A Brief Essay on Software Testing

Antonia Bertolino, Eda Marchetti

Abstract— Testing is an important and critical part of the software development process, on which the quality and reliability of the delivered product strictly depend. Testing is not limited to the detection of "bugs" in the software, but also increases confidence in its proper functioning and assists with the evaluation of functional and nonfunctional properties. Testing related activities encompass the entire development process and may consume a large part of the effort required for producing software. In this chapter we provide a comprehensive overview of software testing, from its definition to its organization, from test levels to test techniques, from test execution to the analysis of test cases effectiveness. Emphasis is more on breadth than depth: due to the vastness of the topic, in the attempt to be all-embracing, for each covered subject we can only provide a brief description and references useful for further reading.

Index Terms — D.2.4 Software/Program Verification, D.2.5 Testing and Debugging.

1. Introduction

Testing is a crucial part of the software life cycle, and recent trends in software engineering evidence the importance of this activity all along the development process. Testing activities have to start already at the requirements specification stage, with ahead planning of test strategies and procedures, and propagate down, with derivation and refinement of test cases, all along the various development steps since the code-level stage, at which the test cases are eventually executed, and even after deployment, with logging and analysis of operational usage data and customer's reported failures.

Testing is a challenging activity that involves several high-demanding tasks: at the forefront is the task of deriving an adequate suite of test cases, according to a feasible and cost-effective test selection technique. However, test selection is just a starting point, and many other critical tasks face test practitioners with technical and conceptual difficulties (which are certainly under-represented in the literature): the ability to launch the selected tests (in a controlled host environment, or worse in the tight target environment of an embedded system); deciding whether the test outcome is acceptable or not (which is referred to as the test oracle problem); if not, evaluating the impact of the failure and finding its direct cause (the fault), and the indirect one (via Root Cause Analysis); judging whether testing is sufficient and can be stopped, which in turn would require having at hand measures of the effectiveness of the tests: one by one, each of these tasks presents tough challenges to testers, for which their skill and expertise always remains of topmost importance.

We provide here a short, yet comprehensive overview of the testing discipline, spanning over test levels, test techniques and test activities. In an attempt to cover all testing related issues, we can only briefly expand on each argument, however plenty of references are also provided throughout for further reading. The remainder of the chapter is organized as follows: we present some basic concepts in Section 2, and the different types of test (static and dynamic) with the objectives characterizing the testing activity in Section 3. In Section 4 we focus on the test levels (unit, integration and system test) and in Section 5 we present the techniques used for test selection. Going on, test design, execution, documentation,d management are described in Sections 6, 7, 8 and 9, respectively. Test measurement issues are discussed in Section 10 and finally the chapter conclusions are drawn in Section 11.

2. Terminology and basic concepts

Before deepening into testing techniques, we provide here some introductory notions relative to testing terminology and basic concepts.

2.1 On the nature of the testing discipline

As we will see in the remainder of this chapter, there exist many types of testing and many test strategies, however all of them share a same ultimate purpose: increasing the software engineer confidence in the proper functioning of the software.

Towards this general goal, a piece of software can be tested to achieve various more direct objectives, all meant in fact to increase confidence, such as exposing potential design flaws or deviations from user's requirements, measuring the operational reliability, evaluating the performance characteristics, and so on (we further expand on test objectives in Section 3.3); to serve each specific objective, different techniques can be adopted.

Generally speaking, test techniques can be divided into two classes:

- Static analysis techniques (expanded in Section 3.1), where the term "static" does not refer to the techniques themselves (they can use automated analysis tools), but

- *Antonia Bertolino is with the Istituto di Scienza e Tecnologie "A. Faedo" Area della ricerca CRD di Pisa, Via Moruzzi 1, 56124 Pisa Italy. E-mail: antonia.bertolino@isti.cnr.it.*
- *Eda Marchetti is with the Istituto di Scienza e Tecnologie "A. Faedo" Area della ricerca CRD di Pisa, Via Moruzzi 1, 56124 Pisa Italy. E-mail: eda.marchetti@isti.cnr.it.*

is used to mean that they do not involve the execution of the tested system. Static techniques are applicable throughout the lifecycle to the various developed artifacts for different purposes, such as to check the adherence of the implementation to the specifications or to detect flaws in the code via inspection or review.

- Dynamic analysis techniques (further discussed in Section 3.2), which exercise the software in order to expose possible failures. The behavioral and performance properties of the program are also observed.

Static and dynamic analyses are complementary techniques [1]: the former yield generally valid results, but they may be weak in precision; the latter are efficient and provide more precise results, but only holding for the examined executions. The focus of this chapter will be mainly on dynamic test techniques, and where not otherwise specified testing is used as a synonymous for "dynamic testing".

Unfortunately, there are few mathematical certainties on which software testing foundations can lay. The firmest one, as everybody now recognizes, is that, even after successful completion of an extensive testing campaign, the software can still contain faults. As firstly stated by Dijkstra as early as thirty years ago [22], testing can never prove the absence of defects, it can only possibly reveal the presence of faults by provoking malfunctions. In the elapsed decades, lot of progress has been made both in our knowledge of how to scrutinize a program's executions in rigorous and systematic ways, and in the development of tools and processes that can support the tester's tasks.

Yet, the more the discipline progresses, the clearer it becomes that it is only by means of *rigorous empirical studies* that software testing can increase its maturity level [35]. Testing is in fact an engineering discipline, and as such it calls for evidences and proven facts, to be collected either from experience or from controlled experiments, and currently lacking, based on which testers can make predictions and take decisions.

2.2 A general definition

Testing can refer to many different activities used to check a piece of software. As said, we focus primarily on "dynamic" software testing presupposing code execution, for which we re-propose the following general definition introduced in [9]:

Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior.

This short definition attempts to include all essential testing concerns: the term *dynamic* means, as said, that testing implies executing the program on (valued) inputs; *finite* indicates that only a limited number of test cases can be executed during the testing phase, chosen from the whole test set, that can generally be considered infinite; *selected* refers to the test techniques adopted for selecting the test cases (and testers must be aware that different selection criteria may yield vastly different effectiveness); *expected* points out to the decision process adopted for establishing whether the observed outcomes of program execution are acceptable or not.

2.3 Fault vs. Failure

To fully understand the facets of software testing, it is important to clarify the terms "fault", "error"[1] and "failure": indeed, although their meanings are strictly related, there are important distinctions between these three concepts.

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, e.g., a missing or incorrect piece of code, is a *fault*. A fault may remain undetected long time, until some event activates it. When this happens, it first brings the program into an intermediate unstable state, called *error*, which, if and when propagates to the output, eventually causes the failure. The process of failure manifestation can be therefore summed up into a chain [42]:

$$\text{Fault} \rightarrow \text{Error} \rightarrow \text{Failure}$$

which can recursively iterate: a fault in turn can be caused by the failure of some other interacting system.

In any case what testing reveals are the failures and a consequent analysis stage is needed to identify the faults that caused them.

The notion of a fault however is ambiguous and difficult to grasp, because no precise criteria exist to definitively determine the cause of an observed failure. It would be preferable to speak about *failure-causing inputs*, that is, those sets of inputs that when exercised can result into a failure.

2.4 The notion of software reliability

Indeed, whether few or many, some faults will inevitably escape testing and debugging. However, a fault can be more or less disturbing depending on whether, and how frequently, it will eventually show up to the final user (and depending of course on the seriousness of its consequences).

So, in the end, one measure which is important in deciding whether a software product is ready for release is its reliability. Strictly speaking, *software reliability* is a probabilistic estimate, and measures the probability that the software will execute without failure in a *given environment* for a *given period of time* [44]. Thus, the value of software reliability depends on how frequently those inputs that cause a failure will be exercised by the final users.

Estimates of software reliability can be produced via testing. To this purpose, since the notion of reliability is specific to "a given environment", the tests must be drawn from an input distribution that approximates as closely as possible the future usage in operation, which is called the *operational distribution*.

[1] Note that we are using the term "error" with the commonly used meaning within the Software Dependability community [42], which is stricter than its general definition in [28].

3. TYPES OF TESTS

The one term *testing* actually refers to a full range of test techniques, even quite different from one other, and embraces a variety of aims.

3.1 Static Techniques

As said, a coarse distinction can be made between dynamic and static techniques, depending on whether the software is executed or not. Static techniques are based solely 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 development, and their earlier usage is of course highly desirable. Considering a generic development process, they can be applied [49]:

- 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 [7], [50]:

- *Software inspection*: the step-by-step analysis of the documents (deliverables) produced, against a compiled checklist of common and historical defects.
- *Software reviews*: 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*: the desktop analysis of the produced code for discovering typing errors that do not violate style or syntax.
- *Algorithm analysis and tracing*: 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 [19]. 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 fact, if the language used for specification has a well-defined semantics, algorithms and tools can be developed to analyze the statements written in that language.

The basic idea of using a formal language for modeling requirements or design is now universally recognized as a foundation for software verification. *Formal verification* techniques are attracting today quite a lot attention from both research institutions and industries and it is foreseeable that proofs of correctness will be increasingly applied, especially for the verification of critical systems.

One of the most promising approaches for formal verification is *model checking* [18]. Essentially, a model checking tool takes in input a *model* (a description of system functional requirements or design) and a *property* that the system is expected to satisfy.

In the middle between static and dynamic analysis techniques, is *symbolic execution* [38], which executes a program by replacing variables with symbolic values.

Quite recently, the automated generation of test data for coverage testing is again attracting lot of interest, and advanced tools are being developed based on a similar approach to symbolic execution exploiting *constraint solving* techniques [3]. A flowgraph path to be covered is translated into a path constraint, whose solution provides the desired input data.

We conclude this section considering the alternative application of static techniques in producing values of interest for controlling and managing the testing process. Different estimations can be obtained by observing specific properties of the present or past products, and/or parameters of the development process..

3.2 Dynamic Techniques

Dynamic techniques [1] obtain information of interest about a program by observing some executions. Standard dynamic analyses 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. Profiling tools are increasingly used today to derive measures of coverage, for instance in order to dynamically identify control flow invariants, as well as measures of frequency, called *spectra*, which are diagrams providing the relative execution frequencies of the monitored entities. In particular, *path spectra* refer to the distribution of (loop-free) paths traversed during program profiling. Specific dynamic techniques also include simulation, sizing and timing analysis, and prototyping [49].

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's 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 5.

In the case of concurrent, non-deterministic systems, the results obtained by testing depend not only on the input provided but also on the state of the system. Therefore, when speaking about test input values, it is implied that the definition of the parameters and environmental conditions that characterize a system state must be included when necessary.

Once the tests are selected and run, another crucial aspect of this phase is the so-called oracle problem, which means deciding whether the observed outcomes are acceptable or not (see Section 7.2).

3.3 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:

- **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's requirement. Generally, it is run by or with the end-users to perform those functions and tasks the software was built for [51].
- **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 [51].
- **Alpha testing:** before releasing the system, it is deployed to some in-house users for exploring the functions and business tasks. Generally there is no test plan to follow, but the individual tester determines what to do [36].
- **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 [36].
- **Reliability achievement:** as said in Section 2.4, testing can also be 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 [44].
- **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 tests targets, including units, integrated units, and systems [50].
- **Regression testing**: According to [28], *regression testing* is the "selective retesting of a system or component to verify that modifications have not caused unintended effects and that the syustem or component still complies with its specified requirements]". In practice, the objective is to show that a system which previously passed the tests still does [51]. 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.
- **Performance testing**: this is specifically aimed at verifying that the system meets the specified performance requirements, for instance, capacity and response time [51].
- **Usability testing**: this important testing activity 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 [51].
- **Test-driven development**: test-driven development is not a test technique per se, but promotes the use of test case specifications as a surrogate for a requirements document rather than as an independent check that the software has correctly implemented the requirements [6].

4. 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* and *system test* [7], [51]. These are the three testing stages of a traditional phased process (such as the classical waterfall). However, even considering different, more modern, process models, a distinction between these three 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.

4.1 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's purpose is to ensure that the unit satisfies its functional specification and/or that its implemented structure matches the intended design structure [7], [51].

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. A good reference for unit test is [30].

4.2 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 [7]. 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) [34]. 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) [51].

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[34]. In this case the test is focused on those classes used in reply to a particular input or system event (thread-based testing) [34]; 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 [40], or even the dynamic relations of inheritance and polymorphism [41]. Such proposals are interesting when the number of classes is not too big; however, test planning in those approaches can begin only at a mature stage of design, when the classes and their relationships are already stable.

A different branch of the literature is testing based on the *Software Architecture*: this specifies the high level, formal specification of a system structure in components and their connectors, as well as the system dynamics. The way in which the description of the Software Architecture could be used to drive the integration test plan is currently under investigation, e.g., [45].

4.3 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 [13]:

- discovering the failures that manifest themselves only at system level and hence were not detected during unit or integration testing;
- increasing the confidence that the developed product correctly implements the required capabilities;
- collecting information useful for deciding the release of the product.

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

System testing makes available information about the actual status of development that other verification techniques such as review or inspections on models and code cannot provide.

Generally system testing includes testing for performance, security, reliability, stress testing and recovery [34], [51]. 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 [44].

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.

4.4 Regression Test

Properly speaking, *regression test* is not a separate level of testing (we listed it in fact among test objectives in Section 3.3.), but may refer to the retesting of a unit, a combination of components or a whole system (see Fig. 1 below) after modification, in order to ascertain that the change has not introduced new faults [51].

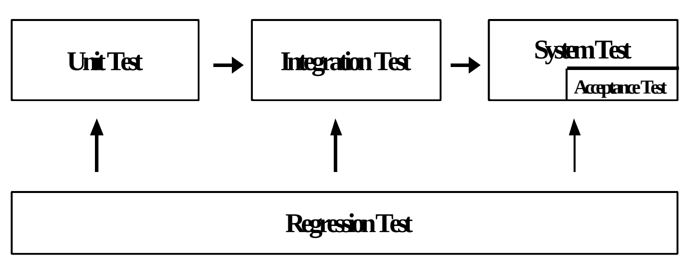

Fig. 1. Logical schema of software testing levels

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. Therefore various types of techniques have been developed to reduce regression testing costs and to make it more effective.

Selective regression test techniques [53] 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.

5. STRATEGIES FOR TEST CASE SELECTION

Effective testing requires strategies to trade-off between the two opposing needs of amplifying testing thoroughness on one side (for which a high number of test cases would be desirable) and reducing times and costs on the other (for which the fewer the test cases the better). Given that test resources are limited, how the test cases are selected becomes of crucial importance. Indeed, the problem of test cases selection has been the largely dominating topic in software testing research to the extent that in the literature "software testing" is often taken as a synonymous for "test case selection".

A decision procedure for selecting the test cases is provided by a *test criterion*.

A basic criterion is *random testing*, according to which the test inputs are picked purely randomly from the whole input domain according to a specified distribution, i.e., after assigning to the inputs different "weights" (more properly probabilities). For instance the *uniform distribution* does not make any distinction among the inputs, and any input has the same probability of being chosen. Under the operational distribution, instead, inputs are weighted according to their probability of usage in operation (as we already said in Section 2.4).

In contrast with random testing is a broad class of test criteria referred to as *partition testing*. The underlying idea is that the program input domain is divided into subdomains within which it is assumed that the program behaves the same, i.e., for every point within a subdomain the program either succeeds or fails: we also call this the "test hypothesis". Therefore, thanks to this assumption only one or few points within each subdomain need to be checked, and this is what allows for getting a finite set of tests out of the infinite domain. Hence a partition testing criterion essentially provides a way to derive the subdomains.

A test criterion yielding the assumption that all test cases within a subdomain either succeed or fail is only an ideal, and would guarantee that any fulfilling test set of test cases always detect the same failures: in practice, the assumption is rarely satisfied, and different set of test cases fulfilling a same criterion may show varying effectiveness depending on how the test cases are picked within each subdomain.

Many are the factors of relevance when a test selection criterion has to be chosen. An important point to always keep in mind is that what makes a test a "good" one does not have a unique answer, but changes depending on the context, on the specific application, and on the goal for testing. The most common interpretation for "good" would be "able to detect many failures"; but again precision would require to specify what kind of failures, as it is well known and experimentally observed that different test criteria trigger different types of faults [5], 0. Therefore, it is always preferable to spend the test budget to apply a combination of diverse techniques than concentrating it on just one, even if shown the most effective.

Paradoxically, test case selection seems to be the least interesting problem for test practitioners. A demonstration of this low interest is the paucity of commercial automated tools for helping test selection and test input generation, in comparison with a profusion of support tools (see Section 7.3) for handling test execution and re-execution (or regression test) and for test documentation. The most practiced test selection criterion in industry probably is still tester's intuition, and indeed *expert testers* may perform as very good selection "mechanisms" (with the necessary warnings against exclusively relying on such a subjective strategy). Empirical investigations [5] showed in fact that tester's skill is the factor that mostly affect test effectiveness in finding failures.

5.1 Selection Criteria Based on Code

Code-based testing, also said "structural testing", or "white box" testing, has been the dominating trend in software testing research during the late 70's and the 80's. One reason is certainly that in those years in which formal approaches to specification were much less mature and pursued than now, the only *RM* formalized enough to allow for the automation of test selection or for a quantitative measurement of thoroughness was the code.

Referring to the fault-error-failure chain described in Section 2.3, the motivation to code-based testing is that potential failures can only be detected if the parts of code related to the causing faults are executed. Hence, by monitoring code coverage one tries to exercise thoroughly all "program elements": depending on how the program elements to be covered are identified several test criteria exist.

In structural testing, the program is modelled as a graph, whose entry-exit paths represent the flow of control, hence it is called a *flowgraph*. Finding a set of flowgraph paths fulfilling a coverage criterion thus becomes a matter of properly visiting the graph (see for instance [11]). Code coverage criteria are also referred to as path-based test criteria, because they map each test input to a unique path p on the flowgraph.

The ideal and yet unreachable target of code-based testing would be the exhaustive coverage of all possible paths along the program control-flow. The underlying test hypothesis here is that by executing a path once, potential faults related to it will be revealed, i.e., it is assumed that every input executing a same path will either fail or succeed (which is not necessarily true, of course).

Full path coverage is not applicable, because banally every program with unbounded loops would yield an infinite number of paths. Even limiting the number of iterations within program loops, which is the usually practised tactic in testing, the number of tests would remain infeasibly high. Therefore, all the proposed code-based criteria attempt to realize cost/effective approximations to path coverage, by identifying specific (control-flow or data-flow) elements of a program that are deemed to be relevant for revealing possible failures, and by requiring that enough test cases to cover all such elements be executed.

The landmark paper in code-based testing is [52], in which a family of criteria was introduced, based on both control-flow and data-flow. A *subsumption* hierarchy between the criteria was derived, based on the inclusion relation such that a test suite satisfying the subsuming criterion is guaranteed to also satisfy the (transitively) subsumed criterion.

Statement coverage is the most elementary criterion, requiring that each statement in a program be exercised at least once. The already mentioned *branch coverage* criterion instead requires that each branch in a program be exercised (in other words, for every predicate its evaluation to true and false should both be tested at least once). Note that complete statement coverage does not assure that all branches are exercised (empty branches would be left out).

Branch coverage is also said "decision coverage", because it considers the outcome of a decision predicate. When a predicate is composed by the logical combination of several conditions, a variation to branch coverage is given by "condition coverage", which requires instead to test the true and false outcome of the individual conditions of predicates. Further criteria consider together coverage of decisions and conditions under differing assumptions (see, e.g., [25]).

It must be kept in mind, however, that code-based test selection is a tautology: it looks for potential problems in a program by using the program itself as a reference model. In this way, for instance, faults of missing functionalities could never be found.

As a consequence, code-based criteria should be more properly used as adequacy criteria. In other terms, testers should take the measures of coverage reached by the executed tests and the signaling of uncovered elements as a warning that the set of test cases are ignoring some parts (and which ones) of the functionalities or of the design. Coverage of unexercised elements should hence be taken as an advice for more thought and not as the compelling test target.

A sensible approach is to use another artifact as the reference model from which the test cases are designed and monitor a measure of coverage while tests are executed, so to evaluate the thoroughness of the test suite. If some elements of the code remain uncovered, additional tests to exercise them should be found, as it can be a signal that the tests do not address some function that is coded.

A final warning is worth that "exercised" and "tested" are not synonymous: an element is really tested only when its execution produces an effect on the output; in view of this statement, under most existing code-based criteria even 100% coverage could leave some statement untested.

5.2 Selection Criteria Based on Specifications

In specification-based testing, the reference model *RM* is derived in general from the documentation relative to program specifications. Depending on how the latter are expressed, largely different techniques are possible [34]. Early approaches [46] looked at the Input/Output relation of the program seen as a "black-box" and manually derived:

- *equivalence classes*: by partitioning the input domain into subdomains of "equivalent" inputs, in the sense explained in Section 5 that any input within a subdomain can be taken as a representative for the whole subset. Hence, each input condition must be separately considered to first identify the equivalence classes. The second step consists of choosing the test inputs representative of each subdomain; it is good practice to take both valid and invalid equivalence classes for each conditions. The Category Partition method that we describe below in this section belongs to this approach.
- *boundary conditions*: i.e., those combinations of values that are "close" (actually on, above and beneath) the borders of the equivalence classes identified both in the input and the output domains. This test approach is based on the intuitive fact, also proved by experience, that faults are more likely to be found at the boundaries of the input and output subdomains.
- *cause-effect graphs*: these are combinatorial logic networks that can be used to explore in systematic way the possible combinations of input conditions. By analysing the specification, the relevant input conditions, or *causes*, and the consequent transformations and output conditions, the *effects*, are identified and modelled into graphs linking the effects to their causes. A detailed description of this early technique can be found in [46].

Approaches such as the ones described above all require a degree of "creativity" [46]. To make testing more repeatable, lot of researchers have tried to automatize the derivation of test cases from formal or semiformal specifications. Early attempts included algebraic specifications [8], VDM [21], and Z [26], while a more recent collection of approaches to formal testing can be found in [27].

Also in specification based testing a graph model is often derived and some coverage criterion is applied on this model. A number of methods rely on coverage of specifications modelled as a *Finite State Machine* (FSM). A review of these approaches is given in [14]. Alternatively, conformance testing can be based on *Labelled Transition Systems* (LTS) models. LTS-based testing has been the subject of extensive research [16] and a quite mature theory now exists. Given the LTS for the specification *S* and one of its possible implementations *I* (the program to be tested), various test generation algorithms have been proposed to produce sound test suites, i.e., such that programs passing the test correspond to conformant implementations according to a defined "conformance relation". An approach for the automatic, on-the-fly generation of test cases has been implemented in the Test Generation and Verification (TGV) [54] tool.

As expectable, specification-based testing nowadays focuses on testing from UML models. A spectrum of approaches has been and is being developed, ranging from strictly formal testing approaches based on UML statecharts [43], to approaches trying to overcome UML limitations requiring OCL (Object Constraint Language) [55] additional annotations [15], to pragmatic approaches using the design documentation as is and proposing automated support tools [4]. The recent tool Agedis [24] supports the model-driven generation and execution of UML-based test suites, built on the above mentioned TGV technology.

5.3 Other Criteria

Specification-based and code-based test techniques are often contrasted as functional vs. structural testing. These two approaches to test selection are not to be seen as alternative, but rather as complementary; in fact, they use different sources of information, and have proved to highlight different kinds of problems. They should be used in combination, depending on budgetary considerations [34]. Moreover, beyond code or specifications, the derivation of test cases can be done starting from other informative sources. Some other important strategies for test selection are briefly overviewed below.

- *Based on tester's intuition and experience*

As said, one of the most widely practiced technique based on the tester intuition and experience is *ad-hoc testing* [36] techniques in which tests are derived relying on the tester's skill, intuition, and experience with similar programs. Ad hoc testing might be useful for identifying special tests, those not easily captured by formalized techniques. Another emerging technology is *Exploratory testing* [37], which is defined as simultaneous learning, test design, and test execution; that is, the tests are not defined in advance in an established test plan, but are dynamically designed, executed, and modified. The effectiveness of exploratory testing relies on the tester's knowledge, which can be derived from various sources: observed product behavior during testing, familiarity with the application, the platform, the failure process, the type of possible bugs, the risk associated with a particular product, and so on.

- *Fault-based*

With different degrees of formalization, fault-based testing techniques devise test cases specifically aimed at revealing categories of likely or pre-defined faults. In particular it is possible that the *RM* is given by expected or hypothesized faults, such as in error guessing , or mutation testing. Specifically in error guessing [36] test cases are designed by testers trying to figure out the most plausible faults in a given program. A good source of information is the history of faults discovered in earlier projects, as well as the tester's expertise. In Mutation testing [50], 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 generated mutants: If a test case is successful in identifying the difference between the program and a mutant, the latter is said to be killed. The underlying assumption of mutation testing, 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.

- *Based on operational usage*

In testing for reliability evaluation, the test environment must reproduce the operational environment of the software as closely as possible (operational profile) [34], [44], [51]. The idea is to infer, from the observed test results, the future reliability of the software when in actual use. To do this, inputs are assigned a probability distribution, or profile, according to their occurrence in actual operation. In particular the Software Reliability Engineered Testing (SRET) [44] is a testing methodology encompassing the whole development process, whereby testing is "designed and guided by reliability objectives and expected relative usage and criticality of different functions in the field."

6. Test Design

We have seen that there exist various test objectives, many test selection strategies and differing stages of the lifecycle of a product at which testing can be applied. Before actually commencing any test derivation and execution, all these aspects must be organized into a coherent framework. Indeed, software testing itself consists of a compound process, for which different models can be adopted.

A traditional test process includes subsequent phases, namely test planning, test design, test execution and test results evaluation.

Test planning is the very first phase and outlines the scope of testing activities, focusing in particular on the objectives, resources and schedule, i.e., it covers more the managerial aspects of testing, rather than the detail of techniques and the specific test cases. A test plan can be already prepared during the requirements specification phase.

Test design is a crucial phase of software testing, in which the objectives and the features to be tested and the test suites associated to each of them are defined [7], [29], [30], [51]. Also the levels of test are planned. Then, it is decided what kind of approach will be adopted at each level and for each feature to be tested. This also includes deciding a stopping rule for testing. Due to time or budget constraints, at this point it can be decided that testing will concentrate on some more critical parts.

An emerging and quite different practice for testing is test driven development, also called Test-First programming, which focuses on the derivation of (unit and acceptance) tests before coding. This approach is a key practice of modern Agile development approaches such as Extreme Programming (XP) and Rapid Application Development (RAD) [6]. The leading principle of such approaches is to make development more lightweight by keeping design simple and reducing as much as possible the rules and the activities of traditional processes felt by developers as overwhelming and unproductive, for instance devoted to documentation, formalized communication, or ahead planning of rigid milestones. Therefore a traditional test design phase as described above does no longer exist, but new tests are continuously created, as opposed to a vision of designing test suites up front. In the XP way, the leading principle is to "code a little, test a little, …" so that developers and customers can get immediate feedbacks.

7. TEST EXECUTION

Executing the test cases specified in test design may entail various difficulties. Below we discuss the various activities implied in launching the tests, and deciding the test outcome. We also hint at tools for automating testing activities.

7.1 Launching the tests

Forcing the execution of the test cases (manually or automatically) derived according to one of the criteria presented in Section 5 might not be so obvious.

If a code-based criterion is followed, it provides us with entry-exit paths over the flowgraph that must be taken, and test inputs that execute the corresponding program paths need be found. Actually, as already said, code-based should be better used as an adequacy criterion, hence in principle we should not look for inputs ad hoc to execute the not covered entities, but rather use the coverage analysis results to understand the weaknesses in the executed test cases. However, in the cycle of testing, monitoring unexecuted elements, finding additional test cases, often conducted under pressure, finding those test cases that increase coverage can be very difficult.

If a specification-based criterion is adopted, the test cases correspond to sequences of events, which are specified at the abstraction level of the specifications; more precisely, they are labels within the signature of the adopted specification language. To derive concrete test cases, these labels must be translated into corresponding labels at code level (e.g., method invocations), and eventually into execution statements to be launched on the User Interface of the used test tool.

7.2 Test Oracles

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 [57].

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, sometimes even derived manually, 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.

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 is not thinkable. Automated oracles must then be implemented. But, of course, if we had available a mechanism that knows in advance and infallibly the correct results, it would not be necessary to develop the system under test: we could use the oracle instead! Hence the need of approximate solutions.

Different approaches can be taken [2]: assertions could be embedded into the program so to provide run-time checking capability; conditions expressly specified to be used as test oracles could be developed, in contrast with using the same specifications (i.e., written to model the system behavior and not for run-time checking); the produced execution traces could be logged and analyzed.

In some cases, the oracle can be an earlier version of the system that we are going to replace with the one under test. A particular instance of this situation is regression testing, in which the test outcome is compared with earlier version executions (which however in turn had to be judged passed or failed). Generally speaking, an oracle is derived from a specification of the expected behavior. Thus, in principle, automated derivation of test cases from specifications have the advantage that by this same task we get an abstract 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.

In view of these considerations, it should be evident that the oracle might not always judge correctly. So the notion of *coverage*[2] 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 [12], whereby a perfect oracle exhibits a 100% coverage, while a less than perfect oracle may yield different measures of accuracy.

7.3 Test Tools

Testing requires fulfilling many labor-intensive tasks, running numerous executions, and handling a great amount of information. The usage of appropriate tools can therefore alleviate the burden of clerical, tedious operations, and make them less error-prone, while increasing testing efficiency and effectiveness. Reference [33] lists suitable characteristics for testing tools used for verification and validation. In the following of this section we present a repertoire of typologies of most commonly used test tools, and refer to[7], [33], [44], [50], [51] for a more complete survey.

[2] It is just an unfortunate coincidence the usage with a quite different meaning of the same term adopted for test criteria.

- *Test harness* (drivers, stubs): provides a controlled environment in which tests can be launched and the test outputs can be logged. In order to execute parts of a system, drivers and stubs are provided to simulate caller and called modules, respectively;
- *Test generators*: provide assistance in the generation of tests. The generation can be random, pathwise (based on the flowgraph) or functional (based on the formal specifications);
- *Capture/Replay*: this tool automatically re-executes, or replays, previously run tests, of which it recorded inputs and outputs (e.g., screens).
- *Oracle/file comparators/assertion checking*: these kinds of tools assist in deciding whether a test outcome is successful or faulty;
- *Coverage analyzer/Instrumenter*: a coverage analyzer assesses which and how many entities of the program flowgraph have been exercised amongst all those required by the selected coverage testing criterion. The analysis can be done thanks to program instrumenters, that insert probes into the code.
- *Tracers:* trace the history of execution of a program;
- *Reliability evaluation tools*: support test results analysis and graphical visualization in order to assess reliability related measures according to selected models.

8. TEST DOCUMENTATION

Documentation is an integral part of the formalization of the test process, which contributes to the coordination and control of the testing phase. Several types of documents may be associated to the testing activities [51], [29]: Test Plan, Test Design Specification, Test Case Specification, Test Procedure Specification, Test Log, and Test Incident or Problem Report. We outline a brief description of each of them, referring to IEEE Standard for Software Test Documentation [29] for a complete description of test documents and of their relationship with one another and with the testing process.

Test Plan: defines test items, features to be or not to be tested, approach to be followed (activities, techniques and tool to be used), pass/fail criteria, the delivered documents, task to be performed during the testing phase, environmental needs, (hardware, communication and software facilities), people and staff responsible for managing designing, preparing, executing the tasks, staffing needs, schedule (including milestones, estimation of time required to do each task, period of use of each testing resources).

Test Design Specification: describes the features to be tested and their associated test set.

Test Case Specification: defines the input/output required for executing and a test case as well as any special constraints or intercase dependencies. A skeleton is depicted in Fig. 2.

Test Case Specification	
Test case ID	The unique identifier associated with the test case
Test items and purpose	The items and features exercised
Input data	The explicit list of the inputs required for executing the test case (values, files database etc)
Test case behaviour	Description of the expected test case behaviour
Output data	The list of the outputs admitted for each feature involved in the test case, possibly associated with tolerance values
Environmental set-up	The hardware/software configurations required
Specific procedural reqs	The constraints and the special procedures required.
Test cases dependencies	The IDs of the test cases that must be executed prior this test case

Fig. 2. Scheme of a possible test case

Test Procedure Specification: specifies the steps and the special requirements that are necessary for executing a set of test case.

Test Log: documents the result of a test execution, including: the occurred failures (if any); the information needed for reproducing them and locating and fixing the corresponding faults; the information necessary for establishing whether the project is complete; any anomalous events. See a summary in Fig. 3.

Test Incident or Problem Report: provides a description of the incidents including inputs, expected and obtained results, anomalies, date and time, procedure steps, environment, attempts to repeat the tests, observations and reference to the test case and procedure specification and test log.

Test Log	
Test log ID	The unique identifier associated with the test log
Items tested	Details of the items tested including environmental attributes
Events	the list of the events occurred including: the start and end date and time of each event ID of the test procedures executed personnel who executed the procedures description of test procedures results environmental details Description of the anomalous events occurred

Fig. 3. Scheme of a possible test log

9. TEST MANAGEMENT

The management processes for software development concern different activities mainly summarized into [32]: initiation and scope definition, planning, execution and control, review and evaluation, closure. These activities also concern the management of the test process even though with some specific characterizations.

In the testing phase in fact a very important component of successful testing is a collaborative attitude towards testing and quality assurance activities. Managers have a key role in fostering a generally favorable reception towards failure discovery during development; for instance, by preventing a mindset of code ownership among programmers, so that they will not feel responsible for failures revealed by their code. Moreover the testing phases could be guided by various aims, for example: in risk-based testing, which uses the product risks to prioritize and focus the test strategy; or in scenario-based testing, in which test cases are defined based on specified system scenarios.

Test management can be conducted at different levels therefore it must be organized, together with people, tools, policies, and measurements, into a well-defined process which is an integral part to the life cycle[3].

In the testing context the main manager's activities can be summarized as [[7], [36], [50], [51]:

- Scheduling the timely completion of tasks
- Estimation of the effort and the resources needed to execute the tasks: An important task in test planning is the estimation of resources required which means organizing not only hardware and software tools but also people. Thus the formalization of the test process also requires putting together a test team, which can involve internal as well as external staff members. The decision will be determined by consideration of costs, schedule, maturity level of the involved organization and the criticality of the application.
- Quantification of the risk associated with the tasks
- Effort/Cost estimation**:** The testing phase is a critical step in process development, often responsible for the high costs and effort required for product release. The effort can be evaluated for example in terms of person-days, months or years necessary for the realization of each project. For cost estimation it is possible to use two kinds of models: static and dynamic multivariate models. The former use historical data to derive empirical relationships, the latter project resource requirements as a function of time. In particular, these test measures can be related to the number of tests executed or the number of tests failed. Finally to carry out testing or maintenance in an organized and cost/effective way, the means used to test each part of the system should be reused systematically. This repository of test materials must be configuration-controlled, so that changes to system requirements or design can be reflected in changes to the scope of the tests conducted. The test solutions adopted for testing some application types under certain circumstances, with the motivations behind the decisions taken, form a test pattern which can itself be documented for later reuse in similar projects.
- Quality control measures to be employed: several measures relative to the resources spent on testing, as well as to the relative fault-finding effectiveness of the various test phases, are used by managers to control and improve the test process. These test measures may cover such aspects as: number of test cases specified, number of test cases executed, number of test cases passed, and number of test cases failed, among others. Evaluation of test problem reports can be combined with root-cause analysis to evaluate test process effectiveness in finding faults as early as possible. Such an evaluation could be associated with the analysis of risks. Moreover, the resources that are worth spending on testing should be commensurate with the use/criticality of the application: specifically a decision must be made as to how much testing is enough and when a test stage can be terminated. Thoroughness measures, such as achieved code coverage or functional completeness, as well as estimates of fault density or of operational reliability, provide useful support, but are not sufficient in themselves. The decision also involves considerations about the costs and risks incurred by potential for remaining failures, as opposed to the costs implied by continuing to test. We detail better this topic in the next section.

[3] In [32], testing is not described as a stand-alone process, but principles for testing activities are included along with both the five primary life cycle processes, and the supporting process. In [31], testing is grouped with other evaluation activities as integral to development throughout the lifecycle.

10. TEST MEASUREMENTS

Measurements are nowadays applied in every scientific field for quantitatively evaluating parameters of interest, understanding the effectiveness of techniques or tools, the productivity of development activities (such as testing or configuration management), the quality of products, and more. In particular, in the software engineering context they are used for generating quantitative descriptions of key processes and products, and consequently controlling software behavior and results. But these are not the only reasons for using measurement; it can permit definition of a baseline for understanding the nature and impact of proposed changes. Moreover, as seen in the previous section, measurement allows managers and developers to monitor the effects of activities and changes on all aspects of development. In this way actions to check whether the final outcome differs significantly from plans can be taken as early as possible[23].

We have already hinted at useful test measures throughout the chapter. It can be useful to briefly summarize them altogether. Considering the testing phase, measurement can be applied to evaluate the program under test, or the selected test set, or even for monitoring the testing process itself [9].

10.1 Evaluation of the Program Under Test

For evaluating the program under test the following measurements can be applied:

Program measurement to aid in test planning and design: considering the program under test, three different categories of measurement can be applied as reported in [7]:

- 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), the statements, the number of unique operands or operators, and the 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 [50]..

Life testing, reliability evaluation: By applying the operational testing for a specific product it is possible either to evaluate its reliability and decide if testing can be stopped or to achieve an established level of reliability. In particular *Reliability Growth* models can be used for predicting the product reliability[44].

10.2 Evaluation of the Test Performed

For evaluating the set of test cases implemented the following measures can be applied:

Coverage/thoroughness measure: Some adequacy criteria require exercising a set of elements identified in the program or in the specification by testing.

Effectiveness: In general a notion of *effectiveness* must be associated with a test case or an entire test suite, but test effectiveness does not yield a universal interpretation.

10.3 Measures for monitoring the testing process

We have already mentioned that one intuitive and diffuse practice is to count the number of failures or faults detected. The test criterion that found the highest number could be deemed the most useful. Even this measure has drawbacks: as tests are gathered and more and more faults are removed, what can we infer about the resulting quality of the tested program? for instance, if we continue testing and no new faults are found for a while, what does this imply? that the program is "correct", or that the tests are ineffective?

It is possible that several different failures are caused by a single fault, as well as that a same failure is caused by different faults. What should be better estimated then in a program, its number of contained "faults" or how many "failures" it exposed? Either estimate taken alone can be tricky: if failures are counted it is possible to end up the testing with a pessimistic estimate of program "integrity", as one fault may produce multiple failures. On the other hand, if faults are considered, we could evaluate at the same level harmful faults that produce frequent failures, and inoffensive faults that would remain hidden for years of operation. It is hence clear that the two estimates are both important during development and are produced by different (complementary) types of analysis.

The most objective measure is a statistical one: if the executed tests can be taken as a representative sample of program behavior, than we can make a statistical prediction of what would happen for the next tests, should we continue to use the program in the same way. This reasoning is at the basis of software reliability.

Documentation and analysis of test results require discipline and effort, but form an important resource of a company for product maintenance and for improving future projects.

11. Conclusions

We have presented a comprehensive overview of software testing concepts, techniques and processes. In compiling the survey we have tried to be comprehensive to the best of our knowledge, as matured in years of research and study of this fascinating topic The approaches overviewed include more traditional techniques, e.g., code-based criteria, as well as more modern ones, such as model checking or the recent XP approach.

Two are the main contributions we intended to offer to the readers: on one side, by putting into a coherent framework all the many topics and tasks concerning the software testing discipline, we hope to have demonstrated that software testing is a very complex activity deserving a first-class role in software development, in terms of both resources and intellectual requirements. On the other side, by hinting at relevant issues and open questions, we hope to attract further interest from academy and industry in contributing to evolve the state of the art on the many still remaining open issues.

In the years, software testing has evolved from an "art" [46] to an engineering discipline, as the standards, techniques and tools cited throughout the chapter demonstrate. However test practice inherently still remains a trial-and-error methodology. We will never find a test approach that is guaranteed to deliver a "perfect" product, whichever is the effort we employ. However, what we can and must pursue is to transform testing from "trial-and-error" to a systematic, cost-effective and predictable engineering discipline.

References

[1] T. Ball, "The concept of dynamic analysis", *Proc.of joint 7th ESEC/7th ACM FSE,* Toulouse, France, vol.24, no. 6, October 1999, pp.: 216 – 234.

[2] L. Baresi, and M. Young, "Test Oracles" Tech. Report CIS-TR-01-02.http://www.cs.uoregon.edu/~michal/pubs/oracles.html

[3] R. Barták, "On-line Guide to Constraint Programming", Prague, http://kti.mff.cuni.cz/~bartak/constraints/, 1998,

[4] F. Basanieri, A.. Bertolino, and E.Marchetti, "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects", *Proc. 5th Int. Conf. UML 2002*, Dresden, Germany, LNCS 2460, pp. 383--397, 2002.

[5] V.R. Basili, and R.W. Selby, R.W. "Comparing the Effectiveness of Software Testing Strategies", *IEEE Trans. Software Eng*. Vol. 13, no. 12, pp. 1278—1296 1987.

[6] K. Beck *Test-Driven Development by Example,* Addison Wesley, November 2002

[7] B. Beizer, *Software Testing Techniques* 2nd Edition, International

Thomson Computer Press, 1990.

[8] G. Bernot, M.C. Gaudel, and B. Marre, "Software Testing Based On Formal Specifications: a Theory and a Tool", *Software Eng. Journal*, vol. 6, pp. 387—405, 1991.

[9] A. Bertolino, "Knowledge Area Description of Software Testing", Chapter 5 of SWEBOK: *The Guide to the Software Engineering Body of Knowledge*. Joint IEEE-ACM Software Engineering Coordination Committee. 2001. http://www.swebok.org/.

[10] A. Bertolino, "Software Testing Research and Practice", *10th International Workshop on Abstract State Machines ASM 2003*, Taormina, Italy, , LNCS 2589, pp. 1-21. March 3-7, 2003.

[11] A. Bertolino, and M. Marré "A General Path Generation Algorithm for Coverage Testing" *Proc. 10th Int. Soft. Quality Week,* San Francisco, Ca. pap. 2T1, 1997.

[12] A. Bertolino, L. Strigini, " On the Use of Testability Measures for Dependability Assessment" *IEEE Trans. Software Eng*., vol. 22, no. 2, pp. 97-108, 1996.

[13] R.V. Binder *Testing Object-Oriented Systems - Models, Patterns, and Tools,* Addison-Wesley, 1999.

[14] G.V. Bochmann, and A. Petrenko,"Protocol Testing: Review of Methods and Relevance for Software Testing", *Proc. Int. Symp. on Soft. Testing and Analysis (ISSTA),* Seattle, pp. 109-124, 1994.

[15] L. Briand, and Y. Labiche, "A UML-Based Approach to System Testing", *Software and Systems Modeling,* vol. 1, no. 1, pp. 10-42, 2002.

[16] E. Brinksma, and J. Tretmans,, "Testing Transition Systems: An Annotated Bibliography", . *Proc. of MOVEP'2k,* Nantes pp. 44-50, 2000.

[17] R.H Carver, and K.C Tai,."Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs". *IEEE Trans. on Soft. Eng*, vol.24, no.6, pp. 471—490, 1998.

[18] E.M. Clarke, O. Grumberg, and D. A. Peled, *Model checking,* MIT Press Cambridge, MA, USA 2000

[19] E.M. Clarke and J. Wing, "Formal Methods: State of the Art and Future Directions", *ACM Computing Surveys*, vol. 28, no. 4, pp. 626-643, 1996

[20] P.d. Coward, "Symbolic Execution Systems – A Review", *Software Eng. J.* pp. 229—239, 1988.

[21] J. Dick, and A. Faivre, "Automating The Generation and Sequencing of Test Cases From Model-Based Specifications" *Proc. FME'93*, LNCS 670, pp. 268—284, 1993.

[22] E.W. Dijkstra,"Notes on Structured Programming" *T.H. Rep. 70-WSK03 1970.* http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

[23] N.E. Fenton, and S.L Pfleeger *Software Metrics - A Rigorous and Practical Approach"*. Second ed. London: International Thomson Computer Press, 1997.

[24] A. Hartman, and K. Nagin "The AGEDIS Tools for Model Based Testing" *International Symposium on Software Testing and Analysis (ISSTA 2004)*, Boston, Massachusetts July 11-14, 2004

[25] K.J. Hayhurst, D.S. Veerhusen, J.J. Chikenski, and L.K. Rierson, "A Practical Tutorial on Modified Condition/Decision Coverage", Nasa/TM-2001-210876, May2001.

[26] R.M. Hierons, "Testing from a Z Specification" *Soft. Testing, Verification and Reliability,* vol. 7, pp. 19-33, 1997.

[27] R. Hierons, J. Derrick, (Eds) "Special Issue on Specification-based Testing" *Soft. Testing, Verification and Reliability*, vol. 10, 2000.

[28] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990.

[29] IEEE Standard for Software Test Documentation, IEEE Std 829-1998.

[30] IEEE Standard for Software Unit Testing IEEE Std. 1008-1987 (R1993).

[31] IEEE Standard: Guide for Developing Software Life Cycle Processes, IEEE Std 1074-1995

[32] IEEE Standard for Information Technology-Software Life cycle processes, IEEE/EIA 12207.0-1996.

[33] Information Technology - Guideline for the evaluation and selection of CASE tools ISO/IEC 14102 1995-E

[34] P. C Jorgensen, *Software Testing a Craftsman's Approach*. CRC Press, 1995.

[35] N. Juristo, A.M. Moreno, and S. Vegas, "Reviewing 25 Years of Testing Technique Experiments", *Empirical Software. Engineering Journal,* vol. 9, no. ½, March 2004, pp. 7-44.

[36] C. Kaner, J. Falk, and H.Q. Nguyen H.Q. *Testing Computer Software*, 2nd Edition, John Wiley & Sons, April, 1999.

[37] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*, Wiley Computer Publishing 2001.

[38] J.C. King. "Symbolic execution and program testing". *Communications of the ACM*, vol.19, no. 7, 1976, pp.385–394.

[39] B. Korel, "Automated Software Test Data Generation", *IEEE Trans. Software Eng.,* vol. 16, no. 8, pp. 870—879, 1990.

[40] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y. Kim, and Y. Song, "Developing an Object-Oriented Software Testing and Maintenance Environment". *Communication of the ACM*, vol. 32, no. 10, 1995, pp.75-87.

[41] Y. Labiche, P. Thévenod-Fosse, H. Waeselynck, and M.H. Durand "Testing Level for Object-Oriented Software"; *Proceeding of ICSE, Limerick*, Ireland, June 2000, pp. 136-145.

[42] J.C. Laprie, "Dependability - Its Attributes, Impairments and-Means",*Predictably Dependable Computing Systems*, B. Randell, J.C. Laprie, H. Kopetz, B. Littlewood, eds.:, Springer , 1995

[43] D. Latella, and M. Massink "On Testing and Conformance Relations for UML Statechart Diagrams Behaviours" *Symposium on Soft.Testing and Analysis ISSTA 2002*, Roma, Italy July 2002.

[44] M.R Lyu, eds., *Handbook of Software Reliability Engineering*, McGraw-Hill, 1996.

[45] H. Muccini, A. Bertolino, P. Inverardi, "Using Software Architecture for Code Testing", *IEEE Transactions on Software Engineering*, vol. 30, no. 3, March 2004, pp. 160-170

[46] G.J. Myers, *The Art of Software Testing*. Wiley 1979.

[47] T.J. Ostrand, and M.J Balcer, M.J, "The Category-Partition Method for Specifying and Generating Functional Tests", *ACM Comm,* vol. 31, no. 6, pp. 676—686, 1988.

[48] R. Pargas, M.J. Harrold, and R. Peck," Test-Data Generation Using Genetic Algorithms", *J. of Soft. Testing, Verifications, and Reliability,* vol. 9, pp. 263—282, 1999.

[49] W. W. Peng, and D.R. Wallace, "Software Error Analysis", *NIST SP 500-209, National Institute of standards and Technology, Gaithersburg MD 20899,* http://hissa.nist.gov/SWERROR/ December 1993.

[50] W. Perry, *Effective Methods for Software Testing*, Wiley 1995.

[51] S.L. Pfleeger, *Software Engineering Theory and Practice*, Prentice Hall, 2001.

[52] S. Rapps, and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng*. vol.11, pp. 367—375, 1985.

[53] G. Rothermel. and M.J. Harrold, "Analyzing Regression Test Selection Techniques", *IEEE Tmnsactions on Software Engineering*, vol. 22, no. 8, pp. 529 – 551, 1996.

[54] TGV--Test Generation from transitions systems using Verification techniques http://www.inrialpes.fr/vasy/cadp/man/tgv.html

[55] J. Warmer, and A. Kleppe O*bject Constraint Language, The: Getting Your Models Ready for MDA,* Second Edition Addison Wesley, 2003.

[56] E.J. Weyuker, "Translatability and Decidability Questions for Restricted Classes of Program Schemas" SIAM J. on Computers , vol. 8, no. 4, pp. 587—598, 1979.

[57] E.J. Weyuker " On Testing Non-testable Programs" *The Computer Journal*, vol. 25, no.4, pp. 465—470, 1982.

M. Wood, M. Roper, A. Brooks, and J. Miller, "Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study", *Proc. ESEC/FSE*, LNCS 1301, 1997.