Software Testing Research: Achievements, Challenges, Dreams

Antonia Bertolino

Antonia Bertolino (http://www.isti.cnr.it/People/A.Bertolino) is a Research Director of the Italian National Research Council at ISTI in Pisa, where she leads the Software Engineering Laboratory. She also coordinates the Pisatel laboratory, sponsored by Ericsson Lab Italy.

Her research interests are in architecture-based, component-based and service-oriented test methodologies, as well as methods for analysis of non-functional properties.

She is an Associate Editor of the *Journal of Systems and Software* and of *Empirical Software Engineering Journal*, and has previously served for the *IEEE Transactions on Software Engineering*. She is the Program Chair for the joint *ESEC/FSE Conference* to be held in Dubrovnik, Croatia, in September 2007, and is a regular member of the Program Committees of international conferences, including ACM ISSTA, Joint ESEC-FSE, ACM/IEEE ICSE, IFIP TestCom. She has (co)authored over 80 papers in international journals and conferences.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Software Testing Research: Achievements, Challenges, Dreams

Antonia Bertolino
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche
56124 Pisa, Italy
antonia.bertolino@isti.cnr.it

Abstract

Software engineering comprehends several disciplines devoted to prevent and remedy malfunctions and to warrant adequate behaviour. Testing, the subject of this paper, is a widespread validation approach in industry, but it is still largely ad hoc, expensive, and unpredictably effective. Indeed, software testing is a broad term encompassing a variety of activities along the development cycle and beyond, aimed at different goals. Hence, software testing research faces a collection of challenges. A consistent roadmap of the most relevant challenges to be addressed is here proposed. In it, the starting point is constituted by some important past achievements, while the destination consists of four identified goals to which research ultimately tends, but which remain as unreachable as dreams. The routes from the achievements to the dreams are paved by the outstanding research challenges, which are discussed in the paper along with interesting ongoing work.

1. Introduction

Testing is an essential activity in software engineering. In the simplest terms, it amounts to observing the execution of a software system to validate whether it behaves as intended and identify potential malfunctions. Testing is widely used in industry for quality assurance: indeed, by directly scrutinizing the software in execution, it provides a realistic feedback of its behavior and as such it remains the inescapable complement to other analysis techniques.

Beyond the apparent straightforwardness of checking a sample of runs, however, testing embraces a variety of activities, techniques and actors, and poses many complex challenges. Indeed, with the complexity, pervasiveness and criticality of software growing ceaselessly, ensuring that it behaves according to the desired levels of quality and dependability becomes more crucial, and increasingly difficult and expensive. Earlier studies estimated that testing can consume fifty percent, or even more, of the development costs [3], and a recent detailed survey in the United States [63] quantifies the high economic impacts of an inadequate software testing infrastructure.

Correspondingly, novel research challenges arise, such as for instance how to conciliate model-based derivation of test cases with modern dynamically evolving systems, or how to effectively select and use runtime data collected from real usage after deployment. These newly emerging challenges go to augment longstanding open problems, such as how to qualify and evaluate the effectiveness of testing criteria, or how to minimize the amount of retesting after the software is modified.

In the years, the topic has attracted increasing interest from researchers, as testified by the many specialized events and workshops, as well as by the growing percentage of testing papers in software engineering conferences; for instance at the 28th International Conference on Software Engineering (ICSE 2006) four out of the twelve sessions in the research track focused on "Test and Analysis".

This paper organizes the many outstanding research challenges for software testing into a consistent roadmap. The identified destinations are a set of four ultimate and unachievable goals called "dreams". Aspiring to those dreams, researchers are addressing several challenges, which are here seen as interesting viable facets of the bigger unsolvable problem. The resulting picture is proposed to the software testing researchers community as a work-in-progress fabric to be adapted and expanded.

In Section 2 we discuss the multifaced nature of software testing and identify a set of six questions underlying any test approach. In Section 3 we then introduce the structure of the proposed roadmap. We summarize some more mature research areas, which constitute the starting point for our journey in the roadmap, in Section 4. Then in Section 5, which is the main part of the paper, we overview several outstanding research challenges and the dreams to which they tend. Brief concluding remarks in Section 6 close the paper.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

2. The many faces of software testing

Software testing is a broad term encompassing a wide spectrum of different activities, from the testing of a small piece of code by the developer (unit testing), to the customer validation of a large information system (acceptance testing), to the monitoring at run-time of a network-centric service-oriented application. In the various stages, the test cases could be devised aiming at different objectives, such as exposing deviations from user's requirements, or assessing the conformance to a standard specification, or evaluating robustness to stressful load conditions or to malicious inputs, or measuring given attributes, such as performance or usability, or estimating the operational reliability, and so on. Besides, the testing activity could be carried on according to a controlled formal procedure, requiring rigorous planning and documentation, or rather informally and ad hoc (exploratory testing).

As a consequence of this variety of aims and scope, a multiplicity of meanings for the term "software testing" arises, which has generated many peculiar research challenges. To organize the latter into a unifying view, in the rest of this section we attempt a classification of problems common to the many meanings of software testing. The first concept to capture would be what is the common denominator, if it exists, between all possible different testing "faces". We propose that such a common denominator can be the very abstract view that, given a piece of software (whichever its typology, size and domain) testing always consists of *observing a sample of executions*, and giving a verdict over them.

Starting from this very general view, we can then concretize different instances, by distinguishing the specific aspects that can characterize the sample of observations:

WHY: why is it that we make the observations? This question concerns the *test objective*, e.g.: are we looking for faults? or, do we need to decide whether the product can be released? or rather do we need to evaluate the usability of the User Interface?

HOW: which sample do we observe, and how do we choose it? This is the problem of *test selection*, which can be done ad hoc, at random, or in systematic way by applying some algorithmic or statistical technique. It has inspired much research, which is understandable not only because it is intellectually attractive, but also because how the test cases are selected -the test criterion- greatly influences test efficacy.

HOW MUCH: how big of a sample? Dual to the question of how do we pick the sample observations (test selection), is that of how many of them do we take (*test adequacy*, or stopping rule). Coverage analysis or reliability measures constitute two "classical" approaches to answer such question.

WHAT: what is it that we execute? Given the (possibly composite) system under test, we can observe its execution either taking it as a whole, or focusing only on a part of it, which can be more or less big (unit test, component/subsystem test, integration test), more or less defined: this aspect gives rise to the various *levels of testing*, and to the necessary scaffolding to permit test execution of a part of a larger system.

WHERE: where do we perform the observation? Strictly related to what do we execute, is the question whether this is done in house, in a simulated environment or in the target final context. This question assumes the highest relevance when it comes to the testing of embedded systems.

WHEN: when is it in the product lifecycle that we perform the observations? The conventional argument is that the earliest, the most convenient, since the cost of fault removal increases as the lifecycle proceeds. But, some observations, in particular those that depend on the surrounding context, cannot always be anticipated in the laboratory, and we cannot carry on any meaningful observation until the system is deployed and in operation.

These questions provide a very simple and intuitive characterization schema of software testing activities, that can help in organizing the roadmap for future research challenges.

3. Software testing research roadmap

A roadmap provides directions to reach a desired destination starting from the "you are here" red dot. The software testing research roadmap is organised as follows:

- the "you are here" red dot consists of the most notable *achievements* from past research (but note that some of these efforts are still ongoing);
- the desired destination is depicted in the form of a set of (four) *dreams*: we use this term to signify that these are asymptotic goals at the end of four identified routes for research progress. They are unreachable by definition and their value exactly stays in acting as the poles of attraction for useful, farsighted research;
- in the middle are the *challenges* faced by current and future testing research, at more or less mature stage, and with more or less chances for success. These challenges constitute the directions to be followed in the journey towards the dreams, and as such they are the central, most important part of the roadmap.

The roadmap is illustrated in Figure 1. In it, we have situated the emerging and ongoing research directions in the center, with more mature topics -the achievements- on their

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Figure 1. Roadmap

left, and the ultimate goals -the dreams- on their right. Four horizontal strips depict the identified research routes toward the dreams, namely:

1. Universal test theory;
2. Test-based modeling;
3. 100% automatic testing;
4. Efficacy-maximized test engineering.

The routes are bottom-up ordered according somehow to progressive utility: the theory is at the basis of the adopted models, which in turn are needed for automation, which is instrumental to cost-effective test engineering.

The challenges horizontally span over six vertical strips corresponding to the WHY, HOW, HOW MUCH, WHAT, WHERE, and WHEN questions characterizing software testing faces (in no specific order).

Software testing research challenges find their place in this plan, vertically depending on the long term dream, or dreams, towards which they mainly tend, and horizontally according to which question, or questions, of the introduced software testing characterization they mainly center on.

In the remainder of this paper, we will discuss the elements (achievements, challenges, dreams) of this roadmap. We will often compare this roadmap with its 2000's predecessor by Harrold [43], which we will refer henceforth as FOSE2000.

4. You are here: Achievements

Before outlining the future routes of software testing research, a snapshot is here attempted of some topics which constitute the body of knowledge in software testing (for a ready, more detailed guide see also [8]), or in which important research achievements have been established. In the roadmap of Figure 1, these are represented on the left side.

The origins of the literature on software testing date back to the early 70's (although one can imagine that the very notion of testing was born simultaneously with the first experiences of programming): Hetzel [44] dates the first conference devoted to program testing to 1972. Testing was conceived like an art, and was exemplified as the "destructive" process of executing a program with the intent of finding errors, opposed to design which constituted the "constructive" party. It is of these years Dijkstra's topmost cited aphorism about software testing, that it can only show the presence of faults, but never their absence [25].

The 80's saw the assumption of testing to the status of an engineered discipline, and a view change of its goal from just error discovery to a more comprehensive and positive view of prevention. Testing is now characterized as *a broad and continuous activity throughout the development process* ([44], pg.6), whose aim is the measurement and evaluation of software attributes and capabilities, and Beizer states: *More than the act of testing, the act of* designing *tests is one of the best bug preventers known* ([3], pg. 3).

Testing process. Indeed, much research in the early years has matured into techniques and tools which help make such "test-design thinking" more systematic and incorporate it within the development process. Several test process models have been proposed for industrial adoption, among which probably the "V model" is the most popular. All of its many variants share the distinction of at least the Unit, Integration and System levels for testing.

More recently, the V model implication of a phased and formally documented test process has been argued by some as being inefficient and unnecessarily bureaucratic, and in contrast more *agile* processes have been advocated. Concerning testing in particular, a different model gaining attention is *test-driven development* (TDD)[46], one of the core *extreme programming* practices.

The establishment of a suitable process for testing was listed in FOSE2000 among the fundamental research topics and indeed this remains an active research today.

Test criteria. Extremely rich is the set of test criteria devised by past research to help the systematic identification of test cases. Traditionally these have been distinguished between white-box (a.k.a. structural) and black-box (a.k.a. functional), depending on whether or not the source code is exploited in driving the testing. A more refined classification can be laid according to the source from which the test cases are derived [8], and many textbooks and survey articles (e.g., [89]) exist that provide comprehensive descriptions of existing criteria. Indeed, so many criteria among which to choose now exist, that the real challenge becomes the capability to make a justified choice, or rather to understand how they can be most efficiently combined. In recent years the greatest attention has been turned to model-based testing, see Section 5.2.

Comparison among test criteria. In parallel with the investigation of criteria for test selection and for test adequacy, lot of research has addressed the evaluation of the relative effectiveness of the various test criteria, and especially of the factors which make one technique better than another at fault finding. Past studies have included several analytical comparisons between different techniques (e.g., [31, 88]). These studies have permitted to establish a subsumption hierarchy of relative thoroughness between comparable criteria, and to understand the factors influencing the probability of finding faults, focusing more in particular on comparing partition (i.e., systematic) against random testing. "Demonstrating effectiveness of testing techniques" was in fact identified as a fundamental research challenge in FOSE2000, and still today this objective calls for further research, whereby the emphasis is now on em-

pirical assessment.

Object-oriented testing. Indeed, at any given period, the dominating paradigm of development has catalyzed testing research for adequate approaches, as we further develop in Section 5.5. In the 90's the focus was on testing of Object-oriented (OO) software. Rejected the myth that enhanced modularity and reuse brought forward by OO programming could even prevent the need for testing, researchers soon realized that not only everything already learnt about software testing in general also applied to OO code, but also OO development introduced new risks and difficulties, hence increasing the need and complexity of testing [14]. In particular, among the core mechanisms of OO development, encapsulation can help hide bugs and makes test harder; inheritance requires extensive retesting of inherited code; and polymorphism and dynamic binding call for new coverage models. Besides, appropriate strategies for effective incremental integration testing are required to handle the complex spectrum of possible static and dynamic dependencies between classes.

Component-based testing. In the late 90's, component-based (CB) development emerged as the ultimate approach that would yield rapid software development with fewer resources. Testing within this paradigm introduced new challenges, which we would distinguish between technical and theoretical in kind. On the technical side, components must be generic enough for being deployed in different platforms and contexts, therefore the component user needs to retest the component in the assembled system where it is deployed. But the crucial problem here is to face the lack of information for analysis and testing of externally developed components. In fact, while component interfaces are described according to specific component models, these do not provide enough information for functional testing. Therefore research has advocated that appropriate information, or even the test cases themselves (as in *Built-In Testing*), are packaged along with the component for facilitating testing by the component user, and also that the "contract" that the components abide to should be made explicit, to allow for verification.

The testing of component-based systems was also listed as a fundamental challenge in FOSE2000. For a more recent survey see [70].

What remains an open evergreen problem is the theoretical side of CB testing: how can we infer interesting properties of an assembled system, starting from the results of testing the components in isolation? The theoretical foundations of *compositional testing* still remain a major research challenge destined to last, and we discuss some directions for research in Section 5.1.

Protocol testing. Protocols are the rules that govern the communication between the components of a distributed system, and these need to be precisely specified in order to facilitate interoperability. Protocol testing is aimed at verifying the conformance of protocol implementations against their specifications. The latter are released by standard organizations, or by consortia of companies. In certain cases, also a standard conformance test suite is released.

Pushed by the pressure of enabling communication, research in protocol testing has proceeded along a separate and, in a sense, privileged trail with respect to software testing. In fact, thanks to the existence of precise state-based specifications of desired behaviour, research could very early develop advanced formal methods and tools for testing conformance to those established standard specifications [16].

Since these results were conceived for a restricted well-defined field of application, they do not readily apply to general software testing. However, the same original problem of ensuring proper interaction between remote components and services arises today on a broader scale for any modern software; therefore software testing research could fruitfully learn from protocol testing the habit of adopting standardized formal specifications, which is the trend in modern service-oriented applications. Viceversa, while early protocols were simple and easily tractable, today the focus is shifting to higher levels of communication protocols, and hence the complexity plague more typical of software testing starts also to become pressing here. Therefore, the conceptual separation between protocol testing and general software testing problems is progressively vanishing.

Reliability testing. Given the ubiquity of software, its reliability, i.e., the probability of failure-free operation for a specified period of time in a specified environment, impacts today any technological product. Reliability testing recognizes that we can never discover the last failure, and hence, by using the operational profile to drive testing, it tries to eliminate those failures which would manifest themselves more frequently: intuitively the tester mimics how the users will employ the system. Software reliability is usually inferred based on *reliability models*: different models should be used, depending on whether the detected faults are removed, in which case the reliability grows, or not, when reliability is only certified.

Research in software reliability has intersected research in software testing in many fruitful ways. Models for software reliability have been actively studied in the years 80's and 90's [58]. These models are now mature and can be engineered into the test process providing quantitative guidance for how and how much to test. For instance, this was done by Musa in his Software-Reliability-Engineered Testing (SRET) approach ([58], Chapt.6), and is also advocated in the Cleanroom development process, which pursues the application of statistical test approaches to yield certified reliability measures [69].

Unfortunately, the practice of reliability testing has not

proceeded at the same speed of theoretical advances in software reliability, probably because it is (perceived as) a complex and expensive activity, but also for the inherent difficulty of identifying the required operational profile [41]. Yet today the demand for reliability and other dependability qualities is growing and hence the need arises for practical approaches to coherently test functional and extra-functional behaviour of modern software-intensive systems, as discussed further in Section 5.5. For future challenges in reliability testing we refer to Lyu's roadmap [57].

5. The routes

In this section we describe the dreams of software testing research, and for each of them some relevant challenges to be addressed to advance the state of the art closer to the dream itself.

5.1. Dream: Universal test theory

One of the longstanding dreams of software testing research would be to build a sound comprehensive theory which is useful to backup and nourish test technology. By asking for a "universal" test theory I mean one coherent and rigorous framework to which testers can refer to understand the relative strengths and limitations of existing test techniques, and to be guided in selecting the most adequate one, or mix thereof, given the present conditions.

Seminal work in software testing theory dates back to the late 70's, when the related notions of a "reliable" [45] or an "ideal" [36] test suite were first introduced. Thanks to this pioneering work, we have logical arguments to corroborate the quite obvious fact that testing can never be exact [25]. But such knowledge *per se*, in addition to the warning that even though many tests passed, the software can still be faulty, provides little guidance about what is it then that we can conclude about the tested software after having applied a selected technique, or going even further, about how we could dynamically tune up our testing strategy as we proceed with accumulating test results, taking into account what we observe.

The dream would be to have a test machinery which ties a statement of the goal for testing with the most effective technique, or combination of techniques, to adopt, along with the underlying assumptions that we need to make. Towards this dream research needs to address several challenges.

Challenge: Explicit test hypotheses

Ultimately, given that testing is necessarily based on approximations (remember we started from the statement that testing amounts to sampling some executions), this universal theory should also make explicit for each technique which are its underlying assumptions, or *test hypotheses*: firstly formalized in [6], the concept of a test hypothesis justifies the common and intuitive test practice behind the selection of every finite test set, by which a sample is taken as the representative of several possible executions. With the exception of few formal test approaches, test hypotheses are usually left implicit, while it would be of utmost importance to make them explicit. In this way, if we perform "exhaustive" testing according to the selected test criterion, from successfully completing the testing campaign we could justifiably conclude that the software is *correct under the stated hypotheses*: i.e., we still know that actually the software could be faulty, but we also know what we have assumed to be true at the origin and could instead be false. This notion is similar to the one of a "fault model", which is used instead in the realm of protocol testing, where a test suite is said to provide fault coverage guarantee for a given fault model.

A summary of test hypotheses behind most common testing approaches is given for instance by Gaudel [34], who mentions among others Uniformity Hypothesis for black-box partition criteria (the software is assumed to behave uniformly within each test subdomain), and Regularity Hypothesis, using a size function over the tests. Such research should be extended to cover other criteria and approaches. The test hypotheses should be modularized by the test objective: different theories/hypotheses would be necessary when testing for reliability, when testing for debugging, and so on.

By making explicit our assumptions, this challenge refines the WHY do we observe some executions.

Challenge: Test effectiveness

To establish a useful theory for testing, we need to assess the effectiveness of existing and novel test criteria. Although as said among the Achievements, several comparison studies have been conducted to this purpose, Fose2000 already signalled that additional research was needed to provide *analytical, statistical, or empirical evidence of the effectiveness of the test-selection criteria in revealing faults*, in order to *understand the classes of faults for which the criteria are useful*. These challenges are still alive. In particular, it is now generally agreed that it is always more effective to use a combination of techniques, rather than applying only one, even if judged the most powerful, because each technique may target different types of fault, and will suffer from a *saturation effect* [58].

Several works have contributed to a better understanding of inherent limitations of different testing approaches, starting from the seminal Hamlet and Taylor' paper discussing partition testing and its underlying assumptions [41]. Yet further work is needed, notably to contextualize such comparisons to the complexity of real world testing (for instance, Zhu and He [90] analyse the adequacy of testing concurrent systems), as well as to refine assumptions at the

bases of such comparisons, to take into account progresses in test automation. For example, even the conventional controversy about the relative merits of *systematic vs. random* techniques is today revitalized by the arising sophisticated methods for automating random test generation (which are discussed in Section 5.3).

This challenge addresses the WHY, HOW and HOW MUCH of testing, in terms of the faults (which and how many) we target.

Challenge: Compositional testing

The ever growing complexity of software makes testing hard, and hinders progress towards any of research dreams, included test theory. Traditionally, test complexity has been addressed by the ancient *divide et impera* strategy, i.e., the testing of a large complex system is decomposed into the separate testing of its composing "pieces". Much past research has addressed techniques and tools for helping incremental testing strategies in organizing and executing progressively different aggregations of components. For example, different strategies have been proposed to generate the *test order* which is more efficient in minimizing the need of stubs and scaffolding, see [18] for a recent comparison. The problem has become particularly relevant today with the emergence of the CB development paradigm, as already discussed in FOSE2000, and even more with the increasing adoption of dynamic system compositions.

So, we need a chapter of testing theory addressing compositional testing: we need to understand how we can *reuse the test results* observed in the separate testing of the individual pieces (be them Units or Components or Subsystems), in particular what conclusions can be inferred about the system resulting from the composition, and which additional test cases must be run on the integration. Several promising directions of study have been undertaken in different contexts. For instance, Hamlet has proposed a simple foundational theory for *component-based software reliability* [40], recently extended with the notion of state [39], but work is still needed to make it generally applicable.

Blundell and coauthors [15] are instead investigating the application to testing of *assume-guarantee* reasoning, a verification technique used to infer global system properties by checking individual components in isolation. Since to be able to verify a component individually, we need to make assumptions about its context, assume-guarantee verification checks whether a component guarantees a property assuming the context behaves correctly, and then symmetrically the context is checked assuming the component is correct. The promise of assume-guarantee testing would be that by observing the test traces of the individual components one could infer global behaviours.

The protocol test community is also actively investigating compositional testing. For example, van der Bijl and coauthors [81] have formally analysed the parallel composition of two given communication components, based on the *ioco*-test theory [79], which works on Labeled Transition Systems. In particular, if two components have been separately tested and proved to be *ioco*-correct, is their integration *ioco*-correct as well? The authors show that in general this cannot be concluded, but the answer can be affirmative for components whose inputs are completely specified [81]. Gotzhein and Khendek [37] instead have considered the glue code for the integration of communicating components, have produced a fault model for it and developed a procedure to find the test cases for the glue.

This challenge is clearly related to WHAT we test.

Challenge: Empirical body of evidence

Today the importance of experimentation to advance the maturity of software engineering discipline certainly does not need to be underlined (Siøberg and coauthors [77] discuss in depth research challenges faced by empirical methods). In every topic of software engineering research, empirical studies are essential to evaluate proposed techniques and practices, to understand how and when they work, and to improve on them. This is obviously true for testing as well, in which *controlled experimentation is an indispensable research methodology* [26].

In FOSE2000, Harrold identified in this regard the following needs: controlled experiments to demonstrate techniques; collecting and making publicly available sets of experimental subjects; and industrial experimentation. All such needs can be confirmed today, and a more recent review of testing technique experiments [48] sadly concluded that *over half of the existing* (testing technique) *knowledge is based on impressions and perceptions and, therefore, devoid of any formal foundation.*

Indeed, by experimenting, we should aim at producing an empirical body of knowledge which is at the basis for building and evolving the theory for testing. We need to examine factors that can be used to early estimate where faults reside and why, so that test resources can be properly allocated. And for doing this we need to have meaningful experiments, in terms of scale, of the subjects used, and of context, which is not always realistic. Banally, for all three aspects, the barrier is cost: careful empirical studies on large scale products, within real world contexts (such as [66]), and possibly replicated by several professional testers so to attain generally valid results are of course prohibitively expensive. A possible way out to overcome such barrier could be that of joining the forces of several research groups, and carrying out distributed and widely replicated experiment. Roughly the idea would be that of launching sort of "Open Experiments" initiative, similarly to how several Open Source projects have been successfully conducted. Awareness of the need to unite forces is spreading, and some efforts are already being taken toward building shared data repositories, as in [26], or distributed experi-

mental testbeds, such as the PlanetLab [68] collection of more than 700 machines connected around the world.

This is a fundamental challenge which spans over all six characterizing questions.

5.2. Dream: Test-based modeling

A great deal of research focuses nowadays on model-based testing, which we discuss below. The leading idea is to use models defined in software construction to drive the testing process, in particular to automatically generate the test cases. The pragmatic approach that testing research takes is that of following what is the current trend in modeling: whichever be the notation used, say e.g. UML or Z, we try to adapt to it a testing technique as effectively as possible.

But if we are allowed to consider the dream, from the tester's viewpoint the ideal situation would be to reverse this approach with respect to what comes first and what comes after: instead of taking a model and see how we can best exploit it for testing, let us consider how we should ideally build the model so that the software can be effectively tested. Wouldn't it be nice if developers -fully aware of the importance and difficulty of thorough model-based testing- care in advance about testing and derive appropriate models already enhanced with information instrumental for testing? This is the motivation why we are here reversing the current view of "model-based testing" towards the dream of "test-based modeling".

Admittedly, this is just a new term for an old idea, as actually we can find already several research directions that more or less explicitly have been working toward approaching this dream. On one side, this notion of test-based modeling is closely related to, actually a factor of, the old idea of "Design-for-testability", which is primarily concerned with designing software so as to enhance Controllability (of inputs) and Observability (of outputs). But also related can be seen former approaches to testing based on assertions, and more recent ones based on Contracts. Assertions in particular have early been recognized as a useful tool to enhance testing, since they can verify at runtime the internal state of a program. Descending from assertions, contracts were originally introduced at the level of classes for OO software, and have then been adopted for components: intuitively, a contract establishes a "legal" agreement between two interacting parties, which is expressed by means of three different types of assertions: pre-conditions, post-conditions and invariants. The step to using such contracts as a reference for testing is short, and much interesting research is going on with promising results, e.g., [20, 52].

Challenge: Model-based testing

The often cited trends in this paper of rising levels of complexity and needs for high quality are driving the cost of testing higher, to the point where traditional testing practices become uneconomic, but fortunately at the other end, the increasing use of models in software development yields perspective of removing the main barrier to the adoption of model-based testing, which is (formal) modeling skills.

Model-based testing is actually a sort of *Back to the future* movie for software testing. Indeed, the idea of model-based testing has been around for decades (Moore [62] started the research on FSM-based test generation in 1956!), but it is in the last few years that we have seen a groundswell of interest in applying it to real applications (for an introduction to the different approaches and tools in model-based testing see, e.g., [80]).

Nonetheless, industrial adoption of model-based testing remains low and signals of the research-anticipated breakthrough are weak. Therefore, beyond theoretical challenges, researchers are today focusing on how to beat the barriers to wide adoption. There are important technical and process-related issues pending.

A widely recognized issue is how can we *combine* different styles of modeling (such as transition-based, pre/post condition-based and scenario-based). For instance, we need to find effective ways to compose state-based and scenario-based approaches [9, 38]. At Microsoft, where model-based testing has been championed for various years now, but with limited follow-up, a *multi-paradigmatic* approach [38] is now pursued to favor a wider adoption. The idea is that models stemming from different paradigms and expressed in any notation can be seamlessly used within one integrated environment. The lesson learned is in fact that forcing users to use a new notation does not work, instead the core of a model-based testing approach should be agnostic and let developers use existing programming notations and environments [38]. We also need ways to combine model-based criteria with other approaches; for instance a promising idea is to use testing over simulations [72] to optimize the test suite and to boost testing.

Process-related issues concern the need to integrate model-based testing practice into current software processes: perhaps the crucial issues here are the two related needs for test management of making test models as abstract as possible, while still retaining the ability to generate executable tests on one side; and of keeping traceability from requirements to tests all along the development process, on the other. We finally also need industrial-strength tools for authoring and interactive modeling, that can help reduce the inadequate education of current testers (or maybe the excessive expertise requirements of proposed techniques).

A special case of model-based testing is *conformance testing*, i.e., checking whether the system under test complies with its specification, under some defined relation (which is strictly related to the test hypotheses previously discussed). Starting from the 70's, many algorithms have

been proposed; a recent extensive overview of current open challenges is given in Broy and coauthors' tutorial on model-based testing for reactive systems [21]. The results achieved so far are impressive on theoretical grounds, but many of the proposed methods are hardly applicable to realistic systems, even though several tools have been produced and some of these are applied in specialized domains. A good overview of tools for model-based conformance testing built on a sound theory is provided by Belinfante and coauthors [4], who highlight the need to improve and ease the application of the theory.

This challenge refers to the HOW we select which test executions to observe, and partly to the HOW MUCH of them.

Challenge: Anti-model-based testing

Parallel to model-based testing, several efforts are being devoted towards novel forms of testing which lay directly on the analysis of program executions, rather than on an a-priori model. Instead of taking a model, devise from it a test plan, and hence compare the test results back to the model, these other approaches collect information from executing the program, either after actively soliciting some execution, or passively during operation, and try to synthesize from these some relevant properties of data or of behaviour. There can be cases in which the models simply do not exist or are not accessible, such as for COTS or legacy components; other cases in which the global system architecture is not decided a-priori, but is created and evolves dynamically along the life of a system; or, a model is originally created, but during development it becomes progressively less useful since its correspondence with the implementation is not enforced and is lost. Hence, symmetrically to model-based testing, we have that (explicitly or implicitly) a model is derived a posteriori via testing, which we refer to as *anti-model-based testing*, as anticipated in [11]. By this term we refer to all various approaches that by means of testing, reverse-engineering a model, in the form of an invariant over the program variables, or in the form of a state-machine, or a Labelled Transition System, or a Sequence diagram, and so on, and then check such a model to detect whether the program behaves appropriately.

Anti-model-based testing can rely over the great advances of dynamic program analysis, which is a very active research discipline today, as discussed by Canfora and Di Penta [22].

We need to be able to infer system properties by reasoning on a limited set of observed traces, or even partial traces, since we might observe the components that form the system. In a recent work, Mariani and Pezzè [59] propose the BCT technique to derive behaviour models for monitored COTS components. In their approach the behavioural models consist of both I/O models, obtained by means of the well-known Daikon dynamic invariant detector [30], and interaction models, in the form of Finite State Automata. Interestingly, these derived models can afterward be used for model-based testing if and when the components are replaced by new ones. A related challenge is to maintain the dynamically derived models up-to-date: depending on the type of upgrade to the system, also the model can need to be refined, as Mariani and Pezzè also observe, outlining some possible strategies.

This challenge as well refers to the HOW and the HOW MUCH we observe of test executions.

Challenge: Test oracles

Strictly related to test planning, and specifically to the problem of how to derive the test cases, is the issue of deciding whether a test outcome is acceptable or not. This corresponds to the so-called "oracle", ideally a magical method that provides the expected outputs for each given test cases; more realistically, an engine/heuristic that can emit a pass/fail verdict over the observed test outputs.

Although it is obvious that a test execution for which we are not able to discriminate between success or failure is a useless test, and although the criticality of this problem has been very early raised in literature [85], the oracle problem has been paid little attention by research and in practice few alternative solutions still exist to human eyeballing. But such state of affairs which is already today not satisfactory, with the increasing complexity and criticality of software applications is destined to become a blocking obstacle to reliable test automation (in fact, the test oracles challenge also overlaps the route toward test automation). Indeed, the precision and efficiency of oracles greatly affects testing cost/effectiveness: we don't want that test failures pass undetected, but on the other side we don't want either to be notified of many false-positives, which waste important resources. We need to find more efficient methods for realizing and automating oracles, modulo the information which is available.

A critical survey of oracle solutions is provided by Baresi and Young [1], who conclude by highlighting areas where research progress is expected, which we borrow and expand below: *Concrete vs. abstract state and behavior:* model-based testing promises to alleviate the oracle problem, since the same model can act as the oracle; however, for oracles based on abstract descriptions of program behavior, the problem remains of bridging the gap between the concrete observed entities and the abstract specified entities; *Partiality:* plausibly partial oracles are the only viable solution to oracle automation: the challenge is to find the best trade-off between precision and cost; *Quantification:* for test oracles implemented via executable specification languages a compromise between expressiveness and efficiency must be sought, so far there is no clear optimum balance nor any fully satisfactory approach to accommodating quantifiers; *Oracles and test case selection:* ideally, oracles should be

orthogonal to test case selection; however, in model-based testing the available models are often used to derive test classes and test-class-specific test oracles together.

This challenge refers to the WHY question, in the sense of what we test against.

5.3. Dream: 100% automatic testing

Far-reaching automation is one of the ways to keep quality analysis and testing in line with the growing quantity and complexity of software. Software engineering research puts large emphasis on automating the production of software, with a bulk of modern development tools generating ever larger and more complex quantities of code with less effort. The other side of the coin is the large danger that the methods to assess the quality of the so produced software, in particular testing methods, cannot keep the pace with such software construction methods.

A large part of current testing research aims at improving the degree of attainable automation, either by developing advanced techniques for generating the test inputs (this challenge is expanded below), or, beyond test generation, by finding innovative support procedures to automate the testing process.

The dream would be a powerful integrated test environment which by itself, as a piece of software is completed and deployed, can automatically take care of possibly instrumenting it and generating or recovering the needed scaffolding code (drivers, stubs, simulators), generating the most suitable test cases, executing them and finally issuing a test report. This idea, although chimeric, has attracted followers, for instance in the early DARPA sponsored initiative for Perpetual Test (also mentioned in FOSE2000) and more recently in Saff and Ernst' Continuous Testing approach [74], which exactly aims to run tests in the background on the developer's machine while they program.

Quite promising steps have recently been made towards realization of this dream for *unit testing*, which is widely recognized as an essential phase to ensure software quality, because by scrutinizing individual units in isolation it can early detect even subtle and deeply-hidden faults which would hardly be found in system testing. Unfortunately, unit testing is often poorly performed or skipped altogether because quite expensive. We need approaches to make it feasible within the industrial development processes. A major component of unit testing high cost is the huge quantity of extra coding necessary for simulating the environment where the unit will be run, and for performing the needed functional checking of the unit outputs. To alleviate such tasks, highly successful between developers have been the frameworks belonging to the family of XUnit. Among these, the most successful is JUnit [47], which permits to automate the coding of Java test cases and their management, and has favored the spread of already mentioned test-driven development.

However such frameworks do not help with test generation and environment simulation. We would like to push automation further, as for example in the Directed Automated Random Testing (DART) approach [35], which fully automates unit testing by: automated interface extraction by static source-code analysis; automated generation of a random test driver for this interface; and dynamic analysis of program behaviour during execution of the random test cases, aimed at automatically generating new test inputs that can direct the execution along alternative program paths.

Another example is provided by the notion of "software agitation" [17], an automatic unit test technique supported by the Agitator commercial tool, which combines different analyses, such as symbolic execution, constraint solving, and directed random input generation for generating the input data, together with the already cited Daikon system [30].

Yet another approach is constituted by Microsoft Parameterized Unit Tests (PUT) [78], i.e., coded unit tests that are not fixed (as it happens for those programmed in XUnit frameworks), but depend on some input parameters. PUTs can describe abstract behavior in concise way by using symbolic execution techniques and by constraint solving can find inputs for PUTs that achieve high code coverage.

The three cited examples are certainly not exhaustive of the quite active and fruitful stage that test automation is currently enjoying. The common underlying trend that emerges is the effort to *combine* and efficiently engineering advances coming from *different types of analysis*, and this, together with the exponential increase of computational resources available, could be the really winning direction towards the 100% automation dream.

Challenge: Test input generation

Research in automatic generation of test inputs has always been very active and currently so many advanced technologies are under investigation that even devoting the whole paper just to this topic would not yield sufficient space to adequately cover it. What is dismaying is that until nowadays all such effort has produced limited impact in industry, where the test generation activity remains largely manual (as reported for instance at ISSTA 2002 Panel[7]). But finally the combination of theoretical progresses in the underlying technologies, such as symbolic execution, model checking, theorem proving, static and dynamic analyses, with technology advances in modeling industry-strength standards and with available computational power seems to make this objective closer and has revitalized researchers' faith.

The most promising results are announced to come from three directions, and especially from their mutual convergence: the already widely discussed model-based approach, "clever" ways to apply random generation, and a wealthy

variety of search-based techniques used for both white-box and black-box test generation.

Concerning *model-based test generation*, clearly research is posing great expectations in this direction since the emphasis on using (formal) models to guide testing resides exactly in the potential for automated derivation of test cases. References to ongoing work have already been provided in speaking of model-based testing challenges. Many of the existing tools are state-based and do not deal with input data. Research is needed to understand how we can incorporate data models within more traditional state-based approaches; one direction could be the introduction of symbolism over the state-based models, which could avoid the state-space explosion during test generation, and would preserve the information present in data definitions and constraints for use during the test selection process. For example, such an approach is being realized by the Symbolic Transition Systems approach [33], which augment transition systems with an explicit notion of data and data-dependent control flow. Also we need to increase the efficiency and potential of automated test generation by reusing within model-based approaches latest advances in theorem proving, model-checking and constraint satisfaction techniques. In particular, dating back to the mid 70's, symbolic execution might be considered the most traditional approach to automated test data generation. Such approach, which has been put aside for some time, because of the many underlying difficulties, is today revitalized by the resurgence of strongly-typed languages and by the development of more powerful constraint solvers; Lee and coauthors [53] survey most promising developments.

Concerning *random test generation*, this used to be considered a shallow approach, with respect to systematic techniques, deemed to be more comprehensive and capable to find important corner cases that would be likely to be overlooked by random techniques. However, previous studies mainly compared strawman implementations of random testing to sophisticated implementations of systematic techniques. Today, several researchers are proposing clever implementations of random testing that appear to outperform systematic test generation, if not else in terms of feasibility. The underlying idea of such approaches is that the random generation is improved dynamically, by exploiting feedback information collected as the tests are executed. For instance, Sen and coauthors have built on top of the cited DART approach, a notion of "concolic testing" [75], which is the combination of concrete (random) testing with symbolic execution. The concrete and symbolic executions are run in parallel and "help" each other. Instead, Pacheco and coauthors [67] randomly select a test case, execute it and check it against a set of contracts and filters.

The most promising direction then is to figure out efficient ways to combine the respective strengths of systematic (model-based) and random testing.

Finally, concerning *search-based test generation*, this consists of exploring the space of solutions (the sought test cases) for a selected test criterion, by using metaheuristic techniques that direct the search towards the potentially most promising areas of the input space. The attractive feature is that this approach appears to be fruitfully applicable to an unlimited range of problems; a recent survey is provided by McMinn [60]. Search-based test data generation is just one possible application of search-based software engineering [42].

This challenge addresses the HOW the observations are generated.

Challenge: Domain-specific test approaches

Domain-specific languages emerge today as an efficient solution towards allowing experts within a domain to express abstract specifications closer to their exigencies of expression, and which can then be automatically translated into optimized implementations. Testing as well can benefit from restricting the scope of application to the needs of a specific domain.

Research should address how domain knowledge can improve the testing process. We need to extend domain-specific approaches to the testing stage, and in particular to find domain-specific methods and tools to push test automation. Domain-specific testing could use specialized kinds of approaches, processes and tools. These in turn need to make use of customizable modeling and transformation tools, hence the challenge also overlaps the test-based modeling route.

Test techniques for specific domains have been investigated, for instance for databases, for GUI usability, for web applications, for avionics, for telecommunication systems; but few works having as their very focus the development of methodologies for exploiting domain-knowledge exist. One interesting pioneering work is due to Reyes and Richardson [71], who early developed a framework, called Siddhartha, for developing domain-specific test drivers. Siddhartha implemented an example-driven, iterative, method for developing domain-specific translators from the Test Specifications to a Domain-specific driver. It however required the tester input in the form of a general, example manually-coded driver. More recently, Sinha and Smidts have developed the HOTTest technique [76], which refers to a strongly typed domain-specific language to model the system under test and demonstrates how this permits to automatically extract and embed domain specific requirements into the test models. I believe such research efforts show promising results in demonstrating the efficiency improvements of specific domain test approaches, and hopefully further research will follow.

This challenge refers to the kind of application being observed, i.e., the WHAT question.

Challenge: On-line testing

In parallel with the traditional view of testing as an activity carried on before release to check whether a program will behave as expected, a new concept of testing is emerging around the idea of monitoring a system's behaviour in real life operation, using dynamic analysis and self-test techniques.

Actually runtime monitoring has been in use for over 30 years, but a renewed interest arises because of the increasing complexity and ubiquitous nature of software systems. Terminology is not uniform, and different terms such as monitoring, runtime testing, on-line testing are used in the literature (Delgado and coauthors [24] present a recent taxonomy). All approaches share the goal to observe the software behavior in the field, with the aims of determining whether it complies with its intended behavior, detecting malfunctions or also performance problems. In some cases an on-line recovery is attempted, in other cases the analysis is conducted off-line in order to produce a profile or to get reliability figures.

One distinguishing feature of on-line testing is that we do not need to devise a test suite to stimulate the system under test, since we limit ourselves to passively observe what happens. In fact, in communication protocol testing monitoring approaches are called *passive testing*. Message exchange along the real channels is traced, and the observed patterns are compared to the specified ones.

In principle, the inherent "passivity" of any on-line testing approaches makes them less powerful than proactive approaches. All approaches can be reconducted to verifying the observed execution traces against assertions which express desired properties, or against specification invariants. For instance, Bayse and coauthors [2] have developed a tool that support passive testing against invariants derived from FSMs; they distinguish between simple invariants and obligation invariants. More in general, on-line testing effectiveness will depend on the identification of the reference assertions. Also, the collection of traces could degrade system performance. We need to understand which are the good places and the right timing for probing the system.

In the midland between classical testing before deployment, and passive monitoring in the field, we could also conceive proactive testing in the field, i.e., actively stimulating the application after deployment, either when some events happen, for instance a component is substituted, or at scheduled intervals. A similar idea is exploited in the so-called "audition" framework [10], proposing an admission testing stage for web services.

Another issue concerns the ability to carry on the testing in the field, especially for embedded applications that must be deployed in a resource constrained environment, where the overhead required by testing instrumentation could not be feasible. An interesting new research direction has been taken by Kapfhammer et al. [49], who are developing the Juggernaut tool for testing Java applications within a constrained environment. Their original idea is to exploit execution information, so far used to tune the test suite, also for adapting the test environment (they use in particular adaptive code unloading to reduce memory requirements). Such idea is attractive and can certainly find many other useful applications.

This challenge concerns mainly the WHERE and WHEN to observe the test executions, with particular attention to dynamically evolving systems.

5.4. Dream: Efficacy-maximized test engineering

The ultimate goal of software testing research, today as it was in FOSE2000, remains that of cost-effectively engineering "practical testing methods, tools and processes for development of high quality software" [43].

All theoretical, technical and organization issues surveyed so far should be reconciled into a viable test process yielding the maximum efficiency and effectiveness (both summarized by the term efficacy). Besides, the inherent technicalities and sophistication of advanced solutions proposed by researchers should be hidden behind easy to use integrated environments. This vision makes such a challenging endeavor that we qualify it as the highest ultimate dream of software testing research.

The main obstacle to such a dream, that undermines all research challenges mentioned so far, is the growing complexity of modern systems. This complexity growth affects not only the system itself, but also the environments in which these systems are deployed, strongly characterized by variability and dynamicity.

Strategies to align the development process so to maximize testing effectiveness belong to *design for testability*. We have already mentioned testability in speaking of models and precode artifacts which can be enhanced so to facilitate testing. However, testability is a broader concept than just how the system is modelled, it also involves characteristics of the implementation, as well as of the test technique itself and its support environment. Indeed, design for testability has been attributed by practitioners to be the primary cost driver in testing [5].

Efficacy-maximized test engineering passes through many challenges, some of which are discussed here below.

Challenge: Controlling evolution

Most testing activities carried on in industry involve retesting already tested code to ascertain that changes either in the program or in the context did not adversely affect system correctness. As pointed out in FOSE2000, because of the high cost of *regression testing*, we need effective techniques to reduce the amount of retesting, to prioritize regression

test cases and to automate the re-execution of the test cases.

In general, we need strategies to scale up regression testing to large composite systems. We have already discussed theoretical issues behind compositional testing (see Section 5.1); we also need practical approaches to regression testing global system properties as some system parts are modified. For instance, given a component which is originally designed with an architecture, we need to understand how to test whether this piece of software evolves in line with its architecture. Such problem is also central to testing of *product families*.

A related idea is *test factoring*, which consists into converting a long-running system test into a collection of many small unit tests. These unit tests can exercise a small part of the system in exactly the same way that the system testing did, but being more focused they can pinpoint failures in specific selected parts of the system. Test factoring is today actively investigated [73, 65, 28] since it promises order-of-magnitude improvements in execution of regression tests.

A common basic assumption of many existing approaches to regression testing is that a software artifact can be assumed for reference, for instance the requirements specification, or the software architecture. For modern software applications which continuously evolve, often dynamically in ways we cannot know in advance, neither such precode artifacts nor even the source code itself might be available, and the testing paradigm is shifting from regression testing towards one of continuous testing at runtime.

Hence, a crucial issue concerns how to maintain control of the quality of software which evolves dynamically in the field. We need to understand what is the meaning of regression testing in such an evolutive context, and how we can modify and extend the basic idea of selective regression testing, i.e., how often do we need to check the execution traces? how can we compare the traces taken in different temporal intervals and understand if the evolution did not bring any malfunctions?

This challenge concerns broadly the WHAT, WHERE and WHEN do we replay some executions following software evolution.

Challenge: Leveraging user population and resources

We have already mentioned the emerging trend of continuous validation after deployment, by means of on-line testing approaches (see Section 5.3), when traditional off-line testing techniques become ineffective. Since software intensive systems can behave very differently in varying environments and configurations, we need practical ways to scale up on-line testing to cover the broad spectrum of possible behaviors. One rising approach to address this challenge is to augment in-house quality assurance activities by using data dynamically collected from the field. This is promising in that it can help to reveal real usage spectra and expose real problems on which to focus the testing activities and on which testing is lacking. For instance, by giving each user a different default configuration, the user base can be leveraged to more quickly expose configurations conflicts or problems, such as in [87]. And fielded profiles can also be used for improving a given test suite, such as in [56, 64, 29]. Although also some commercial initiatives start to appear, such as Microsoft's Customer Experience Improvement Program [61], these efforts are still in their infancy, and one important research challenge left open is how to define efficient and effective techniques to unleash the potential represented by a large number of users, running similar applications, on interconnected machines. This high-level challenge involves several more specific challenges, among which:

- How can we collect runtime data from programs running in the field without imposing too much overhead?

- How can we store and mine the collected (potentially huge) amount of raw data so to effectively extract relevant information?

- How can we effectively use the collected data for augmenting and improving in-house testing and maintenance activities?

This challenge proposes that the users instantiate the WHERE and WHEN to scrutinize software runs.

Challenge: Testing patterns

We have already mentioned under the dream of a useful test theory, that we need to understand the relative effectiveness of test techniques in the types of faults they target. To engineering the test process, we need to collect evidences for such information to be able to find the most effective pattern for testing a system. This is routinely done, when for instance functional testing based on the requirements is combined with measures of code coverage adequacy. Another recurring recommendation is to combine operational testing with specific verification of special case inputs. However, such practices need to be backed up by a systematic effort to extract and organize recurring and proved effective solutions to testing problems into a catalogue of *test patterns*, similarly to what is now a well established scheme for design approaches.

Patterns offer well proven solutions to recurring problems, or, in other words, they make explicit and document problem-solving expertise. As testing is recognized as expensive and effort-prone, making explicit which are successful procedures is highly desirable.

A related effort is Vegas and coauthors characterization schema of how test techniques are selected [82]. They surveyed the type of knowledge that practitioners use to choose the testing techniques for a software project and have produced a formalized list of the relevant parameters. However, organizations that use the proposed schema might not dispose of all required information, hence more recently they are also investigating the sources of information, and

how these sources are to be used [82]. Similar studies are needed to formalize and document successful practices for any other testing-related activity.

In fact this challenge spans over all six questions.

Challenge: Understanding the costs of testing

Since testing does not take place in abstract, but within the concrete world, with its risks, and safety as well as economic constraints, ultimately we want to be able to link the testing process and techniques with their cost.

Each and every research article on software testing starts from claiming that testing is a very expensive activity, but we lack up-to-date and reliable references; it is somehow dismaying that still today references to quantify the high cost of testing cite textbooks dating back to more than twenty years ago. This might admittedly be due to the sensibility of failure data, which are company confidential. Nonetheless, to usefully transfer research advances to practice we need to be able to quantify direct and indirect costs of software testing techniques.

Unfortunately, most research in software testing takes a value-neutral position, as if every fault found is equally important or has the same cost, but this is of course not true; we need ways to incorporate economic value into the testing process, to help test managers apply their judgement and select the most appropriate approaches. Boehm and colleagues have introduced the *value-based software engineering* (VBSE) paradigm [12], in which quantitative frameworks to support software managers decisions are sought that enhance the value of delivered software systems. In particular, various aspects of software quality assurance have been investigated including value-based and risk-based testing, e.g., [13]. VBSE concerns mainly the management of processes, for instance, w.r.t. testing, different types of stakeholder utility functions are considered to trade-off time of delivery vs. market value. We would also need to be able to incorporate estimation functions of the cost/effectiveness ratio of available test techniques. The key question is: given a fixed testing budget, how should it be employed most effectively?

This challenge clearly addresses mainly the HOW and HOW MUCH of testing.

Challenge: Education of software testers

Finally, for software testing as for any other software engineering activity, a crucial resource remains the human factor. Beyond the availability of advanced techniques and tools and of effective processes, the testers' skill, commitment and motivation can make a big difference between a successful test process or an ineffective one. Research on its side should strive for producing engineered effective solutions that are easily integrated into development and do not require deep technical expertise. But we also need to work in parallel for empowering the human potential. This is done by both education and motivation. Testers should be educated to understand the basic notions of testing and the limitations and the possibilities offered by the available techniques. While it is research that can advance the state of the art, it is only by awareness and adoption of those results by the next-coming generation of testers that we can also advance the state of practice. Education must be continuing, to keep the pace with the advances in testing technology. Education by itself poses several challenges, as discussed in [54].

It is evident that education must cover all characterizing aspects of testing.

5.5. Transversal challenges

By transversal challenges we identify some research trends that go through all the four identified dreams. In particular we discuss here two transversal challenges.

Challenge: Testing within the emerging development paradigm

The history of software engineering research is phased by the subsequent emerging of novel paradigms of development, which promise to release higher quality and less costly software. Today, the fashion is Service-oriented Computing and many interesting challenges emerge for the testing of service-oriented applications.

Several similarities exist with CB systems, and as in CB testing, services can be tested from different perspectives, depending on who is the involved stakeholder [23]. The service developer, who implements a service, the service provider, who deploys and makes it available, and the service integrator, who composes services possibly made available by others, access different kinds of information and have different testing needs. Except for the service developer, black-box test techniques need to be applied, because design and implementation details of services are not available.

One peculiar aspect of services is that they are forced to make available a standard description in computer processable format to enable search and discovery. So, given that this is often the only information available for analysis, research is investigating how to exploit this compulsory specification for testing purposes. Currently, this description only includes the service interface in terms of the signature of methods provided (for instance the WSDL definition for Web Services). Clearly method signatures provide poor expressiveness for testing purposes, and in fact researchers aim at enriching such description to allow for more meaningful testing.

Towards promoting interoperability, a first concern is to ensure that the services comply with established standardized protocols for message exchange. For instance, guidelines have been released by the WS-I (Web Services-Interoperability) organization, along with testing tools to

monitor and check that the message exchanged comply with the guidelines. Such an approach is certainly necessary to assure interoperability, but not sufficient, in particular it does not test dynamic behaviour and is not concerned with the verification of extra-functional properties.

Conceptually, off-line and on-line testing of services can be distinguished. With regard to off-line testing, the two approaches that emerge are model-based and mutation. For model-based testing of services, the general idea is to assume that the service developers and/or integrators make available models suitable for the automatic derivation of test cases. We need then to adapt the wealth of existing approaches to model-based testing to the context and constraints of services, and several proposals are being made, e.g. [55, 32, 50]. Mutation strategies adapted to services instead foresee the mutation of the input messages, as presented in [23].

On-line testing (discussed in 5.3), assumes a special importance for testing of services, since monitoring the real-world execution is the only way to observe the application behaviour. A service-oriented application generally results from the integration of several services, controlled and owned by different organizations. As a result, the control of an application is distributed, and the services composing it discover each other at run time and can change without prior notice, so nobody can predict the receiver or the provider of a given service. Monitoring of services introduces many subtle problems, relative to performance degradation, production of undesired side effects and cost. We need to understand on one side how we can observe the execution in a distributed network without (too) negatively affecting the system performance; on the other, we need means to reason at the abstract level of service composition and understand what and when we need to check.

Challenge: Coherent testing of functional and extra-functional properties

By far the bulk of software testing literature addresses functionality testing, i.e., checking that the observed behaviour complies with the logic of the specifications. But this is not enough to guarantee the real usefulness and adequacy to purpose of the tested software: as importantly, well-behaving software must fulfill extra-functional properties, depending on the specific application domain. Notably, while conventional functionality testing does not provide for any notion of time, many features of the exhibited behaviour of a piece of software can depend on *when* the results are produced, or on *how long* they take to be produced. Similarly, while functionality testing does not tackle resource usage and workloads, in specific domains, such as telecommunications, performance issue account for a major fault category [84].

We would like to have test approaches to be applied at development time that could provide feedback as early as possible. On-going research that can pave the way is not much, and adopted approaches can be classified into model-based and genetic. Among the former, we need effective ways to enhance models with desired extra-functional constraints. In this direction, researchers from Aalborg University [51] have been long investigating the extension of existing conformance testing theory to timed setting, producing a tool that can generate test cases from Timed Automata and execute them monitoring the produced traces.

Model-based approaches are certainly an important instrument also for real-time embedded systems, but they will probably take a long course before being akin to large-scale application, also in view of the many technical issues that need to be modelled, such as environment dependency, distribution, resource constraints. It is thus advisable to look in parallel for innovative approaches: for instance, Wegener and Grochtmann [83] have proposed the use of evolutionary algorithms. They reduce real-time testing to the optimization problem of finding the best-case and the worst-case values of execution time. Such idea could be extended to other extra-functional properties, by appropriately translating the constraint into an optimization problem.

6. Conclusions

We believe that software testing is a lively, difficult and richly articulated research discipline, and hope that this paper has provided a useful overview of current and future challenges. Covering into one article all ongoing and foreseen research directions is impossible; we have privileged broadness against depth, and the contribution of this paper should be seen rather as an attempt to depict a comprehensive and extensible roadmap, in which any current and future research challenge for software testing can find its place. The picture which emerges must be taken as a work-in-progress fabric that the community may want to adapt and expand.

It is obvious that those goals in such roadmap which have been settled as the dreams are destined to remain so. However, in a research roadmap the real thing is not the label on the finish, but the pathways along the traced routes. So, what actually is important that researchers focus on to sign progress are those called the challenges, and certainly the roadmap provides plenty of them, some at a more mature stage, other just beginning to appear.

What is assured is that software testing researchers do not risk to remain without their job. Software testing is and will continue to be a fundamental activity of software engineering: notwithstanding the revolutionary advances in the way it is built and employed (or perhaps exactly because of), the software will always need to be eventually tried and monitored. And as extensively discussed in this paper, for sure we will need to make the process of testing more ef-

fective, predictable and effortless (which coincides with the ultimate of the four testing dreams).

Unfortunately, progress may be slowed down by fragmentation of software testing researchers into several disjoint communities: for instance, different events have been established by the communities as the loci where to meet to discuss the latest results, such as the *ACM International Symposium on Software Testing and Analysis* (ISSTA), or the *IFIP International Conference on the Testing of Communicating Systems* (TESTCOM) events, just to cite a couple, showing little overlap between PC members, participation, mutual knowledge and citations (which is a pity). In addition to scientific challenges faced by testing research, which have been discussed in Section 5, then we would like to also rise a challenge, which is opportunistic: the time has come that the different existing test research communities eventually converge and reconcile the respective achievements and efforts, since this would certainly be of the greatest benefit to advance the state of art[1].

A necessary concluding remark concerns the many fruitful relations between software testing and other research areas. By focussing on the specific problems of software testing, we have in fact overlooked many interesting opportunities arising at the border between testing and other disciplines. Some have been just touched upon in this paper, for instance model-checking techniques, see [27] (e.g., to drive model-based testing), or the use of search-based approaches, see [42], for test input generation, or the application of test techniques to assess performance attributes, see [86]. We believe that really many are the openings that may arise from a more holistic approach to software testing research, and in [19] readers can certainly find and appreciate many new interesting synergies spanning across the research disciplines of software engineering.

7. Acknowledgements

Summarizing the quite broad and active field of software testing research has been a tough challenge. While I remain the only responsible of imprecisions or omissions, there are lot of people whom I am indebted. First, with the aim of being as comprehensive and unbiased as possible, I asked several colleagues to send me both a statement of what they considered the topmost outstanding challenge faced by software testing research, and a reference to relevant work (indifferently other authors' or their own paper) that this paper could not miss to cite. Out of the many I invited, I warmly thank for contributing: Ana Cavalli, S.C. Cheung, Sebastian Elbaum, Mike Ernst, Mark Harman, Bruno Legeard, Alex Orso, Mauro Pezzè, Jan Tretmans, Mark Utting, Margus Veanes; their contributions have been edited and incorporated in the paper. I would also like to thank Lars Frantzen, Eda Marchetti, Ioannis Parissis, and Andrea Polini for the discussion on some of the presented topics. Daniela Mulas and Antonino Sabetta helped with drawing the Roadmap figure. I would also like to sincerely thank Lionel Briand and Alex Wolf for inviting me and for providing valuable advice.

This work has been partially supported by the Marie Curie TAROT Network (MRTN-CT-2004-505121).

[1] Notably, among its goals the current Marie Curie TAROT Network http://www.int-evry.fr/tarot/ has that of joining researchers from the software testing and protocol testing communities.

References

[1] L. Baresi and M. Young. Test oracles. Technical report, Dept. of Comp. and Information Science, Univ. of Oregon, 2001. http://www.cs.uoregon.edu/m̃ichal/pubs/oracles.html.

[2] E. Bayse, A. R. Cavalli, M. Núñez, and F. Zaïdi. A passive testing approach based on invariants: application to the wap. *Computer Networks*, 48(2):235–245, 2005.

[3] B. Beizer. *Software Testing Techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[4] A. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In *[21]*.

[5] S. Berner, R. Weber, and R. Keller. Observations and lessons learned from automated testing. In *Proc. 27th Int. Conf. on Sw. Eng.*, pages 571–579. ACM, 2005.

[6] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 1991.

[7] A. Bertolino. ISSTA 2002 Panel: is ISSTA research relevant to industrial users? In *Proc. ACM/SIGSOFT Int. Symp. on Software Testing and Analysis*, pages 201–202. ACM Press, 2002.

[8] A. Bertolino and E. Marchetti. Software testing (chapt.5). In P. Bourque and R. Dupuis, editors, *Guide to the Software Engineering Body of Knowledge SWEBOK, 2004 Version*, pages 5–1–5–16. IEEE Computer Society, 2004. http://www.swebok.org.

[9] A. Bertolino, E. Marchetti, and H. Muccini. Introducing a reasonably complete and coherent approach for model-based testing. *Electr. Notes Theor. Comput. Sci.*, 116:85–97, 2005.

[10] A. Bertolino and A. Polini. The audition framework for testing web services interoperability. In *Proc. EUROMICRO '05*, pages 134–142. IEEE, 2005.

[11] A. Bertolino, A. Polini, P. Inverardi, and H. Muccini. Towards anti-model-based testing. In *Proc. DSN 2004 (Ext. abstract)*, pages 124–125, 2004.

[12] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Gruenbacher, editors. *Value-Based Software Engineering*. Springer-Verlag, Heidelberg, Germany, 2006.

[13] S. Biffl, R. Ramler, and P. Gruenbacher. Value-based management of software testing. In *[12]*.

[14] R. V. Binder. *Testing Object-Oriented Systems Models, Patterns, and Tools*. Addison Wesley Longman, Inc., Reading, MA, 2000.

[15] C. Blundell, D. Giannakopoulou, and C. S. Pasareanu. Assume-guarantee testing. In *Proc. SAVCBS '05*, pages 7–14. ACM Press, 2005.

[16] G. V. Bochmann and A. Petrenko. Protocol testing: review of methods and relevance for software testing. In *Proc. ACM/SIGSOFT Int. Symp. Software Testing and Analysis*, pages 109–124, 1994.

[17] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. ACM/SIGSOFT Int. Symp. Software Testing and Analysis*, pages 169–180. ACM Press, 2006.

[18] L. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Trans. Softw. Eng.*, 29(7):594– 607, 2003.

[19] L. Briand and A. Wolf, editors. *Future of Software Engineering 2007*. IEEE-CS Press, 2007.

[20] L. C. Briand, Y. Labiche, and M. M. Sówka. Automated, contract-based user testing of commercial-off-the-shelf components. In *Proc. 28th Int. Conf. on Sw. Eng.*, pages 92–101. ACM Press, 2006.

[21] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems - Advanced Lectures, LNCS 3472*. Springer Verlag, 2005.

[22] G. Canfora and M. Di Penta. The forthcoming new frontiers of reverse engineering. In *[19]*.

[23] G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, March/April 2006.

[24] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872, 2004.

[25] E. Dijkstra. Notes on structured programming. Technical Report 70-WSK03, Technological Univ. Eindhoven, 1970. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[26] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Eng.*, 10(4):405–435, 2005.

[27] M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, and W. Visser. Formal software analysis : Emerging trends in software model checking. In *[19]*.

[28] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proc. 14th ACM/SIGSOFT Int. Symp. on Foundations of Sw Eng.*, pages 253–264. ACM Press, 2006.

[29] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Softw. Eng.*, 31(4):312–327, 2005.

[30] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, to appear.

[31] P. Frankl and E. Weyuker. Provable improvements on branch testing. *IEEE Trans. Softw. Eng.*, 19(10):962–975, 1993.

[32] L. Frantzen, J. Tretmans, and R. d. Vries. Towards model-based testing of web services. In *Proc. Int. Workshop on Web Services - Modeling and Testing (WS-MaTe2006)*, pages 67–82, 2006.

[33] L. Frantzen, J. Tretmans, and T. Willemse. A symbolic framework for model-based testing. In *Proc. FATES/RV*, LNCS 4262, pages 40–54. Springer-Verlag, 2006.

[34] M.-C. Gaudel. Formal methods and testing: Hypotheses, and correctness approximations. In *Proc. FM 2005, LNCS 3582*, pages 2–8. Springer-Verlag, 2005.

[35] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN PLDI'05*, pages 213–223, 2005.

[36] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Softw. Eng.*, 1(2):156–173, June 1975.

[37] R. Gotzhein and F. Khendek. Compositional testing of communication systems. In *Proc. IFIP TestCom 2006, LNCS 3964*, pages 227–244. Springer Verlag, May 2006.

[38] W. Grieskamp. Multi-paradigmatic model-based testing. In *Proc. FATES/RV*, pages 1–19. LNCS 4262, August 15-16, 2006.

[39] D. Hamlet. Subdomain testing of units and systems with state. In *Proc. ACM/SIGSOFT Int. Symp. on Software Testing and Analysis*, pages 85–96. ACM Press, 2006.

[40] D. Hamlet, D. Mason, and D. Woit. Theory of software reliability based on components. In *Proc. 23rd Int. Conf. on Sw. Eng.*, pages 361–370, Washington, DC, USA, 2001. IEEE Computer Society.

[41] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng.*, 16(12):1402–1411, 1990.

[42] M. Harman. The current state and future of search-based software engineering. In *[19]*.

[43] M. J. Harrold. Testing: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 61–72. IEEE Computer Society, 2000. In conjunction with ICSE2000.

[44] W. Hetzel. *The Complete Guide to Software Testing, 2nd Edition*. QED Inf. Sc., Inc., 1988.

[45] W. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. Softw. Eng.*, SE-2(3):208– 215, 1976.

[46] D. Janzen, H. Saiedian, and L. Simex. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, Sept. 2005.

[47] JUnit.org. http://www.junit.org/index.htm.

[48] N. Juristo, A. M. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. *Empirical Softw. Eng.*, 9(1-2):7–44, 2004.

[49] G. M. Kapfhammer, M. L. Soffa, and D. Mosse. Testing in resource constrained execution environments. In *Proc. 20th IEEE/ACM Int. Conf. on Automated Software Engineering*, Long Beach, California, USA, November 2005. ACM Press.

[50] C. Keum, S. Kang, I.-Y. Ko, J. Baik, and Y.-I. Choi. Generating test cases for web services using extended finite state machine. In *Proc. IFIP TestCom 2006, LNCS 3964*, pages 103–117. Springer Verlag, 2006.

[51] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proc. 5th ACM Int. Conf. on Embedded Softw.*, pages 299–306. ACM Press, 2005.

[52] Y. Le Traon, B. Baudry, and J.-M. Jézéquel. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng.*, 32(8):571–586, 2006.

[53] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam. Using symbolic execution to guide test generation: Research articles. *Softw. Test. Verif. Reliab.*, 15(1):41–61, 2005.

[54] T. C. Lethbridge, J. Daz-Herrera, R. J. LeBlanc., and J. Thompson. Improving software practice through education: Challenges and future trends. In *[19]*.

[55] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang. BPEL4WS unit testing: Framework and implementation. In *Proc. of ICWS'05*, pages 103–110, 2005.

[56] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 141–154. ACM Press, 2003.

[57] M. Lyu. Software reliability engineering: A roadmap. In *[19]*.

[58] M. Lyu (ed.). *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, and IEEE CS Press, Los Alamitos, 1996.

[59] L. Mariani and M. Pezzè. Dynamic detection of COTS components incompatibility. *IEEE Software*, to appear.

[60] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, Sept. 2004.

[61] Microsoft Research. Customer experience improvement program, 2006. http://www.microsoft.com/ products/ceip/.

[62] E. F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, pages 129–153, 1956.

[63] NIST. The economic impacts of inadequate infrastructure for software testing, May 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[64] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. Joint meeting of the European Soft. Eng. Conf. and ACM/SIGSOFT Symp. on Foundations of Soft. Eng. (ESEC/FSE'03)*, pages 128–137, 2003.

[65] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proc. 3rd Int. ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, may 2005.

[66] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[67] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. Technical Report MSR-TR-2006-125, Microsoft Research, Redmond, WA.

[68] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, 2003.

[69] J. Poore, H. Mills, and D. Mutchler. Planning and certifying software system reliability. *IEEE Software*, pages 87–99, Jan. 1993.

[70] M. J. Rehman, F. Jabeen, A. Bertolino, and A. Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, to appear.

[71] A. Reyes and D. Richardson. Siddhartha: a method for developing domain-specific test driver generators. In *Proc. 14th Int. Conf. on Automated Software Engineering*, pages 81 – 90. IEEE, 12-15 Oct. 1999.

[72] M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Simulation-based test adequacy criteria for distributed systems. In *Proc. 14th ACM/SIGSOFT Int. Symp. on Foundations of Sw Eng.*, pages 231–241. ACM Press, 2006.

[73] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. 20th Int. Conf. on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, November 9–11, 2005.

[74] D. Saff and M. Ernst. An experimental evaluation of continuous testing during development. In *Proc. ACM/SIGSOFT Int. Symp. on Software Testing and Analysis*, pages 76–85. ACM, July, 12-14 2004.

[75] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Joint meeting of the European Soft. Eng. Conf. and ACM/SIGSOFT Symp.on Foundations of Soft. Eng. (ESEC/FSE'05)*, pages 263–272. ACM, 2005.

[76] A. Sinha and C. Smidts. HOTTest: A model-based test design technique for enhanced testing of domain-specific applications. *ACM Trans. Softw. Eng. Methodol.*, 15(3):242–278, 2006.

[77] D. Sjøberg, T. Dybå, and M. Jørgensen. The future of empirical methods in software engineering research. In *[19]*.

[78] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Softw.*, 23(4):38–47, 2006.

[79] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts and Tools*, 17:103–120, 1996.

[80] M. Utting and B. Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan and Kaufmann, 2006.

[81] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In *Proc. FATES 2003, LNCS 2931*, 2003.

[82] S. Vegas, N. Juristo, and V. Basili. Packaging experiences for improving testing technique selection. *The Journal of Systems and Software*, 79(11):1606–1618, Nov. 2006.

[83] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Syst.*, 15(3):275–298, 1998.

[84] E. Weyuker and F. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Soft. Eng.*, 26(12):1147–1156, 2000.

[85] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[86] M. Woodside, G. Franks, and D. Petriu. The future of software performance engineering. In *[19]*.

[87] C. Yilmaz, A. M. A. Porter, A. Krishna, D. Schmidt, A. Gokhale, and B. Natarajan. Preserving distributed systems critical properties: a model-driven approach. *IEEE Software*, 21(6):32–40, 2004.

[88] H. Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Trans. Softw. Eng.*, 22(4):248–255, 1996.

[89] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[90] H. Zhu and X. He. A theory of behaviour observation in software testing. Technical Report CMS-TR-99-05, 24, 1999.