a system component is used in an unexpected manner. Improperly tested code, executed in a way that the developer did not intend, is often the primary culprit for security vulnerability. Robustness and other related attributes of web services can be assessed through the testing phase and are designed first off by analyzing

[4] `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`

WSDL document to know what faults could affect the robustness quality attribute of web services, and secondly by using the fault-based testing techniques to detect such faults. Focusing in particular on testing aspects, the different strategies and approaches that have been developed over the years can be divided into passive or active mechanisms [44, 17].

Passive mechanisms consist of observing and analyzing messages that the component under test exchanges with its environment and are specially used either for fault management in networks or for checking whether a system respects its security policy. Active testing is based on the generation and the application of specific test cases in order to detect faults.

All the techniques have the purpose of providing evidence in security aspects, i.e., that an application faces its requirements in the presence of hostile and malicious inputs. Like functional testing, security testing relies on what is assumed to be a correct behaviour of the system, and on non-functional requirements. However, the complexity of (web) security testing is bigger than functional testing, and the variety of different aspects that should be taken into consideration during a testing phase implies the use of a variety of techniques and tools. An important role in software security is played by negative testing [44, 17], i.e., test executions attempting to show that the application does something that it is not supposed to do. Negative tests can discover significant failures, produce strategic information about the model adopted for test case derivation, and provide overall confidence in the quality and security level of the system. Other common adopted methodologies and techniques include [44, 17]:

- Fuzz testing: it involves generating semivalid data and submitting them in defined input fields or parameters (files, network protocols, API calls, and other targets) in an attempt to break the program and find bugs. Semivalid data are correct enough to keep parsers from immediately dismissing them, but still invalid enough to cause problems. Fuzzing covers a significant portion of negative test cases without forcing the tester to deal with each specific test case for a given boundary condition. In the specific area of web service security, fuzzing inputs can often be generated by programmatically analyzing the WSDL or sample SOAP requests and making modifications to the structure and content of valid requests. File fuzzing strategy is also used for detecting XML data vulnerability;

- Injection: in general, web services can interact with a variety of systems and for this reason they must be resistant to injection attacks when external systems are accessed or invoked. The most prevalent injection vulnerabilities include SQL injection, command injection, LDAP injection, XPath injection, and code injection. Recently, SQL Injection Attack has become a major threat to web applications. SQL injection occurs when a database is queried with an SQL statement which

contains some user-influenced inputs that are outside the intended parameters range;

- Policy-based testing: an important aspect in the security of modern information management systems is the control of accesses. Data and resources must be protected against unauthorized, malicious or improper usage or modification. For this purpose, several standards have been introduced that guarantee authentication and authorization, such for instance the eXtensible Access Control Markup Language (XACML) [5], and to rule the writing of the access control policies. Thus policy-based testing is the testing process to ensure the correctness of policy specifications and implementations. By observing the execution of a policy implementation with a test input (i.e., access request), the testers may identify faults in policy specifications or implementations, and validate whether the corresponding output (i.e., access decision) is as intended. Although policy testing mechanisms vary because there is no single standard way to specify or implement access control policies, in general, the main goals to conduct policy testing are to ensure the correctness of the policy specifications, and the conformance between the policy specifications and implementations. The recent approaches on XACML policy testing are divided in the following main categories: i) Fault Models and Mutation Testing that are based on a fault model to describe simple faults in XACML policies; ii) Testing criteria that determine whether sufficient testing has been conducted and it can be stopped, and measure the degree of adequacy or sufficiency of a test suite. They include structural coverage criteria and fault coverage criteria; iii) Test generation proposals specifically focused on the access control policies [75, 25].

6.3 GUI-based Testing

Currently, with the widespread use of graphical user interface (GUI), the GUI testing is receiving a lot of attention. However, the GUI development includes different activities that force to face the GUI testing from different perspectives such as test coverage, test case generation, test oracle and regression testing. Among these, the test case generation is one of the most important and therefore the one that received most attention in literature. As stated in [80] some proposals for GUI testing can be based on the manual derivation of test cases, i.e. method calls to the instances of the classes under test. Assertions are inserted in the test cases to determine whether the classes/methods are executed correctly. However, because manual coding of test cases can be tedious, capture/replay techniques can be adopted. These proposals capture sequences of events that testers perform manually on the

[5] `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`

GUI, define the test case as a sequence of input events and replay derived test cases automatically on the GUI.

Referring to the classification provided into [93] the available proposals can be grouped as in the following:

- Based on Finite State Machines: this kind of approach is the most adopted for the GUI testing. Usually the input and output sequences are used for generating hierarchal models based on them. FSM can be then exploited for different testing approaches such as coverage for all-paths and all-transitions. Peculiarities of such kind of testing are [14]: i) GUI has an enormous number of states and every state should be tested; ii) the GUI input space is very large because of permutations of inputs and events which effect the GUI; iii) the complex dependencies cannot be avoided in the GUI system; iv) the GUI system can cause external effects at any time;

- Goal-Driven approaches: these proposals exploit the definition of predefined goals that can be used both as input and as output, by generating sequences of actions, which reach these goals. These sequences of actions are used as test cases for GUIs. This technique allows the analysis of user interactions using model checking, and the synthesis of user interactions to executable GUI applications;

- Based on abstractions: in these proposals, abstractions may be based on structural features of GUI applications, e.g., the enabledness of a button (enabled or disabled) using a boolean value, or the current value of slider control using an integer value. Abstraction can be either manually inferred or automatically derived by static analysis of the code [9];

- Model-Based approaches: all these techniques require the creation of a model of the software or its GUI, and algorithms to use the model to generate test cases. In some cases, the models are created manually; in others, they are derived in an automated manner. Also the test-case generation for some techniques is manual, but for most is automated. In [80] an exhaustive overview of the most recent proposals is provided.

6.4 Mobile Testing

Rapid advances of mobile technology and wireless networking push the wide adoption of mobile applications in different critical domains, from banking to mobile cyber-physical systems, to patients monitoring. According to the recent studies, the market for cloud-based mobile applications will grow exponentially in the next years. To guarantee the reliability and security of such mobile applications, their testing is required on many different configurations

of devices and operating systems. Approaches to make such testing automated, systematic and measurable are needed. According to recent studies [3], the revenues from mobile application testing tools will reach $800 million by the end of 2017 due to the test automation demand for mobile applications. What makes testing mobile software more demanding than testing computer software is the great complexity of the environment. Specifically, the diversity of relevant devices, operating systems, and networks as well as the fast updates of mobile browsers and mobile platforms and technologies make the testing of mobile web applications very expensive.

Recent research addresses Mobile Testing as a Service (MTaaS) [49] as a possible solution to reduce the complexity of mobile testing by leveraging a cloud-based scalable mobile testing environment to assure pre-defined quality of service requirements and service-level agreements. The authors of [49] discuss about issues and solutions of MTaaS proposing a mobile testing as a service infrastructure aiming at reducing high costs in current mobile testing practice and environments and supporting mobile scalability test. The work in [105] presents a set of existing cloud-based services and tools for mobile application testing whereas [112] proposes a framework, called Automated Mobile Testing as a Service (AM-TaaS), which offers automated test for mobile applications including emulation of mobile devices with different characteristics under virtual machines and cloud infrastructure. However, well-defined test models and processes that address the distinct needs of mobile testing are yet lacking. In particular, quality of mobile applications has different meaning and criticality for different users and in different contexts. Nowadays, the main challenge about mobile testing is improving its cost effectiveness by combining:

- Use of test automation: traditional testing of mobile applications, often done by manual execution of test cases and visual verification of the results, is an effort-intensive and time-consuming process. Automating the testing activities produces a great reduction of costs and effort in the whole mobile applications software development lifecycle;

- Use of emulators and actual devices: emulators can be useful for testing features of the application that are device independent. However, due to diversity in mobile hardware, actual devices will be used for validating the results;

- Testing the mobile environment for application complexity: this is needed in order to address the varieties of mobile and wireless technologies, like Wi-Fi, WiMax, 3G, 4G, etc., as well as to validate performance and robustness of the mobile system in real-world network conditions;

- Testing on heterogeneous mobile platforms: the applications will be

tested on different platforms for checking compatibility with different mobile operating systems;

- Large-scale on-demand mobile test services: they allow to implement testing techniques and strategies responding to on-demand requests of test generation, execution, and control.

6.5 Big Data Testing

Big data provides not only large amounts of data but also various data types such as images, videos, interactive maps, time stamps and associated metadata. The main four characteristics of big data are: volume, variety, velocity, and value. Data is continuously being generated from machines, sensors from Internet of Things, mobile devices, network data traffic, and application logs. This requires advances in data storage, mining and business intelligence technologies making it possible to preserve increasing amounts of data generated directly or indirectly by users and analyze it to yield valuable new insights [81].

Big data also presents new challenges with respect to their maintenance, testing and benchmarking. While big data systems have advanced their capabilities of data analysis, scalability, processing and fault tolerance, database testing and benchmarking have not moved forward to provide data generators, data sets, and workloads. The authors of [4] provide a comparative analysis of techniques for big data testing, including genetic algorithms, clustering techniques, performance and regression test approaches. In particular, the emerging challenges in the field of testing big data relate to the need of quickly generating huge, realistic and scalable data sets, as well as the need for well-defined workloads that capture the nature of novel, modern analysis tasks [5].

Experimental studies often reuse data sets from well-known standardized benchmarks for performance evaluation of database systems, like TPC-C [1], and XMLGen [2]. They typically provide open-source tools for data and workload generation, which can be easily adapted and used by third parties reducing the overall effort required to prepare and execute the experiment [5]. An alternative approach deals with the implementation of custom data generators for the comparison of approaches for large-scale data analysis.

The main limitations of existing approaches of benchmarking and testing of big data systems are about the lack of realistic data sets. Existing data sets are build on simplistic assumptions such as uniform distributions or oversimplified schema that often are not representative for real-world data. An attempt to overcome these limitations is to automatically extract the domain information from a ground truth data set, which is often available in practice. This information is then integrated into a data generator specification for a specific target environment and then used to create a concrete

data generator instance that is able to mimic the original data set [5].

Different solutions to generate representative data sets in the big data context and to control the size of the test set, deal with input space partitioning testing associated to input-domain model. The tester partitions the input-domain model, selects test values from partitioned blocks, and applies combinatorial coverage criteria to generate tests. Parallel computing and Hadoop are used to speed up data generation [70].

Finally, the evaluation and testing of big data analytic systems requires more complex, realistic, and universally useful workload specifications that involve machine learning algorithms, information extraction, and graph analysis/mining.

6.6 Automotive Testing

In automotive domain, one of the most important aspects is security. Currently, existing solutions for security assessment of software systems basically perform either static analysis (i.e., the software is not executed), or dynamic analysis (performed by executing the software on specific inputs), although any life cycle will likely apply a combination of both. A 2009 survey carried out within the NIST SAMATE project [6], provides an analysis of more than 70 tools either specifically conceived for security assessment or more generally for software correctness related to security, and catalogues them under the following categories:

- Static Analysis: aids analysts in locating security-related issues;
- Source Code Fault Injection: the source code is instrumented by inserting changes and then executed to observe the changes in state and behavior that emerge;
- Dynamic Analysis: refers generically to security testing approaches, such as coverage-based or profiling;
- Architectural Analysis: aims at identifying flaws in the software architecture and determining resulting risks to information assets;
- Pedigree Analysis: identifies software coming from an external source (e.g., open source software);
- Binary Code Analysis and Disassembler Analysis: both categories review binary code. Since source code is not needed, they can be applied to Commercial Off-The-Shelf (COTS) components;
- Binary Fault Injection: focuses on likely faults in the real-time operation of the software (e.g., memory faults or other error conditions provided by the processor);

[6] `http://samate.nist.gov`

- Fuzzing: a form of negative testing, in which the software is exercised under random invalid data, generally specific to a particular type of input;
- Malicious Code Detectors: search for malicious logic embedded in programs;
- Bytecode Analysis: bytecode contains more semantic information about the program execution than an equivalent binary. Bytecode analysis tools are actively investigated.

Many works try to transfer existing security assessment methods to the automotive domain, addressing specific risks and challenges of automotive software. Nowadays, several works [95] concur that security cannot be addressed separately from safety in the software engineering process of automotive systems, thus assessment approaches should converge and complement each other.

The widespread utilization of model-based design and production code generation in the automotive development process enables the application of automatic verification and validation techniques earlier in the software lifecycle, and at an higher level of abstraction using for instance Simulink models [108].

However, many testing methodologies and tools are customized to address specific challenges of the automotive domain. In [7] a detailed survey of the most recent proposals is provided.

Other security testing activities are aimed at finding implementation errors that could be exploited by an outside attacker to cause potential functionality problems. Such activities are also used to establish to what extent the target system can resist an attack and generally consist of: i) functional automotive security testing able to ensure general compliance with the specifications and standards for the security functionality implemented, encryption algorithms and authentication protocols of a vehicular IT system; ii) vulnerability scanning applied to all relevant applications, source codes, networks, and backend infrastructures of an automotive system in order to test the system for common security vulnerabilities such as security loopholes or security configurations with known weaknesses taken from a continuously updated database of automotive security vulnerabilities; iii) fuzzing that is used to expose the implementation to unexpected, invalid, or random input in the hope that the target will react in an unexpected way and, as a result, uncover new vulnerabilities; iv) penetration tests, applied in a final step to test security of the whole system by applying attacking methods conducted by trusted individuals to check whether ECUs are vulnerable and could allow unauthorized users access.

7 Discussion and Conclusions

This Chapter presented a journey through the world of software testing and its emerging techniques, ranging over many fields from definition to organization, from its applicability and analysis of effectiveness, to the challenges and specific issues of some of the most important application domains. However, due to the vast and articulated research discipline, the covering into one chapter of all ongoing and foreseen research directions and emerging technologies is impossible. Indeed, this Chapter proposes broadness against depth overview and it is an attempt to depict a comprehensive and extensible roadmap of the most important topics, challenges and future directions of testing activity. Figure 1 provides a graphical overview of the main chapter contents.

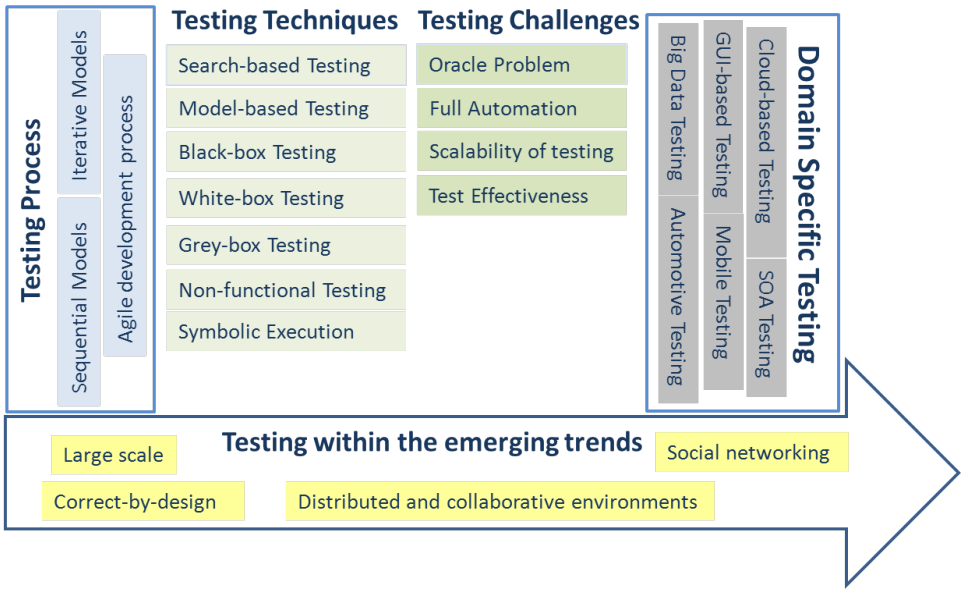

Figure 1: Software Testing Roadmap

This Chapter would be a reference of the most recent testing techniques and presented an almost complete overview of the new methods, approaches and tools useful to the reader for managing, controlling and evaluating software testing development. It targets both students, researchers, software developers and practitioners looking for solutions for their problems and improvements in the different activities of the testing process. As emerged from this Chapter, software testing is not an isolated activity; it has many fruitful relations between different areas of software engineering and many times it is the bridge between different disciplines.

Due to the increasing complexity of software systems and the variety of execution environments, software testing will continue to be a key activity

in the software development process and quality assurance.

Over the years software testing has had to keep the path of the trends, innovations and modifications provided by the new software development paradigms. In order to trigger always updated and practical solutions, software testing needs to learn from the state of practice and to elaborate the future vision useful to predict the problems, market exigencies and quality issues that could rise. In particular, the growth of cloud computing bring the need of benchmarks and on-demand test environment construction to measure the performance and scalability metrics of new applications considering the special features of cloud such as dynamic scalability, scalable testing environments, SLA-based requirements, and cost-models. New testing methods will be developed to assess compatibility, interoperability and multi-tenancy ability of cloud applications that must be able to work across multiple environments, various cloud platforms, client technologies, and browsers.

Possible suggestions for more in-depth and effective methodologies and approaches can be the use of realistic settings and large-scale-systems as well as the simulation or involvement of real stakeholders working simultaneously in distributed environments. In particular, large scale and software intensive IT systems are created by integrating and orchestrating independently controlled and managed systems. The most limiting factor of their development is software validation, which typically requires very costly and complex testing processes. Existing approaches for testing these systems focus on orchestrating the different phases of the testing process including test operation, test injection, monitoring and reporting, sometimes according to response-time, bandwidth-usage, throughput and adaptability, with the aim of continuous integration and deployment. New testing techniques able to deal with the inherent complexity of these systems will be based on the *divide-and-conquer* principle, which is commonly used for architecting complex software. They will leverage a novel test orchestration theory and toolbox enabling the creation of complex test suites for large systems as the composition of simple testing units. These new solutions will allow the reusability of testing knowledge, architectures and code, making testing activity more effective and less expensive. Moreover, these new testing techniques, based on the orchestration topology of simple test cases, will also target the important problem of automated partial oracle derivation of tests of large systems by inferring test oracle specification from the composition of expected and obtained outputs of the executed testing units.

An important role is given by the possibility of leveraging the software testing proposals from the specific programming language and execution environment so that high-level, application-independent attributes such as trust, security, performance and so on can be easily and automatically verified with a drastic reduction of testing time and effort.

Considering in particular the new collaborative and distributed software development processes such as Agile, Scum and DevOp, the testing tech-

niques should make easier the simultaneous verification and the automatic alignment of test activity with decomposition and distribution of the target systems. Additionally, due to the short development time, new proposals drastically decreasing learning time and addressing the multiple levels of IT personnel are becoming a real pressing so improve software testing and productivity.

Moreover, due to the increasing popularity of social networking services and their massive integration into the users' everyday life, another future target of testing activity is represented by the specific requirements of these services. Besides, performance and scalability validation of the decentralized online social networks, new testing techniques will be devoted to assess security and privacy of users' personal data in order to prevent local attacks. These techniques will specifically target assessment of privacy and sharing information policies as well validation of decentralized management of users' social profiles and data storage solutions.

Finally, the recent trend of adopting correct or secure-by-design development process forces testing activity to propose tools and methodology useful for testing and analyzing design level artifacts. In particular, simulation, model checking and model-based approaches should be the key success for improving the quality of the developed systems.

References

[1] TPC Benchmarks & Benchmark Results. `http://www.tpc.org/`. Online; accessed April 2017.

[2] XMark-An XML Benchmark Project. `http://www.xml-benchmark.org/`. Online; accessed April 2017.

[3] ABI Research's Mobile Application Technologies. $200 Million Mobile Application Testing Market Boosted by Growing Demand for Automation. `https://www.abiresearch.com/press/200-millionmobile-application-testing-market-boos`.

[4] A. Abidin, D. Lal, N. Garg, and V. Deep. Comparative analysis on techniques for big data testing. In *2016 Int. Conf. on Information Technology (InCITe) - The Next Generation IT Summit on the Theme - Internet of Things: Connect your Worlds*, pages 219–223, Oct 2016.

[5] Alexander Alexandrov, Christoph Brücke, and Volker Markl. Issues in big data testing and benchmarking. In *Proc. of the Sixth Int. Workshop on Testing Database Systems*, page 1. ACM, 2013.

[6] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[7] Bertolino Antonia, Calabro' Antonello, Di Giandomenico Felicita, Lami Giuseppe, Lonetti Francesca, Marchetti Eda, Martinelli Fabio, Matteucci Ilaria, and Mori Paolo. Secure Software Engineering for Connected Vehicles: A Research Agenda. `http://www.iit.cnr.it/en/node/36711`. TR Technical reports n. IIT TR-18/2015.

[8] A. Arcuri and L. Briand. Formal analysis of the probability of interaction fault detection using random testing. *IEEE Transactions on Software Engineering*, 38(5):1088–1099, Sept 2012.

[9] Stephan Arlt, Evren Ermis, Sergio Feo-Arenis, and Andreas Podelski. *Verification of GUI Applications: A Black-Box Approach*, pages 236–252. Springer Berlin Heidelberg, 2014.

[10] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12:161–166, 2011.

[11] Xiaoying Bai, Muyang Li, Bin Chen, Wei-Tek Tsai, and Jerry Gao. Cloud testing tools. In *IEEE 6th Int. Symp. on Service Oriented System Engineering (SOSE)*, pages 1–12, 2011.

[12] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.

[13] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *arXiv preprint arXiv:1610.00502*, 2016.

[14] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. Graphical user interface (GUI) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10):1679–1694, 2013.

[15] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[16] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Whitening SOA testing. In *Proceedings of the 7th Int. Conf. ESEC-FSE*, pages 161–170. ACM, 2009.

[17] Cesare Bartolini, Antonia Bertolino, Francesca Lonetti, and Eda Marchetti. Approaches to functional, structural and security SOA testing. In *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*, pages 381–401. IGI Global, 2012.

[18] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Ioannis Parissis. Data flow-based validation of web services compositions: Perspectives and examples. In *Architecting Dependable Systems V*, pages 298–325. Springer, 2008.

[19] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. WS-TAXI: A WSDL-based testing tool for web services. In *International Conference on Software Testing Verification and Validation (ICST)*, pages 326–335. IEEE, 2009.

[20] Fevzi Belli, Christof J Budnik, Axel Hollmann, Tugkan Tuglular, and W Eric Wong. Model-based mutation testing: approach and case studies. *Science of Computer Programming*, 120:25–48, 2016.

[21] Abdelghani Benharref, Rachida Dssouli, Mohamed Adel Serhani, and Roch Glitho. Efficient traces' collection mechanisms for passive testing of web services. *Inf. and Software Technology*, 51(2):362–374, 2009.

[22] Antonia Bertolino. Software testing. *SWEBOK*, page 69, 2001.

[23] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[24] Antonia Bertolino, Said Daoudagh, Donia El Kateb, Christopher Henard, Yves Le Traon, Francesca Lonetti, Eda Marchetti, Tejeddine Mouelhi, and Mike Papadakis. Similarity testing for access control. *Information and Software Technology*, 58:355–372, 2015.

[25] Antonia Bertolino, Said Daoudagh, Francesca Lonetti, and Eda Marchetti. The x-create framework-a comparison of xacml policy testing strategies. In *WEBIST*, pages 155–160, 2012.

[26] Antonia Bertolino, Said Daoudagh, Francesca Lonetti, and Eda Marchetti. XAMUT: XACML 2.0 mutants generator. In *IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 28–33. IEEE, 2013.

[27] Antonia Bertolino and Eda Marchetti. A brief essay on software testing. *Software Engineering, The Development Process*, 3, 2005.

[28] Paul Beynon-Davies, Chris Carne, Hugh Mackay, and Douglas Tudhope. Rapid application development (rad): an empirical review. *European Journal of Information Systems*, 8(3):211–223, 1999.

[29] A. Bhat and S. M. K. Quadri. Equivalence class partitioning and boundary value analysis - a review. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1557–1562, March 2015.

[30] Egon Börger and Robert Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Science & Business Media, 2012.

[31] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability*, 23(4):261–313, 2013.

[32] Peter Buxmann, Thomas Hess, and Sonja Lehmann. Software as a service. *Wirtschaftsinformatik*, 50(6):500–503, 2008.

[33] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[34] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Comm. of the ACM*, 56(2):82–90, 2013.

[35] Inderveer Chana, Ajay Rana, et al. Empirical evaluation of cloud-based testing techniques: a systematic review. *ACM SIGSOFT Software Engineering Notes*, 37(3):1–9, 2012.

[36] SR Dalal, MR Lyu, and CL Mallows. *Software Reliability*. Wiley Online Library, 2014.

[37] Martin D Davis and Elaine J Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM'81 Conference*, pages 254–257. ACM, 1981.

[38] Marcio Eduardo Delamaro, JC Maidonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE transactions on software engineering*, 27(3):228–247, 2001.

[39] Arilo C. Dias-Neto and Guilherme H. Travassos. A picture from the model-based testing area: Concepts, techniques, and challenges. In *Advances in Computers*, volume 80, pages 45 – 120. Elsevier, 2010.

[40] Edsger Wybe Dijkstra. Notes on structured programming, 1970.

[41] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[42] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[43] Khaled El Emam, Walcelio Melo, and Jean-Normand Drouin. *SPICE: The theory and practice of software process improvement and capability determination.* IEEE Computer Society Press, 1997.

[44] Michael Felderer, Matthias Büchler, Martin Johns, Achim D Brucker, Ruth Breu, and Alexander Pretschner. Chapter one-security testing: A survey. *Advances in Computers*, 101:1–51, 2016.

[45] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach.* CRC Press, 2014.

[46] Martin Fowler and Jim Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[47] Jerry Gao, Xiaoying Bai, and Wei-Tek Tsai. Cloud testing-issues, challenges, needs and practice. *Software Engineering: An Int. Journal*, 1(1):9–23, 2011.

[48] Jerry Gao, Xiaoying Bai, Wei-Tek Tsai, and Tadahiro Uehara. Testing as a service (TaaS) on clouds. In *IEEE 7th Int. Symp. on Service Oriented System Engineering (SOSE)*, pages 212–223. IEEE, 2013.

[49] Jerry Gao, Wei-Tek Tsai, Ray Paul, Xiaoying Bai, and Tadahiro Uehara. Mobile Testing-as-a-Service (MTaaS)–Infrastructures, Issues, Solutions and Needs. In *IEEE 15th Int. Symp. on High-Assurance Systems Engineering (HASE)*, pages 158–167. IEEE, 2014.

[50] Emily Geisen and Jennifer Romano Bergstrom. *Usability Testing for Survey Research.* Morgan Kaufmann, 2017.

[51] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proc. of the 2015 Int. Symp.on Software Testing and Analysis*, pages 211–222. ACM, 2015.

[52] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. An empirical evaluation and comparison of manual and automated test selection. In *Proc. of the 29th ACM/IEEE Int. Conf. on Automated software engineering*, pages 361–372. ACM, 2014.

[53] Dorothy Graham and Mark Fewster. *Experiences of test automation: case studies of software test automation.* Addison-Wesley Professional, 2012.

[54] John V Guttag and James J Horning. *Larch: languages and tools for formal specification.* Springer Science & Business Media, 2012.

[55] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[56] Lom Messan Hillah, Ariele-Paolo Maesano, Libero Maesano, Fabio De Rosa, Fabrice Kordon, and Pierre-Henri Wuillemin. Service functional testing automation with intelligent scheduling and planning. In *Proc. of the 31st Annual ACM Symp. on Applied Computing*, SAC '16, pages 1605–1610, New York, NY, USA, 2016. ACM.

[57] Douglas Hoffman. Cost benefits analysis of test automation. *STAR West*, 99, 1999.

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

[59] Tomas Isakowitz, Michael Bieber, and Fabio Vitali. Web information systems. *Communications of the ACM*, 41(7):78–80, 1998.

[60] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sept 2011.

[61] Z. M. Jiang and A. E. Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, Nov 2015.

[62] Abhinaya Kasoju, Kai Petersen, and Mika V Mäntylä. Analyzing an automotive testing process with evidence-based software engineering. *Information and Software Technology*, 55(7):1237 – 1259, 2013.

[63] Mohd Ehmer Khan, Farmeena Khan, et al. A comparative study of white box, black box and grey box testing techniques. *Int. Jour. of Advanced Computer Science and Applications (IJACSA)*, 3(6), 2012.

[64] B. A. Kitchenham, T. Dyba, and M. Jorgensen. Evidence-based software engineering. In *Proceedings. 26th International Conference on Software Engineering*, pages 273–281, May 2004.

[65] Henrik Kniberg. *Scrum and XP from the Trenches*. Lulu. com, 2015.

[66] Philippe Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[67] Kevin Lano. *Model-driven software development with UML and Java*. Course Technology Press, 2009.

[68] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[69] William E Lewis. *Software testing and continuous quality improvement*. CRC press, 2016.

[70] N. Li, A. Escalona, Y. Guo, and J. Offutt. A scalable big data test framework. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–2, April 2015.

[71] Pablo Loyola, Matt Staats, In-Young Ko, and Gregg Rothermel. Dodona: Automated oracle data set selection. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 193–203, New York, NY, USA, 2014. ACM.

[72] Nancy A Lynch and Mark R Tuttle. An introduction to input/output automata. 1988.

[73] Michael R Lyu et al. Handbook of software reliability engineering. 1996.

[74] Cláudio Magalhães, Flávia Barros, Alexandre Mota, and Eliot Maia. Automatic selection of test cases for regression testing. In *Proc. of the 1st Brazilian Symp. on Systematic and Automated Software Testing*, page 8. ACM, 2016.

[75] Evan Martin. Automated test generation for access control policies. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 752–753. ACM, 2006.

[76] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proc. of WWW*, pages 667–676, May 2007.

[77] Bob McFeeley. Ideal: A user's guide for software process improvement. Technical report, DTIC Document, 1996.

[78] Phil McMinn. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pages 153–163. IEEE, 2011.

[79] Phil McMinn, Mark Stevenson, and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, pages 1–4. ACM, 2010.

[80] Atif M Memon and Bao N Nguyen. Advances in automated model-based system testing of software applications with a gui front-end. *Advances in Computers*, 80:121–162, 2010.

[81] K. Michael and K. W. Miller. Big data: New opportunities and new challenges [guest editors' introduction]. *Computer*, 46(6):22–24, June 2013.

[82] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, (3):223–226, 1976.

[83] Samar Mouchawrab, Lionel C Briand, Yvan Labiche, and Massimiliano Di Penta. Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments. *IEEE Transactions on Software Engineering*, 37(2):161–187, 2011.

[84] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[85] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, February 2011.

[86] Corina S Pasareanu, Peter C Mehlitz, David H Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26. ACM, 2008.

[87] Corina S Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):339–353, 2009.

[88] Mark Paulk. Capability maturity model for software. *Encyclopedia of Software Engineering*, 1993.

[89] Christer Persson and Nur Yilmazturk. Establishment of automated regression testing at abb: Industrial experience report on'avoiding the pitfalls'. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 112–121. IEEE Computer Society, 2004.

[90] Dennis K Peters and David Lorge Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[91] Dessislava Petrova-Antonova, Kunka Kuncheva, and Sylvia Ilieva. Automatic generation of test data for xml schema-based testing of web services. In *10th International Joint Conference on Software Technologies (ICSOFT)*, volume 1, pages 1–8. IEEE, 2015.

[92] Mauro Pezze and Cheng Zhang. Automated test oracles: A survey. *Advances in Computers*, 95:1–48, 2015.

[93] Imran Ali Qureshi and Aamer Nadeem. Gui testing techniques: a survey. *International Journal of Future Computer and Communication*, 2(2):142, 2013.

[94] Rudolf Ramler and Klaus Wolfmaier. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 85–91. ACM, 2006.

[95] Christopher Robinson-Mallett. Coordinating security and safety engineering processes in automotive electronics development. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, pages 45–48, 2014.

[96] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *IEEE International Conference on Software Maintenance (ICSM '99)*, pages 179–188, 1999.

[97] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[98] David Saff and Michael D Ernst. Reducing wasted development time via continuous testing. In *14th International Symposium on Software Reliability Engineering*, pages 281–292. IEEE, 2003.

[99] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. An information retrieval approach for regression test prioritization based on program changes. In *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, volume 1, pages 268–279. IEEE, 2015.

[100] Ronnie ES Santos, Cleyton VC de Magalhães, and Fabio QB da Silva. The use of systematic reviews in evidence based software engineering: a systematic mapping study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 53. ACM, 2014.

[101] Koushik Sen. Concolic testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM.

[102] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005.

[103] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 237–247. ACM, 2015.

[104] Praveen Ranjan Srivastava. Test case prioritization. *Journal of Theoretical and Applied Information Technology*, 4(3):178–181, 2008.

[105] Oleksii Starov, Sergiy Vilkomir, Anatoliy Gorbenko, and Vyacheslav Kharchenko. Testing-as-a-service for mobile applications: state-of-the-art survey. In *Dependability Problems of Complex Information Systems*, pages 55–71. Springer, 2015.

[106] Richard H Thayer and Robin Hunter. *Software Process Improvement*. IEEE Computer Society, 2001.

[107] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra. Automating test automation. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 881–891, June 2012.

[108] Ashish Tiwari. Formal semantics and analysis methods for simulink stateflow models. Technical report, Citeseer, 2002.

[109] Jan Tretmans. Model based testing with labelled transition systems. *Formal methods and testing*, pages 1–38, 2008.

[110] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[111] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[112] I. K. Villanes, E. A. B. Costa, and A. C. Dias-Neto. Automated mobile testing as a service (am-taas). In *2015 IEEE World Congress on Services*, pages 79–86, June 2015.

[113] Manish Virmani. Understanding devops & bridging the gap from continuous integration to continuous delivery. In *Fifth International Conference on Innovative Computing Technology (INTECH)*, pages 78–82. IEEE, 2015.

[114] Kun Wu, Chunrong Fang, Zhenyu Chen, and Zhihong Zhao. Test case prioritization incorporating ordered sequence of program elements. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 124–130. IEEE Press, 2012.

[115] S Xanthakis, C Ellis, C Skourlas, A Le Gall, S Katsikas, and K Karapoulios. Application of genetic algorithms to software testing. In *Proceedings of the 5th International Conference on Software Engineering and Applications*, pages 625–636, 1992.

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

[117] Jie Zhang, Muyao Zhu, Dan Hao, and Lu Zhang. An empirical study on the scalability of selective mutation testing. In *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*, pages 277–287. IEEE, 2014.

[118] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. Time-aware test-case prioritization using integer linear programming. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 213–224. ACM, 2009.

[119] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.