Software Testing Research and Practice

Antonia Bertolino

ISTI-CNR, Area della Ricerca CNR di Pisa, Italy
bertolino@iei.pi.cnr.it

Abstract. The paper attempts to provide a comprehensive view of the field of software testing. The objective is to put all the relevant issues into a unified context, although admittedly the overview is biased towards my own research and expertise. In view of the vastness of the field, for each topic problems and approaches are only briefly tackled, with appropriate references provided to dive into them. I do not mean to give here a complete survey of software testing. Rather I intend to show how an unwieldy mix of theoretical and technical problems challenge software testers, and that a large gap exists between the state of the art and of the practice.

1 Foreword

It is with pleasure that I received the invitation to present my views and perspectives on Software Testing at the 10th International Workshop on Abstract State Machines. The testing of software is the research field I have been involved in for several years now, and in this talk I will argue the important role of testing in research and in practice.

On the other hand, I will not hide a little trepidation to talk in front of the somewhat exclusive ASM "club", that has been enrolling highly qualified researchers and practitioners from worldwide for almost 20 years [18]. There is an underlying motivation: whereby the ASM approach to system analysis and design preaches very precise abstract models, and a stepwise and throughout mathematically verifiable refinement from these abstract models down to executable code, in contrast testing is intrinsically an empirical, pragmatic activity.

Testing by its nature can never conclude anything mathematically valid, as it amounts to taking a sample and trying to infer a generally valid judgement on the whole from the observed part. To complicate things, when the object of testing includes software, in making the inference we cannot rely on any certain continuity property [44] as in the testing of physical systems. We can/must work towards making the sampling less ad hoc, and more systematic. We can/must develop support tools to automate the clerical activities, and to partially automate the intellectual tasks. We can/must try to incorporate quantitative, measurable notions within the analysis of test results. Yet, the fact remains that *program testing can be used to show the presence of bugs, but never to show their absence*, as incisively stated by Dijkstra [28] as far as thirty years ago.

Though, testing is a challenging activity, and can greatly contribute to the engineering of quality programs. I contrasted the formal rigor of the ASM approach to the inherent pragmatism of testing. However, the contrast is only artificial: on the contrary, it is just within a well formalized process, such as ASM's, that testing provides the highest benefits. Indeed, by looking at the constant concern on putting ASMs to use in industrial applications [18], and at efforts devoted to exploit ASM models to drive testing [42, 43], I feel confident that this talk will find in the ASM community positive reception and a fertile soil.

1.1 Paper Structure

I introduce a definition of software testing and related terminology in Sect. 2. I discuss test case selection in Sect. 3 and other relevant issues in Sect. 4. I overview testing in relation with the development process in Sect. 5. Very brief conclusions are drawn in Sect. 6.

2 Software Testing

To start with I will re-propose here the definition of Software Testing introduced in [9][1]. That definition was worked out as a reference framework allowing for both arranging the relevant issues of software testing within a unifying vision, and tracing the most active research directions.

Definition 1. *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.*

The underlined terms identify each a corresponding key concern for software testers. Let me briefly analyze them:

- dynamic: this term is meant to explicitly signify that testing implies executing the program on (valued) inputs. To be precise, the input value alone is not always sufficient to determine a test, as a system behavior generally depends on the system state (and non deterministic system might even react with different behaviors to a same input in the same state). Different from testing, and complementary with it, are static analysis techniques, such as peer review and inspection [33, 31] (sometimes improperly referred to as "static testing"); program execution on symbolic inputs, or symbolic evaluation [24]; formal verification techniques, such as theorem provers, model checkers, data-flow analyzers. Yet another practice is to apply traditional test approaches to "test" the executable specifications: this might be useful to

[1] This reference is a chapter within the Guide to the Software Engineering Body of Knowledge (SWEBOK), providing an overview and references to acquire the basic "generally accepted" notions behind the Software Testing Knowledge Area, and could serve as a starting point for testing newcomers.

early reveal potential inconsistencies in the model. All these approaches are important, but are left outside the scope of this paper. I focus here expressly on testing the implementation. Indeed, while all the above approaches are useful to evaluate the internal "correctness" of a software system, testing is the only procedure allowing for the assessment of its behavior when executed within the target environment.

- finite: for even simple programs, so many test cases are theoretically possible that exhaustive testing (i.e., exercising it on every input) would require even years to execute (e.g., Dijkstra [28] calculated that the exhaustive testing of a multiplier of two 27-bit integers taking "only" -at the time- some tens of microseconds for a single multiplication would require more than 10000 years). Generally, the whole test set can be considered infinite. In contrast, the number of executions that can realistically be observed must obviously be finite (and affordable). Clearly, "enough" testing to get reasonable assurance of acceptable behavior should be performed. This basic need points to well known problems of testing, both technical in nature (criteria for deciding test adequacy) and managerial in nature (estimating the effort to put in testing). Testing always implies a trade-off between limited resources and schedules, and inherently unlimited test requirements;
- selected: the many proposed *test criteria* essentially differ in how they select the (finite) test suite. Testers should be constantly aware that different techniques may yield largely different effect, also depending on the context (kind of application, maturity of the process and the organization, expertise of testers, tool support are among the influencing factors). How to identify the most suitable selection criterion under given conditions is a very complex problem [76]; in practice risk analysis techniques and test engineering expertise are applied;
- expected: it must be possible (although not always easy) to decide whether the observed outcomes of program execution are acceptable or not, otherwise the testing would be useless. The observed behavior may be checked against user's expectations (commonly referred to as *testing for validation*) or against a specification (*testing for verification*, also said conformance testing). The test pass/fail decision is commonly referred in the testing literature to as the *oracle problem*.

2.1 Fault vs. Failure

Although strictly related, fault and failure denote different notions. When a test is executed and the oracle response is "fail", this means that the program exposed an undesired behavior: properly this is called a *failure*, and can be defined [55] as the deviation of the delivered service from the function the program was intended for.

Instead, the originating cause of the failure is said a *fault*. A fault may remain dormant long time, until it is activated and brings the program to a state which, if propagated to the observable output, can lead to a failure: this intermediate unstable state is indicated as an *error*. Hence, the chain

fault → error → failure

expresses a causality relationship, which can be iterated recursively, as a fault itself can be seen as the consequence of a failure of some other system: for instance a coding fault is a failure of the program developer.

Mechanisms of how faults propagate to failures have been extensively modelled [69, 60, 77] and empirically studied [34, 50], and even specific test techniques have been devised based on the resulting models, such as for example the weak mutation approach [51].

Notwithstanding its intuitive meaning, defining formally a fault is extremely difficult. If, as it is often done, a fault is characterized for practical purposes as "what must be changed" (in code, specifications, design, ...) to eliminate the failed behavior, we should then accept the fact that widely different fixes could be done to remove a same failure. Therefore, the term fault is ambiguous and should be avoided: in [37], the authors suggest to refer instead to "failure regions" of the input domain such that when a point within them is executed the program fails. Having said this, I will anyway refer to "fault" in the following, when it is necessary to explicitly distinguish between the effect (the failure) and its cause.

Important considerations apply for testers. By testing we can expose failures, but only subsequent off-line analysis can identify the originating faults. Such analysis in modern large system can be extremely costly, and some techniques for automating fault localization in code have been developed, e.g. [22].

Another important difference regards the evaluation of a system through the test results: counting failures or counting faults may yield quite different results[2] and implications. A historical reference in this regard is [1], in which empirical data showed that most faults are ineffective, as they would never be activated in the lifetime of a system: probably only ten per cent of design faults are worth fixing. Although quite obvious, this consideration is ignored in several studies that base their evaluation of a test selection criterion not on a measure of the potential impact of failures found (as done in [37]), but just on their number.

3 Test Selection

The problem of test cases selection has been the largely dominating topic in software testing research to the extent that "software testing" is often taken as a synonymous for "test case selection". Researchers creativity seems endless in proposing novel criteria for picking out a "good" sample of potential program behaviors. A comprehensive survey until 1997 is [87], but new criteria have been proposed since.

An important point to always keep in mind is that what makes a test a "good" one does not have a unique answer, but it changes depending on the

[2] Note that the number of faults can be significantly lower from the number of failures exposed by the tests, as a same fault can stimulate several failures, if the program is not fixed immediately.

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 [6, 86]. 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 [57].

Moreover, fault removal is only one of the two potential goals for software testing; the second one is evaluating some specified quality, e.g., reliability testing, usability testing, performance testing, and so on. Indeed, the final result of a validation process should be to achieve an objective measure of the confidence that can be put on the software being developed. Test suites that are good for one purpose, could not be as good for another.

In general, a test criterion is a means of deciding which a "good" set of test cases should be. A more precise definition is provided below.

Definition 2. *A test criterion C is a decision predicate defined on triples (P, RM, T), where P is a program, RM is a reference model related to P, and T is a test suite. When $C(P, RM, T)$ holds, it is said that T satisfies criterion C for P and RM.*

This definition refers to a broad category of test approaches referred to as "partition" testing: the adopted reference model RM induces (directly or indirectly) a partitioning of the program input domain into subdomains and it is tacitly assumed (this is the underlying test hypothesis) that for every point within a same subdomain the program either succeeds or fails. Therefore, thanks to the test hypothesis 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. The basic notion of grouping the inputs into subdomains that constitute equivalence classes for test purposes were early formalized in [85, 68].

Partition testing is contrasted with "random testing", by which instead test inputs are randomly drawn from the whole domain according to a specified distribution[3](most often the uniform or the operational distributions). The relative merits of these two different test philosophies have been highly debated (see e.g. the sequence of [30, 45, 84, 37]).

Testers can use a test criterion for guiding in proactive way the selection of test cases (so that when the selection terminates the criterion is automatically fulfilled), or for checking after the fact if the executed (and anyhow else selected) suite is sufficient. In the latter case, the criterion provides a *stopping rule* for testing, i.e., for given P and RM a test suite T satisfying $C(P, RM, T)$ is deemed to be *adequate*. For instance, a tester could execute a test suite manually derived from the analysis of the requirements specification document, and use a coverage

[3] An approach somewhere in the middle between pure partition testing and random testing is the statistical testing approach [74], by which the distribution of the inputs is designed so that the randomly drawn test cases have a high probability to satisfy a specified criterion.

analyzer tool during test execution for measuring the percentage of program branches covered, stopping the testing as soon as this percentage reaches a fixed threshold.

Test criteria can be classified according to the kind of *RM* [9]: it can be as informal as "tester expertise", or strictly formal, as in the case of conformance testing from a formal specification or also of code-coverage criteria. The advantages of a formal *RM* are evident: the selection of test cases, or otherwise adequacy evaluation, can be automatized.

Many are the factors of relevance when a test selection criterion has to be chosen. A framework within which existing test criteria can be categorized and compared with regard to their potential utilization in a context has been developed and empirically evaluated [76].

Paradoxically, test case selection seems to be on the other hand 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 for handling test execution and re-execution (or regression test) and for test documentation, both in the specification of tests and in the logging and analysis of test results. Indeed, much progress could be done in this direction, as test criteria that have been state-of-art for more than twenty years, and could greatly improve test cost/effectiveness, remain still almost unknown to practitioners (as recently discussed in [8]). The most practiced test selection criterion in industry probably is still tester's expertise, and indeed expert testers may perform as very good selection "mechanisms". Empirical investigations [6] showed in fact that tester's skill is the factor that mostly affect test effectiveness in finding failures.

3.1 Selection Criteria Based on Code

Code-based 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 hopefully allow for the automation of test selection or adequacy measurement was the code.

These criteria are also called path-based, because they map each test input to a unique path p on the flowgraph corresponding to *RM*. 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 basic 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, 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 failures, and by requiring that enough paths to cover all such elements be executed. In particular, in data flow-based testing, the graph model is annotated with information about how the program variables are defined and used, and a test is aimed at exercising how a value assigned to a variable is used along different control-flow paths.

The landmark paper in code-based testing is [67], in which a family of criteria is 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. This family hierarchy has remained still today an important reference point: whenever a new criterion is proposed, its place in this hierarchy is located. Indeed, since 1985, many other criteria have been added to the Rapps-Weyuker family (e.g., [29, 35]).

To automate path-based testing, the program is modelled as a graph and the entry-exit paths over the graph are considered. Finding a set of graph paths fulfilling a coverage criterion thus becomes a matter of properly visiting the graph (see for instance [12]). However, two though problems immediately arise:

- not all graph paths correspond to actual program paths, i.e., the applied graph visiting algorithm could select some paths which are not executable. The *unfeasible paths* problem is a crux of code-based test automation, and its incidence grows as we go up in the subsumption hierarchy.
- finding an input that executes a selected graph path is not only an undecidable problem in principle [82], but also a quite difficult one to solve in practice. Traditionally, symbolic execution [23] was attempted, with scarce practical success. More recent approaches include dynamic generation based on optimization [54] or genetic algorithms [65].

In consequence of these problems, code-based criteria should be more properly used as adequacy criteria. After all, 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. It is hence more sensible to use another artifact as the RM and measure code 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 warning is worth that "exercised" and "tested" are not synonymous, and as shown in [35] under most existing code-based criteria even 100% coverage could even leave untested statements.

3.2 Selection Criteria Based on Specifications

In specification-based testing, the RM is derived in general from the documentation relative to program specifications. Depending on how these are expressed, largely different techniques are possible. Very early approaches [62] looked at the Input/Output relation of the program "black-box" and manually derived equivalence classes, or boundary conditions, or cause-effect graphs.

Lot of researchers have tried to automatize the derivation of test cases from formal or semiformal specifications. Early attempts include algebraic specifications [7], VDM [27], and Z [48], while a more recent collection of approaches can be found in [49]. An influential work is [64], introducing the Category-Partition (CP) method for the automated generation of functional tests from annotated semi-formal specifications. CP is a simple, intuitive, yet effective method for functional testing, and since its appearance its basic principle has been applied to specifications in several languages, including recently UML [5].

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 [17].

LTS-based conformance testing has been the subject of extensive research [20]: given the LTS for the specification S and one of its possible implementation I, expressed using LTSs or Input/Output Transition Systems (IOTSs), some "Implementation relations" (imp) are defined, formalizing the notion of conformance of I with respect to S [75]. In such a formal setting, test generation algorithms must produce sound test suites, i.e., such that models passing the test correspond to conformant implementations. A related approach [53] uses Input/Output LTS (IOLTS) to formalize the specification, the implementation and the test purposes (TP). An approach for the automatic, on-the-fly generation of test cases to verify the TP has been implemented in the Test Generation and Verification (TGV) [73] tool.

As expectable, specification-based testing nowadays focuses on testing from UML models. A spectrum of approaches is being developed, ranging from strictly formal testing approaches based on UML statecharts [56], to approaches trying to overcome UML limitations requiring OCL additional annotations [19], to pragmatic approaches using the design documentation as is and proposing automated support tools [5]. The currently ongoing Agedis [2] project aims at developing a UML based test tool, based on the already cited TGV environment.

3.3 Other Criteria

Several other test criteria have been proposed, but size limitations do not allow me to tackle them in detail. I only cite that the RM could be given by expected or hypothesized faults, such as in Error guessing [62], or Mutation testing [26]. Another criterion is operational testing [59], in which RM is how the users will exercise a system (see also Section 4.4).

4 Selecting the Test Cases is not the Only Issue

Having derived a test suite is only one piece of the testing puzzle and more complex pieces have to be put in place before the picture is complete. Other test related activities present technical and conceptual difficulties that are underrepresented in research (but well present to practitioners): 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; if not, evaluating the impact of the failure and finding its direct cause (the fault) and indirect one (Root Cause Analysis); judging whether the test campaign is sufficient, which in turn would require having at hand effectiveness measures of the tests: one by one all these tasks present though challenges.

These activities have received marginal attention in software testing research. One argument is that being these issues technological in kind, in contrast with the more theoretical and intuitive problem of test selection, the approaches pursued are specific to an application context, and are less easily generalizable.

4.1 Test Execution

Forcing the execution of the test cases (manually or automatically) derived according to one of the criteria above mentioned might not be obvious. As said, if a code-based criterion is followed, it provides us with entry-exit paths over the graph model, and test inputs that execute the corresponding program paths need be found. If a specification-based criterion relying on coverage of a LTS or a FSM is adopted, the test cases are paths over them corresponding to sequences of events, that are specified at the abstraction level of the specifications; more precisely they are labels within the signature of the adopted language. To derive concrete test cases, the labels of the specification language must be translated into corresponding labels at code level, and eventually into execution statements to be launched on the GUI of the used test tool.

The larger the system, the more desirable to keep the abstraction level of the specification high enough to preserve an intuitive view of relevant functions on which the system architect can reason. The translation difficulty also depends on the degree of formality of the refinement process from the high level specification to design and to code. In fact, if a strict formal refinement process is adopted, the translation can be as simple as applying some substitution rules (coded along development). In real world practice, this is not generally the case, and a heavy effort for manual translation may be required, as discussed in [11].

In addition to translating the specified test cases into executable runs, another requirement is the ability to put the system into a state from which the specified tests can be launched. This is sometimes referred to as the *test precondition*. In synchronous systems before a specific command can be executed, several runs in sequence are required to put the system in the suitable test precondition. An effective way to deal with this is to arrange the selected test cases into suitable sequences, such that each test leaves the system into a state that is the precondition for the subsequent test in the sequence. The problem has been early formalized and tackled (for VDM specifications) in [27]. This approach cannot easily scale up to the integration testing of large, complex systems, in which the specified tests involve actions specific to exercise a subsystem. It can be alleviated by always defining the tests at the external interfaces, i.e., as complete I/O sequences.

A new dimension to the problem is added in concurrent systems allowing for nondeterminism. In this case, the system behavior not only depends on the internal status, but also on the interleaving of events with system tasks and other concurrently running systems. When testing reveals a failure, the task of recreating the conditions that made it occur is termed test *replay*. Exact replay requires mechanisms for capturing the happening of synchronization events and memory access, and for forcing the same order of events when a test is replayed. The deterministic approach was originally introduced in [21] for the Ada language. More recent algorithms and tools have been proposed for Java [32]. The approach is highly intrusive, as it requires to heavily instrument the system. A more pragmatic approach otherwise is to keep repeating a test until the desired sequence is observed (fixing a maximum number of iterations).

An orthogonal problem arises during integration testing, when testing only parts of a larger system. Indeed, the testing task itself requires a large programming effort: to be able to test a piece of a large system we need to "simulate" the surrounding environment of the piece under tests (e.g., the caller and called methods). This is done by developing ad hoc drivers and stubs; some commercial test tools exist than can facilitate these tasks.

4.2 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 [83].

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.

Different approaches can be taken. In a scenario in which a limited number of test cases is executed, sometimes even derived manually, the oracle is 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, and 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: we could use the oracle instead! Hence the need of approximate solutions.

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 correct). 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* [4] of an oracle is introduced. It could be measured 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].

A recent survey of approaches to automated test oracles is [4]. The authors overview the use of assertions embedded into the program and providing run-time checking capability; the use of conditions expressly specified to be used as test oracles, in contrast with using the "pure" specifications (i.e., written to model the system behavior and not for run-time checking); and finally the analysis of execution traces.

4.3 Analysis of Test Results

Starting to talk of test case selection, I mentioned that researchers continuously strive for finding "good" criteria. But what makes a criterion better than another? In general terms 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. Some people misconceive the meaning of coverage measures and confuse coverage with effectiveness. More properly, coverage is relative to the tests themselves and measure their thoroughness in exercising RM. Being systematic and trying of not leaving elements of code or of the specification untested is certain a prudent norm, but should be properly understood for what it is. A real measure of test effectiveness should be relative to the program and should allow testers to quantify the effect of testing on the program's attribute of interest, so that the test process can be kept under control.

We have already mentioned that one intuitive and diffuse practice is to count the number of failures or faults detected. The test criterion that founds 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?

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.

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

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.

4.4 The Notion of Software Reliability

Reliability is one of the several factors, or "ilities", into which the quality of software has been decomposed to be able to measure it, and by far it is the factor for which today the most developed theory exists. Strictly speaking, software reliability is the probability that the software will execute without failure in a given environment for a given period of time [59]. In particular, to assess the reliability "in a given environment", this input distribution should approximate as closely as possible the *operational distribution* for that environment.

The extreme difficulty of identifying an operational distribution for software systems is one of the arguments brought by opponents of software reliability [45]. However, the practical approach proposed by Musa to define even a course operational profile by grouping different typologies of users and functionalities has demonstrated great effectiveness ([59] Chapt. 5).

After the tests have been executed, a reliability measure is obtained by analyzing the test results against a selected reliability model. *Reliability growth* models are used to model the situation that as a program is tested and failures are encountered, the responsible faults are found and fixed, and the program is again submitted to testing; this process continues until it is evaluated that an adequate reliability measure has been reached. These models are in fact called reliability growth models, understanding that as the software undergoes testing and is repaired, its reliability increases.

The first software reliability growth models appeared in the early 1970's. Today, tens of models are in use (see [59], Chapt. 3). No "universally best" model can be individuated; studies have shown that although a few models can be rejected because they perform badly in all situations, no single model always performs well for all potential users. However, at the current state of the art, it is usually possible to obtain reasonably accurate reliability predictions.

Reliability, or statistical, models are used instead after the debugging process is finished to assess the reliability one can expect from the software under test, which is also called *life testing*. If a set n of test cases randomly drawn from the *operational* distribution is executed, and f is the number of test inputs that have failed, then an estimate of reliability is very simply given by

$$R = 1 - \frac{f}{n}$$

When there are not enough observed failures against which the predicted reliability can be checked, we could evaluate the probability that the reliability is high enough by using a purely statistical estimate. Assuming that the probability of failure per input ϑ is constant over time, and that each test outcome is independent from the others, an upper bound for the failure probability can be estimated with confidence (1-C), where:

$$C \leq (1 - \vartheta)^n$$

In safety critical applications the reliability requirements are so high that no feasible test campaign can ever be sufficient to demonstrate they have been reached. This is known as the "ultra-high reliability" problem, and only the combined usage of different means for getting evidence of correct behavior can have some hope to attack the problem [58].

5 Putting It Altogether in a Seamless Process

While several problems have been discussed so far, the real big challenge ahead is to work out a unified process within which all these test tasks are gracefully complementing each other, and testing as a whole is not an activity detached from construction (and posterior to it), but the two things, building and checking, become two faces of the same coin, two seamlessly integrated activities. What does this mean? For instance, that there is not anymore in the Laboratory a CASE tool from vendor X for modelling and design using methodology A, and another tool from vendor J for testing according to technique B, but a *unique* environment for modelling, design *and* testing, relying on the same notation, terminology, and reference framework. As obvious as this sounds, reality is yet quite far from it.

Detachment between development and testing has been the key cause for leaving testing as a second-class activity, something that is recognized as important but that anyway one can get rid of in emergency. Researchers have their responsibility in this, as software testing research has not liaised with research in formal methods and modelling notations, and results itself fragmented into specialized sectors. A notable example in this sense is the separation between the two communities addressing the test of general applications, and the more specialized test of communication systems and protocols. Until very recently research in these two sectors has been carried out with almost no interactions, but an interesting initiative is now attempting to coordinate researchers throughout Europe into the Testnet Network [72].

Another evidence is given by the UML: while a big emphasis has been devoted to how to use UML for modelling and how the notation expressiveness and precision could be improved, relatively little attention has been focused on how UML could advance testing.

5.1 (Specify/Design for) Testability

What are the properties that enhance the capability to test a program? How should we model/specify/design a system during development so that testing: i) requires less effort, and ii) is more effective? Clearly, practical answers to such questions would have a great impact in software industry, as testing continues to be very expensive. The term "testability" captures this intuitive notion of "ease" to test a program and is included in the ISO/IEC 9126 [52] list of software quality attributes.

Research has addressed the two questions i) and ii) above along two separate tracks. For example, in [3] the authors define a notion of "testability" linked to the effort needed to accomplish the coverage required by various testing methods, e.g., branch coverage. In [38] the testability of a component is evaluated through the I/O mapping structure and the fan-in/fan-out measure. This deterministic notion of testability attempts to base the evaluation of the effort to be put in testing on measurable properties of program, with the aim to suggest design approaches that can eventually make testing less expensive. This is useful work towards addressing question i), but this measure of testability has no necessary relationship to the ability to detect faults (which was question ii).

A different, probabilistic notion of testability is that in [78, 15]. Program "testability" is here taken as the probability that a program will fail under test, if it is faulty. Having a knowledge of the testability of a program in this sense would permit to interpret with higher precision the results of testing and obtain more favorable predictions than allowed by "black-box" based inference alone.

5.2 Test Phases

Testing of large systems is organized into phases, i.e., the testing task is partitioned into a phased process, addressing at each step the testing of a subsystem. Integration testing refers to the testing of the interactions between subsystems along system composition. An incremental systematic approach should be taken, as opposed to a big-bang approach. The aim is to keep complexity under control and to eventually arrive at the final stage of system testing with all the composing subsystems extensively tested. At each stage test selection is closely related with the object under test.

Some white-box approaches propose to derive integration test cases based on the call structure among modules and measure inter-procedural coverage [47]. In OO systems, integration tests consist of interleaved sequences of module paths and messages, and they are derived considering the interaction patterns between objects, for instance the collaborations or the client-server hierarchy [16].

Until very recently, specification- or design-based integration testing has been performed ad hoc. With the emergence of Software Architecture (SA) as a discipline to describe and analyze large, complex systems, several authors have advocated the use of architectural models also to drive testing, and in particular to select relevant behaviors of interactions between system components. The key issue would be to use the SA *not only* to derive a set of test cases, *but also* to drive the decomposition of a system into subsystems, thus identifying a suitable integration test process. There is not much work on this. In [10, 11], we derive from a formal SA model the LTS, on which graph coverage techniques can be applied. The added value of working at the architectural level of description is that the tester can single out relevant sequences by operating a (trace-preserving) abstraction (called an *observation-function*) over the complete LTS and thus deriving an abstract ALTS. By applying different observation-functions, different system configurations can be tested.

It should be emphasized that most practical testing actually corresponds to *regression testing*, i.e., after software modification or evolution already a set of already passed tests are re-executed to verify that no new faults have been introduced. As regression testing might be very expensive, lot of research has been devoted to select an optimal subset of tests [70].

5.3 Test Patterns

A practical instrument to the design of complex systems are patterns. A design pattern is an abstract description of a recurring problem, together with a general arrangement of elements and procedures that has proved to be useful to solve it. Patterns have always been used by expert designers and engineers: they form their cultural expertise. The famous book from Gamma et al. [39] was the first methodical attempt to document and catalogue design patterns for OO design.

Symmetric to design patterns comes the idea of identifying and logging interesting and recurrent patterns in the testing of complex systems. Unfortunately, there is not much work in this sense. The encyclopedic Binder's book [16] collects patterns for test design in the context of OO systems, organized into class test patterns, subsystem test patterns and system test patterns, with each pattern offering a stand-alone solution for a scope-specific test plan. Certainly more research and empirical work towards the definition of test patterns is desirable, and an eventual catalogue would testify a more mature status of the discipline of testing.

5.4 Testing in Component-oriented Development

The emerging paradigm of development is Component Based (CB), in which a complex system is built by assembling simpler pieces, similarly to what is routine practice in any traditional engineering domain. Ideally, by adopting a CB approach, production times and costs can be greatly reduced, while at the same time more manageable and reliable systems can be obtained. However, the real take-up of a CB approach requires to solve a lot of still open questions (see [25] for a possible list).

Certainly a revision of development processes is necessary, and this revision also involves the testing stage. In particular [13], the traditional three phases of the testing process (unit, integration and system test) can be revisited in terms of three corresponding new phases, respectively referred to as component, deployment and system test. Indeed, the tests performed by the developer are clearly inadequate to guarantee a dependable usage of the component in the target application environment (that of the system constructor). The testing should be repeated also in the final environment, while the component interacts with the other components.

Component testability [40] has been hence identified as an important quality to make easier the evaluation of components by means of testing. Four different perspectives on testability are identified:

- Component Observability: this defines the ease with which the behaviour of a component can be observed in term of its in/out parameters
- Component Traceability: this defines the capability to trace its behavior and the state of its attributes
- Component Controllability: this defines the ease with which the behaviour of the component can be controlled on its in/out, operations and behaviour
- Component Understandability: this refers to how much component information is provided and how well it is presented

In particular, to increase the Understandability of a component, some authors have proposed to augment the components themselves with additional information, in form of meta-data [63, 71].

A new challenge that arises from the inherently distributed nature of CB production is the so-called *component trust problem*, which refers to the difficulty to understand what a component really does. The trust problem is strongly related to the customer capability to validate the adequacy of a candidate component to his/her specific purposes. Different approaches are under study; in particular some authors support the constitution of independent agencies that act as a software certification laboratory (SCL), performing extensive testing in different usage domains [79, 80]. Another approach proposes a form of self certification, in which the component developer releases to the component user a set of test cases, in the form of a XML file, to retest the component in the final application environment [61].

Regarding testing, in our knowledge there is not much work addressing the problem. A first approach proposes to embed test cases in the component itself (Built-In Tests) in the form of methods externally visible and specifically assigned to the execution of tests [81]. A disadvantage of this approach is the size growth of components. To overcome this problem, another approach introduced the concept of a testable architecture. This architecture foresees that the components implement a particular interface for testing purposes, that permits to execute pre-built tests without the necessity to include them in the component code [41].

In this context, we are currently developing a framework for facilitating the deployment testing of components [14]. In this framework the specification of functional tests by the system assembler and the implementation details of components are decoupled thanks to the use of generic *test wrappers*. These dynamically adapt the component interfaces, permitting to easily reconfigure the subsystem under test after the introduction/removal/substitution of a component, and to optimize test reuse, by keeping trace of the test cases that directly affect the wrapped component.

6 A Brief Conclusive Remark

I hope this general overview has showed the several complex facets of the often undervalued testing task, and has moved the curiosity of some ASM scientists while challenging them to put in place this visionary dream of a seamless

build&test process within ASM. About the references provided, they span a period of over than 20 years. Not always the citations are the most recent entries for a topic, but sometimes have been chosen as the most representative. A recent look to future research perspectives is provided in [46].

Although there is much room for automation in each of the involved activities, the tester's expertise remains essential as much as a need for approximate solutions under constrains remains. Again, testing is a challenging and important activity. After having opened with the famous Dijkstra aphorism that highlights its limitations, let me conclude with the as famous quotation from Knuth[5]: *Beware of bugs in the above code; I have only proved it correct, not tried it.*

References

1. Adams, E.: Optimizing Preventive Service of Software Products. IBM J. of Research and Development **28**,1 (1984) 2–14
2. Agedis: Automated Generation and Execution of Test Suites for DIstributed Component-based Software. <http://http://www.agedis.de/index.shtml>
3. Bache, R., Müllerburg, M.: Measures of Testability as a Basis for Quality Assurance. Software Engineering Journal. **5** (March 1990) 86–92
4. Baresi, L., Young, M.: Test Oracles. Tech. Report CIS-TR-01-02. On line at <http://www.cs.uoregon.edu/ michal/pubs/oracles.html>
5. Basanieri, F., Bertolino, A., Marchetti, E.: The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects. Proc. 5th Int. Conf. UML 2002, Dresden, Germany. LNCS 2460 (2002) 383–397
6. Basili, V.R., Selby, R.W.: Comparing the Effectiveness of Software Testing Strategies. IEEE Trans. Software Eng. **13**(12) (1987) 1278–1296
7. Bernot, G., Gaudel M.C., Marre, B.: Software Testing Based On Formal Specifications: a Theory and a Tool. Software Eng. Journal **6** (1991) 387–405
8. Bertolino, A.: ISSTA 2002 Panel: Is ISSTA Research Relevant to Industrial Users?. In [36] 201-202 (see also following entries 203–209)
9. Bertolino, A.: Knowledge Area Description of Software Testing. Guide to the SWEBOK, Joint IEEE-ACM Software Engineering Coordinating Committee. (2001) Online at: <http://www.swebok.org>
10. Bertolino, A., Corradini, F, Inverardi, P., Muccini, H.: Deriving Test Plans from Architectural Descriptions. Proc. Int. ACM Conf. on Soft. Eng. (2000) 220–229
11. Bertolino, A., Inverardi, P., Muccini, H.: An Explorative Journey from Architectural Tests Definition downto Code Tests Execution. Proc. Int. Conf. on Soft. Eng. (2001) 211–220
12. Bertolino, A., Marré, M.: A General Path Generation Algorithm for Coverage Testing. Proc. 10th Int. Soft. Quality Week, San Francisco, Ca. (1997) pap. 2T1
13. Bertolino, A., Polini, A.: Re-thinking the Development Process of Component-Based Software, ECBS 2002 Workshop on CBSE, Lund, Sweden (2002)
14. Bertolino, A., Polini, A.: WCT: a Wrapper for Component Testing. Proc. Fidji'2002, Luxembourg (to appear) (2002)
15. Bertolino, A., Strigini, L.: On the Use of Testability Measures for Dependability Assessment IEEE Trans. Software Eng. **22**(2) (1996) 97–108

[5] This sentence was in a personal communication to P. van Emde Boas (1977).

16. Binder, R.V.: Testing Object-Oriented Systems - Models, Patterns, and Tools. Addison-Wesley (2000)
17. Bochmann, G.V., Petrenko, A.: Protocol Testing: Review of Methods and Relevance for Software Testing. Proc. Int. Symp. on Soft. Testing and Analysis (ISSTA), Seattle (1994) 109–124.
18. Börger, E.: The Origins and the Development of the ASM Method for High Level System Design and Analysis. J. of Universal Computer Science **8**(1) (2002) 2–74. On line at: <http://www.jucs.org/jucs_8_1/the_origins_and_the>
19. Briand, L., Labiche, Y.: A UML-Based Approach to System Testing. Software and Systems Modeling **1**(1) (2002) 10–42
20. Brinksma, E., Tretmans, J.: Testing Transition Systems: An Annotated Bibliography. Proc. of MOVEP'2k, Nantes (2000) 44–50
21. Carver, R.H., Tai, K.-C.: Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs. IEEE Trans. on Soft. Eng. **24**(6) (1998) 471–490
22. Choi, J-D., Zeller, A.: Isolating Failure-Inducing Thread Schedules. In [36] 210–220
23. Coward, P.D.: Symbolic Execution Systems - A Review. Software Eng. J. (1988) 229–239
24. Clarke, L.A., Richardson, D.J.: Applications of Symbolic Evaluation. The J. of Systems and Software **5** (1985) 15–35
25. Crnkovic, I.: Component-based Software Engineering - New Challenges in Software Development. John Wiley&Sons (2001)
26. DeMillo, R.A., Offutt, A. J.: Constraint-Based Test Data Generation. IEEE Trans. Software Eng. **17**(9) (1991) 900–910
27. Dick, J., Faivre, A.: Automating The Generation and Sequencing of Test Cases From Model-Based Specifications. Proc. FME'93, LNCS 670 (1993) 268–284
28. Dijkstra, E.W.: Notes on Structured Programming. T.H. Rep. 70-WSK03 (1970) On line at: <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>
29. Duesterwald, E., Gupta, R., Soffa, M. L.: Rigorous Data Flow Testing Through Output Influences. Proc. 2nd Irvine Software Symposium, Irvine, CA. (1992) 131–145
30. Duran, J. W., Ntafos, S. C.: An Evaluation of Random Testing. IEEE Trans. Software Eng. **SE-10**(4) (1984) 438-444
31. Dunsmore, A., Roper, M., Wood, M.: Further Investigations into the Development and Evaluation of Reading Techniques for Object-Oriented Code Inspection. Proc. 24th Int. Conf. on Soft. Eng. Orlando, FL, USA (2002) 47–57
32. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded Java Program Test Generation. IBM Systems Journal **41** (2002) 111–125
33. Fagan, M.R.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal **15**(3) (1976) 182–211
34. Fenton, N.E., Ohlsson, N.: Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Trans. Software Eng. **26**(8) (2000) 797-814
35. Forgacs, I., Bertolino, A.: Preventing Untestedness in Data-flow Based Testing. Soft. Testing, Verification and Reliability **12**(1)(2001) 29–61
36. Frankl, P.G. (Ed.): Proc. ACM Sigsoft Int. Symposium on Soft. Testing and Analysis ISSTA 2002, Soft. Engineering Notes **27**(4) Roma, Italy (July 2002)
37. Frankl, P.G., Hamlet, R.G., Littlewood, B., Strigini, L.: Evaluating Testing Methods by Delivered Reliability. IEEE Trans. Software Eng. **24**(8) (1998) 586–601
38. Freedman, R.S.: Testability of Software Components. IEEE Trans. Software Engineering **17**(6) (1991) 553–564
39. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)

40. Gao, J.: Component Testability and Component Testing Challenges, Proc. ICSE Workshop on Component-Based Soft. Eng. (2000) http://www.sei.cmu.edu/cbs/cbse2000/paper/
41. Gao, J., Gupta, K., Gupta, S., Shim, S.: On Building Testable Software Components. Proc. ICCBSS2002, LNCS 2255 (2002) 108–121
42. Gargantini, A., Riccobene, E.: ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. J. of Universal Computer Science **7**(11) (2001) 1050–1067. On line at: <http://www.jucs.org/jucs_7_11/asm_based_testing_coverage>
43. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating Finite State Machines from Abstract State Machines. In [36] 112-122
44. Hamlet, D.: Continuity in Software Systems. In [36] 196-200
45. Hamlet, D., Taylor, R.: Partition Testing Does Not Inspire Confidence. IEEE Trans. Software Eng. **16**(12) (1990) 1402-1411
46. Harrold, M.J.: Testing: A Roadmap. In A. Finkelstein (Ed.), The Future of Software Engineering, ACM Press (2000) 63–72
47. Harrold, M.J., Soffa, M.L.: Selection of Data for Integration Testing.IEEE Software (March 1991) 58–65
48. Hierons, R.M.: Testing from a Z Specification. Soft. Testing, Verification and Reliability **7**(1997) 19–33
49. Hierons, R., Derrick, J. (Eds): Special Issue on Specification-based Testing. Soft. Testing, Verification and Reliability **10** (2000)
50. Hiller, M., Jhumka, A., Suri, N.: PROPANE: An Environment for Examining the Propagation of Errors in Software. In [36] 81–85
51. Howden, W.E.: Weak Mutation Testing and Completeness of Test Sets. IEEE Trans. Software Eng. **8**(4) (1982) 371–379
52. ISO/IEC 9126, Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for Their Use. (1991)
53. Fernandez, J.-C., Jard, C., Jeron, T., Nedelka, L., Viho, C.: Using On-the-fly Verification Techniques for the Generation of Test Suites. Proc. of the 8th Int. Conf. on Computer Aided Verification (1996)
54. Korel, B.: Automated Software Test Data Generation. IEEE Trans. Software Eng. **16**(8) (1990) 870–879
55. Laprie, J. C.: Dependability - Its Attributes, Impairments and Means. In [66] 3–18
56. Latella, D., Massink, M.: On Testing and Conformance Relations for UML Statechart Diagrams Behaviours. In [36] 144–153
57. Littlewood, B., Popov, P.T., Strigini, L., Shryane, N.: Modeling the Effects of Combining Diverse Software Fault Detection Techniques. IEEE Trans. Software Eng. **26**(12) (2000) 1157–1167
58. Littlewood, B., Strigini, L.: Validation of Ultra-High Dependability for Software-based Systems. Communications of the ACM. **36**(11) (1993) 69–80
59. Lyu, M.R. (Ed.): Handbook of Software Reliability Engineering. IEEE Comp. Soc. Press/McGraw-Hill (1996)
60. Morell, L.J.: A Theory of Fault-based Testing. IEEE Trans. Software Eng. **16**(8) (1990) 844–857
61. Morris, J., Lee, G., Parker, K., Bundell, G.A., Lam, C.P.: Software Component Certification. IEEE Computer (September 2001) 30–36
62. Myers, G.J.: The Art of Software Testing. Wiley. (1979)
63. Orso, A., Harrold, M.J., Rosenblum, D.: Component Metadata for Software Engineering Tasks. Proc. EDO2000, LNCS 1999 (2000) 129–144
64. Ostrand, T.J., Balcer, M.J.: The Category-Partition Method for Specifying and Generating Functional Tests. ACM Comm. **31**(6) (1988) 676–686

65. Pargas, R., Harrold, M.J., Peck, R.: Test-Data Generation Using Genetic Algorithms. J. of Soft. Testing, Verifications, and Reliability **9** (1999) 263–282
66. Randell, B., Laprie, J.C., Kopetz, H., Littlewood B., Eds.: Predictably Dependable Computing Systems, Springer (1995)
67. Rapps, S., Weyuker, E.J.: Selecting Software Test Data Using Data Flow Information. IEEE Trans. Software Eng., **SE-11** (1985) 367–375
68. Richardson, D.J, Clarke, L.A.: Partition Analysis: A Method Combining Testing and Verification. IEEE Trans. Software Eng., **SE-11** (1985) 1477–1490
69. Richardson, D., Thompson, M.C.: The Relay Model for Error Detection and its Application. Proc. 2nd Wksp Soft. Testing, Verification, and Analysis. Banff Alberta, ACM/Sigsoft and IEEE (July 1988) 223–230
70. Rothermel, G., Harrold, M.J.: Analyzing Regression Test Selection Techniques. IEEE Trans. Software Eng., **22**(8) (1996) 529–551
71. Stafford, J.A., Wolf, A.L.: Annotating Components to Support Component-Based Static Analyses of Software Systems, Proc. Grace Hopper Celeb. of Women in Computing (2001)
72. TESTNET – Integration of Testing Methodologies. <http://http://www-lor.int-evry.fr/testnet/>.
73. TGV–Test Generation from transitions systems using Verification techniques. On line at <http://www.inrialpes.fr/vasy/cadp/man/tgv.html>
74. Thevenod-Fosse P., Waeselynck H., Crouzet Y.: Software Statistical Testing. In [66] 253–272
75. J. Tretmans. Conformance Testing with Labeled Transition Systems: Implementation Relations and Test Generation. Computer Networks and ISDN Systems **29** (1996) 49–79
76. Vegas, S.: Characterisation Schema for Selecting Software Testing Techniques. Int. Software Engineering Research Network (ISERN'01) Strachclyde, Scotland (2001)
77. Voas, J. M.: PIE: A Dynamic Failure-Based Technique. IEEE Trans. Software Eng. **18**(8) (1992) 717-727
78. Voas, J.M., Miller, K. W.: Software Testability: The New Verification. IEEE Software (May 1995) 17–28
79. Voas, J.: Certifying Off-the-Shelf Software Components. IEEE Computer (June 1998) 53–59
80. Voas, J.: Developing a Usage-Based Software Certification Process. IEEE Computer (August 2000) 32–37
81. Wang, Y., King, G., Wickburg, H.: A Method for Built-in Tests in Component-based Software Maintenance, Proc. European Conference on Soft. Maintenance and Reengineering (1998) 186–189
82. Weyuker, E.J.: Translatability and Decidability Questions for Restricted Classes of Program Schemas. SIAM J. on Computers **8**(4) (1979) 587–598
83. Weyuker, E.J.: On Testing Non-testable Programs. The Computer Journal **25**(4) (1982) 465–470
84. Weyuker, E.J., Jeng, B.: Analyzing Partition Testing Strategies. IEEE Trans. Software Eng. **17**(7) (1991) 703–711
85. Weyuker, E.J., Ostrand, T.J.: Theories of Program Testing and the Application of Revealing Subdomains. IEEE Trans. Software Eng. **SE-6**(1980) 236-246
86. Wood, M., Roper, M., Brooks, A., Miller, J.: Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. Proc. ESEC/FSE, LNCS 1301 (1997) 262–277
87. Zhu, H., Hall, P.A.V., May, J.H.R.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys, **29** (1997) 366–427