Model-Based Testing

Ina Schieferdecker
Andreas Hoffmann
Competence Center Modeling and Testing, Fraunhofer Institute for Open Communication Systems (FOKUS), Berlin, Germany

Abstract

Model-Based Testing (MBT) constitutes a number of technologies, methods, and approaches, with the aim of improving the quality, efficiency, and effectiveness of test processes, tasks, and artifacts. Started as a pure academic field of application, it has gained significance for industrial domains in recent years. Moreover, the ongoing adoption of Model-Based Engineering techniques by industrial-grade software engineering companies provides a solid basis to introduce and apply MBT approaches and hopefully will lead to a larger acceptance of model-based techniques. This entry presents background information on testing in general, motivates the application of MBT approaches in particular, and reviews related work and standards.

INTRODUCTION

Software-intensive systems are playing an ever-increasing role in business and industrial products such as cars, trains, manufacturing machines, industrial automation systems, mobile phones, and financial computer-supported systems. The complexity of such systems is increasing both in terms of code and algorithms as well as their interconnectivity. Furthermore, the qualitative importance of the software embedded in these products is growing because it is taking over more and more essential controlling (sometimes safety-relevant) functions as in driving assistance and engine management in vehicles, or controlling functions in a milling machine. Since software is taking over more important functions, failures occurring in this software could potentially lead to disastrous effects, at worst possibly endangering human life or the environment, for example, if an autopilot is not working as expected. Even in less extreme cases, failures can lead to a financial disaster.

With this in mind, an effective quality assurance mechanism supporting the software development is indispensable. One of the most important means of quality assurance is testing, since it is the only proof under real circumstances. Unfortunately, the costs for testing are still taking a huge percentage of the overall development costs, i.e., 30–40%.

In industrial domains such as automotive, automation, telecommunication, and financials, model-based development is already partly in use. For example, Simulink/Stateflow,[1] UML,[2,3] Labview,[4] and BPML[5] models are already in use in some of the industrial domains named above. In the business domain, mainly large-scale business process models are used to describe the business and IT services. In the automotive domain, UML and Simulink/Stateflow models are used. The telecommunication and railway domain often refer to UML models and in the industrial automation domain, UML, Simulink/Stateflow, and Labview models are used. Moreover, other proprietary (partly self-developed) modeling languages can be found in several industrial domains. Thus, model-based development is, in industry as well as in academia, clearly seen as a means of making the development of software-based systems more effective, reliable, and maintainable.

The situation with Model-Based Testing (MBT) is somewhat different. Although concepts and tool prototypes have been available in the academic world for many years, industrial-grade tools are less often found and are rarely used in industrial-grade processes. Nevertheless, MBT can be seen as an efficient way to reduce the efforts and costs for testing. The increasing acceptance for model-based development processes resulting in a stronger formalization of development artifacts yields a promising basis to introduce MBT approaches on a larger scale. Last but not least, success stories at Motorola[6,7] and the availability of mature tools[8] have shown that MBT approaches are now applicable to industry-grade test processes.

BACKGROUND

Modeling and testing are research and industrial activities that are both self-standing domains with their own

Encyclopedia of Software Engineering DOI: 10.1081/E-ESE-120044686

terminologies, but they are also closely related or even dependent on one another. Many experienced experts are concentrating their work on one field only. Due to the purpose and focus of this study on testing, this entry provides background information on testing and quality in general and on the test process in particular.

General Considerations on Testing

Software testing is used in association with verification and validation (V&V). *Verification* is the checking or testing of items, including software, for conformance and consistency with an associated specification. Software testing is one kind of verification, which also uses techniques such as reviews, inspections, and walkthroughs. *Validation* is the process of checking that what has been specified is what the user actually wanted.

Testing

Software testing is one of the most important analytical quality assurance methods. Essential to a good test quality is the systematic design of test cases. The test cases defined decide about the kind and scope of the test. In most cases, test models and test case designs are difficult to automate, but MBT is one of the most promising approaches for addressing this problem. In MBT, test cases can be automatically derived from a system model to be tested. This approach is supported by a number of methodologies and tools dealing with the creation and generation of system models, simulation of those models, creation and/or generation and execution of test suites, etc. Because implementation changes might as well be captured in the model, MBT reduces test maintenance costs, and developers only have to regenerate the test in order to have the changes affect all tests. The MBT tools enhance team communication because the model, test suite, and trace provide a clear and unified view of both the System Under Test (SUT) and the test.

Model-based testing

A model is usually an abstract, partial representation of the system under the test's desired behavior. The test cases derived from this model are functional tests on the same level of abstraction as the model. These test cases are collectively known as the abstract test suite. For a SUT, various system models might exist, such as

- requirements models,
- information models,
- workflow models,
- architectural models,
- behavioral models,
- configuration models,
- deployment models,
- performance models,
- risk models,
- environment models, and
- usage models.

Model-based test methods (see Refs. [9, 10]) differ in the system model being considered, the methods taken for test generation, and finally, the way the test results are being obtained. The system models are classified into aspects of the subject being considered, the redundancy being contained in, the behavioral characteristics being reflected, and finally the behavioral paradigms being used.

There are many different ways to "derive" tests from a model. Because testing is usually experimental and based on heuristics, there is no one best way to do this. It is common to consolidate all test derivation-related design decisions into a package that is often known as "test requirements," "test purpose," or even "use case." This package can contain, e.g., information about the part of the model that should be the focus for testing, or about the conditions where it is correct to stop testing. Because test suites are derived from models and not from source code, MBT is usually seen as one form of black-box testing. In some aspects, this is not completely accurate. MBT can be combined with source-code-level test coverage measurement, and functional models can be based on existing source code in the first place.

Especially in Model-Driven Engineering or in OMG's (Object Management Group) MDA (Model-Driven Architecture), the model is built before or parallel to the development process of the SUT. Recent work outlines how to combine system development and testing along this MDA paradigm.

The effectiveness of MBT is primarily due to the potential for automation to increase effectiveness and efficiency. This is typically guided by test selection criteria that also serve as termination criteria for testing. If the system model is machine-readable and formal to the extent that it has a well-defined behavioral interpretation, test cases can in principle be derived mechanically. Often, the system model is translated to or interpreted as a finite-state automaton or a state transition system. To find test cases, the automaton is searched for executable paths. A possible execution path can be the basis for a test case—extended, e.g., by timers, verdicts, and defaults to cover unexpected responses. Depending on the complexity of the SUT and the corresponding model the number of paths can be very large, because of the huge amount of possible behaviors of the system. For finding appropriate test cases, i.e., paths that refer to a certain requirement to check, the search of the paths has to be guided.

Integrated test development

Software testing can be implemented at any time in the development process; however, the main part of the testing activities occurs after the requirements have been defined and the coding process has been completed. Often, due to delays in the development cycle, the testing cannot start at the proper schedule and due to the frequent change in requirements and poor documentation, a testing team will often not be able to reach its estimated goals.

To avoid such problems, testing activities (e.g., test planning, test specification, test implementation, test execution, and test evaluation) have to be well planned and seamlessly aligned with the system development activities. Existing process model like the V-Model[11] and the Rational Unified Process[12] do cover already main test activities. An enhancement of the traditional V-Model, the so-called W-Model,[13] explicitly models the relationship between development activities and testing activities.

The W-Model is a process model for software development processes. It is based on the far-spread V-Model. Besides the main development activities (requirement definition, functional and technical system model, and component specification), the W-Model especially focuses test activities. The test activities start early in the development process and are directly tied to the development activities. Developers with specialized knowledge in the field of quality assurance and particularly testing are directly involved in the individual development activities. Thus, they are able to influence the system specification with respect to testability and maintainability and can align the test activities with their related system development activities. Fig. 1 shows an interpretation of the W-Model for system development process.

Besides the heavyweight integration of testing activities by means of large-scale process models (see above), in recent years, certain lightweight integration strategies have been developed. Especially in the field of agile development,[14] test-driven approaches[14,15] have become more and more important.

The seamless integration of development activities and testing activities is one of the big challenges of model-based development. On the one hand, the use of formal or semiformal models for system development and testing allows a more fine-grained integration of the development artifacts. The use of standardized modeling tools (e.g., UML Tools) in combination with integrated model repositories supports traceability, integrated versioning, and fine-grained maintenance of development artifacts in such a way that was not possible some years ago. On the other hand, the complexities of such repositories, especially in distributed development scenarios, actually blow up the capabilities of most of the industrial-grade development tools. Managing the complexity of distributed model-based development processes, model interchange, and maintenance of models over the complete software life cycle is one of the big research areas currently addressed by companies like IBM, Microsoft, etc.

Test strategies

Testing software is a complex process. Hence, international standards and best practice recommendation agreed on a set of terms, activities, and strategies that allow the definition of systematic test approaches. Basically, most specialists seem to agree with Weyuker[11] that at least three stages of correctness testing are absolutely necessary for reliable software-based systems:

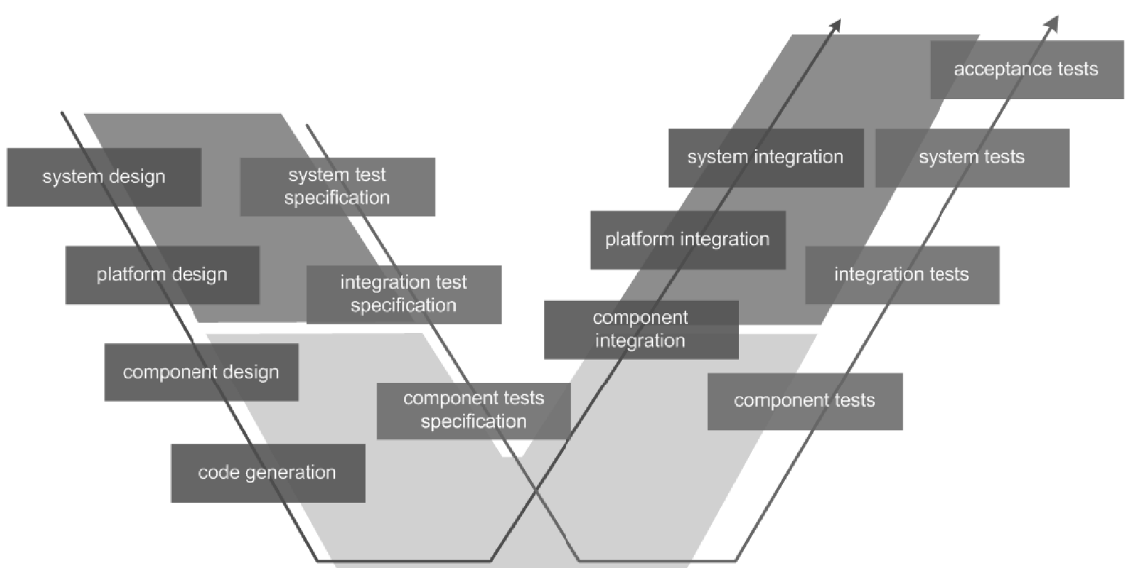

Fig. 1 W-Model for system development process.

- **Unit/component testing**, in which individual components or software modules are tested.
- **Integration testing**, in which the subsystems formed by integrating the individually tested components are tested as an entity.
- **System testing**, in which the functionalities of the SUT would be verified in a "real-world" scenario. This might include not only functionality tests but also non-functional tests, like load tests as well as performance tests, if such requirements are set on the system.

Moreover, functional and non-functional testing are distinguished. Whereas functional testing directly relates to the functional specification of a software system, non-functional testing addresses the non-functional properties of the SUT. Special methods exist for both functional and non-functional approaches. Functional approaches consider the coverage of the functional specification (specification-based testing) or the coverage of the input data domain of a system. Functional tests are usually systematically derived by methods like equivalence partitioning, boundary value analysis, all-pairs testing, etc.

To test non-functional aspects of software, the following methods are used:

- Performance testing checks whether the SUT can handle large quantities of data or users. This is generally referred to as scalability.
- Stability or load testing checks whether the software is working well in a defined period of time. This activity is as well referred to as load testing.
- Usability testing checks whether the user interface conforms to usability standards (i.e., is easy to use and understandable).
- Security testing is essential for software that processes confidential data to prevent system intrusion and checks data integrity, confidentiality, authorization, availability, and non-repudiation.

Concerning the availability of information on the SUT, black-box testing and white-box testing are distinguished. Both form actually the most important techniques to ensure software quality. Moreover, grey-box testing is a mix of both and especially supports the test designer with additional information on the SUT during the test specification process.

- *Black-box* testing is a test case design technique in which test cases are derived and selected based on an analysis of the specification of the functionality of a component or system without reference to its internal structure. It evaluates the outputs of a SUT in response to stimuli sent by a test system. Black-box testing cannot guarantee that all parts of the implementation have been tested. Instead, it discovers faults of omission, indicating that part of the specification has not been fulfilled. The target of black-box testing is to cover the functionality of the test object as thoroughly as possible. Black-box testing refers to methods like equivalence partitioning, boundary value analysis, all-pairs testing, random testing, and specification-based testing.
- *White-box* testing is a testing technique performed on the internal structure of the component or system. Its basis is to cover the structure of the test object as thoroughly as possible. White-box testing does not guarantee that the complete specification has been implemented and is much more expensive than black-box testing. It requires the source code to be produced before the tests can be planned and is much more laborious in the determination of suitable input data and the determination if the software is or is not correct. The following methods for white-box testing exist: API (application programmers' interface) testing, code coverage testing, fault injection methods, mutation testing, etc.
- *Grey-box* testing involves having access to internal data structures and algorithms for the purpose of designing test cases. The test execution itself is carried out with pure black-box approaches. In order to fully test a software product, both black- and white-box testings are required.

Integration strategies. Integration testing is the phase of testing where individual software components/modules are combined and tested in an assembly. This phase of integration testing takes components/modules as input that already have been tested, aggregates them, applies tests especially designed for integration issues, and delivers as its output the integrated system ready for system testing. Similar to unit or module testing, during the integration, the missing components have to be replaced by dummies (for missing message/service consumers) and test drivers (for missing message/service providers).

For integration testing, different integration strategies are possible. In general, two different approaches are distinguished:

- Vertical integration addresses the composition of entities that obey a hierarchically ordered structure (e.g., inheritance structure, decomposition of large systems in subsystem, and well-defined communication relationships with a clear consumer–provider relationship).
- Horizontal integration addresses the integrated components as loosely, non-hierarchically coupled entities like objects in an object-oriented environment. The interdependencies between these components are often not specified directly and are in most cases only visible during runtime (e.g., objects that are coupled by method calls).

For the vertical integration we can rely on a defined hierarchy structure that forms the fundamental basis for the integration. We distinguish different strategies:

- the top-down integration,
- the bottom-up integration, and
- the outside-in integration.

The difference between top-down and bottom-up integration is clear. Whereas the top-down approach starts with main service providers and integrates in direction to the service consumers, the bottom-up approach works the other way round. One starts with the service consumers and proceeds with the service providers. The advantages of the bottom-up approach are the early integration with hardware and low-level driver and the direct accessibility of the interfaces to be tested[16] whereas the advantages of the top-down approach are the early test of top-level services and control structures and the ability to simulate error situations by means of dummies that simulate erroneous component behavior.

The outside-in integration is a combination of the top-down and the bottom-up approach and intends to subsume the best of both.[16]

The integration order for a vertical integration is determined by the sequence of messages or procedure calls that form the interdependencies between the components. That is, the integration starts with an arbitrary component and integrates the components that are involved in the communication step by step.[16]

Protocol testing. The term "protocol testing" addresses the validation of communication protocols and their respective realization in hardware and software. Communication protocols are more or less formal descriptions of the interactions that occur between a defined set of components in general and between a set of software components in particular. One of the main issues of protocol testing is conformance testing. Conformance testing is required to confirm if the concrete realization of a protocol conforms to a given standard. Standardized procedures for protocol testing and protocol testing processed have been developed by ISO and European Telecommunication Standards Institute (ETSI).

With the emergence of large, software-based telecommunication systems in the 1980s, the detailed and efficient test of communication protocols became more and more fundamental. Furthermore, the application of Formal Description Techniques (FDTs) for the specification of protocols in the early 1990s has led to a noticeable paradigm shift in the field of protocol testing as well. Nevertheless, most of the relevant research activities lie in the past. Actually, protocol testing is not a subject of comprehensive research anymore. A good summary of the state of the art of protocol testing is given in Refs. [17, 18].

In Ref. [18] different FDTs and the related test generation methods are explained. The author focuses on different generation methods. He distinguishes methods that address test generation from Mealy machine models (MMMs) (the T-Method,[19] the U-Method,[20] the D-Method,[21] and the W-Method[22]) and from unique input/output sequences (UIO sequences) (the UIOSs).[23–25] Moreover, the entry summarizes methods for test coverage and the application of failure models.

In Ref. [17], different kinds of protocol tests are distinguished:

- The tests during the development phase of a protocol or of a component that realizes a certain protocol aim to find errors in the implementation. They are carried out by developers and are similar to normal software (unit) tests.
- Conformance tests check whether an implementation conforms to a given protocol specification. In the case of standards, the specification is the standard itself.
- Interoperability tests address the interoperability between component different implementation of the same protocol. Often, the conformance to a specification is not sufficient to guarantee the interoperability of interacting components that emanate different distributors and implementers. Interoperability testing fills this gap.

Besides the given kinds of protocol test, the issues of performance and robustness have to be addressed by additional tests. Moreover, the generation of test cases is addressed. The author distinguishes between test generations on the basis of finite-state machines (FSM) (the W-method[22] and the UIO-method[23–25]) and Labeled Transition Systems (LTS) (Ioco Theory[26]).

Product and test quality

Measuring quality of a product is a difficult task. Quality is rather defined as the bundle of attributes present in a commodity and, where appropriate, the level of the attribute for which the consumer (software users) holds a positive value. Defining the attributes of software quality and determining the metrics to assess the relative value of each attribute are not formalized processes. Compounding the problem is that numerous metrics exist to test each quality attribute. Because users place different values on each attribute depending on the product's use, it is important that quality attributes be observable to consumers. In the following, we provide information regarding the product quality of a software product in the "Product quality" section. The "Test quality" section introduces measures for test quality, and the "Standards on testing quality" section lists explicit standards on testing quality.

Product quality. In 1991, the International Organization for Standardization (ISO) adopted ISO 9126 as the standard for software quality (ISO, 1991). It is structured around six main attributes listed below (subcharacteristics are listed in parenthesis):

- Functionality (suitability, accurateness, interoperability, compliance, security)
- Reliability (maturity, fault tolerance, recoverability)
- Usability (understandability, learnability, operability)
- Efficiency (time behavior, resource behavior)
- Maintainability (analyzability, changeability, stability, testability)
- Portability (adaptability, installability, conformance, replaceability)

Although a general set of standards has been agreed upon, the appropriate metrics to test how well software meets those standards are still poorly defined. Publications by IEEE (1988, 1996) have presented numerous potential metrics that can be used to test each attribute. These metrics include

- fault density,
- requirements compliance,
- test coverage, and
- mean time to failure.

The problem is that no metric is able to unambiguously measure a particular quality attribute. Different metrics may give different rank orderings of the same attribute, making comparisons across products difficult and uncertain.

The lack of quality metrics leads most companies to simply count the number of defects that emerge when testing occurs. Few organizations engage in other advanced testing techniques, such as forecasting field reliability based on test data and calculating defect density to benchmark the quality of their product against others. Regardless of the metric's quality, certain software attributes are more amenable to being measured than other attributes.

Pressman[27] describes the attributes that can be measured reliably and consistently across various types of software programs:

- effort, time, and capital spent in each stage of the project;
- number of functionalities implemented;
- number and type of errors remediate;
- number and type of errors not remediate;
- meeting scheduled deliverables; and
- specific benchmarks.

Interoperability, reliability, and maintainability are difficult to measure, but they are important when assessing the overall quality of the software product.

Determining which metric to choose from the family of available metrics is a difficult process. No unique measure exists that a developer can use or a user can apply to perfectly capture the concept of quality. Determining which metric to use is further complicated because different users have different preferences for software attributes.

Meeting quality requirements at each stage is supposed to ensure quality of the end product. To achieve quality, the system attributes must be clearly defined. The schedule for the project has to be taken into consideration as well. The usability feature from the quality triangle depicted in Fig. 2 suggests that users must be considered to ensure quality. It is not unusual for some authors to relate software quality to reliability and make reliability a component of conformance to features.

Test quality. Besides the quality of the software product itself the quality of the test specifications and implementations become more and more important in recent years. Test quality addresses different quality aspects of test specification and test implementations. Among others the following questions arise:

- How effective are the given test cases, test suites, etc. in terms of fault-revealing capabilities, specification coverage or code coverage, etc.?
- How modular and reusable are the test suites?
- How mature are the test suites by means of reliability, understandability, etc.?

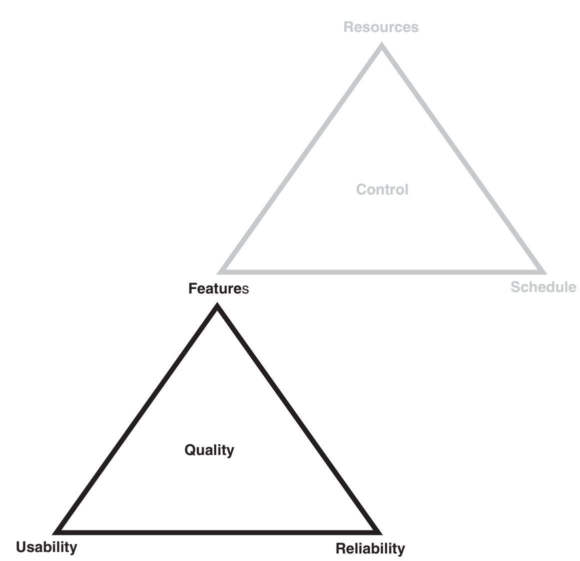

Fig. 2 The software quality triangle.

Approaches concerning the quality of test specifications and test implementations are constantly subject of discussions and various test quality methods and test quality metrics have been developed. Most of them are dedicated to individual quality aspects like code coverage to measure the effectiveness of test cases,[28–30] mutation analysis to provide insights to code stability and effectiveness of the test cases, and test ordering to show the interconnection of tests.

A more general view on the different quality aspects of test specifications is given in Ref. [31]. Fig. 3 shows a quality model for test specifications.

The model distinguishes between different quality characteristics and is derived from the ISO/IEC 9126 quality model. In the following we describe the different characteristics:

- **Test effectivity**: the test effectivity characteristic describes the capability of the specified tests to fulfill a given test purpose.

    — In the context of test specification, the *suitability* aspect is characterized by test coverage. Coverage constitutes a measure for test completeness and can be measured on different levels, e.g., the degree to which the test specification covers system requirements, system specification, or test purpose descriptions.

    — The *test correctness* characteristic denotes the correctness of the test specification with respect to the system specification or the test purposes. Furthermore, a test specification is only correct when it always returns correct test verdicts and when it has reachable end states.

    — The *fault-revealing capability* has been added to the list of subcharacteristics. Obtaining a good coverage with a test suite does not make any statement about the capability of a test specification to actually reveal faults. Usage of cause–effect analysis[32] for test creation or usage of mutation testing may be indicators for increased attention to the fault-revealing capability.

- **Reliability**: the reliability characteristic describes the capability of a test specification to maintain a specific level of performance under different conditions. In this context, the word “performance” expresses the degree to which needs are satisfied. The reliability subcharacteristics maturity, fault tolerance, and recoverability of ISO/IEC 9126 apply to test specifications as well.

    — Test results should always be reproducible in subsequent test runs if generally possible. Otherwise, debugging the SUT to locate a defect becomes hard to impossible. Test repeatability includes the demand for deterministic test specifications.

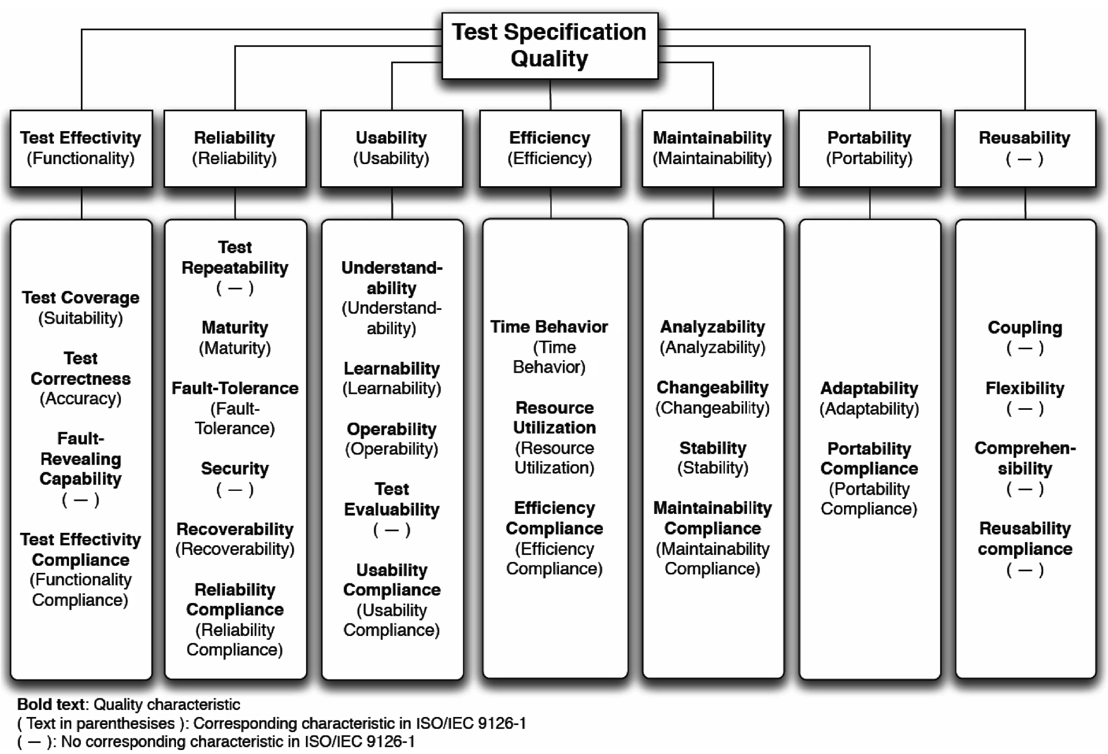

Fig. 3 The test specification quality model.

- The security subcharacteristic covers issues such as included plain-text passwords that play a role when test specifications are made publicly available or are exchanged between development teams.

- **Usability**: the usability attributes characterize the ease to actually instantiate or execute a test specification.

  - *Understandability* is important since the test user must be able to understand whether a test specification is suitable for his needs. Documentation and description of the overall purpose of the test specification are key factors—also to find suitable test selections.
  - The *learnability* of a test specification pursues a similar target. To properly use a test suite, the user must understand how it is configured, what kinds of parameters are involved, and how they affect test behavior. Proper documentation or style guides have positive influence on this quality as well.
  - A test specification has a poor *operability* if it, e.g., lacks appropriate default values, or a lot of external, i.e., non-automatable, actions are required in the actual test execution. Such factors make it hard to set up a test suite for execution or they make execution time-consuming due to a limited automation degree.
  - A new test-specific subcharacteristic in usability is *test evaluability*. The test specification must make sure that the provided test results are detailed enough for a thorough analysis. An important factor is the degree of detail of richness in test log messages.

- **Efficiency**: the efficiency characteristic relates to the capability of a test specification to provide acceptable performance in terms of speed and resource usage.
- **Maintainability**: maintainability of test specifications is important when test developers are faced with changing or expanding a test specification. It characterizes the capability of a test specification to be modified for error correction, improvement, or adaption to changes in the environment or requirements.

  - The analyzability aspect is concerned with the degree to which a test specification can be diagnosed for deficiencies. Test specifications should be well structured to allow code reviews. Test architecture, style guides, documentation, and generally well-structured code are elements that have influence in the quality of this property.
  - The changeability subcharacteristic describes the capability of the test specification to enable necessary modifications to be implemented, e.g., badly structured code or a test architecture that is not expandable may have negative impact on this quality aspect. Depending on the test specification language used, unexpected side effects due to a modification have negative impact on the stability aspect.

- **Portability**: portability in the context of test specification does only play a very limited role since test specifications are not yet instantiated. Therefore, installability (ease of installation in a specified environment), coexistence (with other test products in a common environment), and replaceability (capability of the product to be replaced by another one for the same purpose) are too concrete. However, adaptability is relevant since test specifications should be capable to be adapted to different SUTs or environments. For example, hardcoded SUT addresses (e.g., IP addresses or port numbers) or access data (e.g., user names) in the specification make it hard to adapt the specification for other SUTs.
- **Reusability**: although reusability is not part of ISO/IEC 9126, we consider this aspect to be particularly important for test specifications since it matters when test suites for different test types are specified. For example, the test behavior of a performance or stress test specification may differ from a functional test, but the test data, such as predefined messages, can be reused between those test suites.

Actually, coverage analysis on code and specification (requirements coverage) and static analysis of the test case code are the main measures to ensure test case quality.

In static code analysis, the so-called test smells are used to assess the code quality of test suites. The term "test smell" is derived from the term "code smell" and in general specifies flaws in the design or code of a test. Test smells describe test artifacts that are too long, complex, include unnecessary redundant code, exposing or breaking encapsulation of the application code, run slow, or make inappropriate assumptions on external resources.

Mutation analysis[33,34] allows measuring the effectiveness of test suites and helps optimizing automatically derived test suites. Commercial tools are rarely available.

Standards on testing quality. This section provides lists of relevant standards on software testing, software quality, etc.

Software testing:

- ISTQB "Standard Glossary of Terms Used in Software Testing" 2007 presents concepts, terms, and definitions designed to aid communication in (software) testing and related disciplines.[35]
- IEEE 829–2008 "IEEE Standard for Software Test Documentation" specifies the form and content of individual test documents.[36]
- BS 7925–2:1998 "Software Testing. Software Component Testing, Part 2" defines the process for

software component testing using specified test case design and measurement techniques.[37]
- IEEE 1028–2008 "IEEE Standard for Software Reviews and Audits" defines five types of software reviews and audits, together with procedures required for the execution of each review and audit type.[38]
- ISO 9646-1:1994 "Information Technology—Open Systems Interconnection—Conformance Testing Methodology and Framework—Part 1: General Concepts" specifies a general methodology for testing the conformance of products to OSI specifications, which the products claimed to implement.[39]
- ETSI ES 201 873 series defines the Testing and Test Control Notation version 3 (TTCN-3).[40]

Software quality and software quality management:

- IEEE 1061–1998 "Software-Quality Metrics Methodology" defines that for high-quality software the software's attributes must be clearly defined.[41]
- ISO 9001:2008 "Quality Management Systems—Requirements" specifies requirements for quality management systems for organizations.[42]
- ISO/IEC NP 90003 "Software Engineering. Guidelines for the Application of ISO 9001:2000 to Computer Software" provides guidelines for organizations in the application of ISO 9001:2000 to computer software.[43]
- ISO/IEC 9126-1:2001 "Software Engineering—Product Quality—Part 1: Quality Model" defines a product quality model.[44]
- IEEE 730–2002 "IEEE Standard for Software Quality Assurance Plans" provides uniform, minimum acceptable requirements for preparation and content of Software Quality Assurance Plans (SQAPs). This standard applies to the development and maintenance of critical software.[45]

Software evaluation:

- ISO/IEC 14598-1:1999 "Information Technology—Software Product Evaluation—Part 1: General Overview" gives an overview on how to evaluate software products.[46]
- ISO/IEC 25051:2006 "Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Requirements for Quality of Commercial Off-The-Shelf (COTS) Software Product and Instructions for Testing" defines quality requirements for COTS software products.[47]
- ISO/IEC 14102:2008 "Information Technology—Guideline for the Evaluation and Selection of CASE Tools" defines both a set of processes and a structured set of CASE tool characteristics for use in the technical evaluation and the ultimate selection of a CASE tool.[48]
- IEEE 982.1–2005 "IEEE Standard Dictionary of Measures of the Software Aspects of Dependability" is a standard dictionary of measures of the software aspects of dependability for assessing and predicting the reliability, maintainability, and availability of any software system.[49]

Critical software evaluation:

- IEEE 16085-2006 "Systems and Software Engineering—Life Cycle Processes—Risk Management" deals with risk management during the software life cycle process.[50]
- The Canadian Standards Association approved CSA-396.1.1, a "Quality Assurance Program for Previously Developed Software Used in Critical Applications."[51]
- RTCDO Std 178b "Software Considerations in Airborne Systems and Equipment Certification" deals with the development of software for aviation.[52]
- EN 50128:2001 "Railway Applications—Communication, Signaling and Processing Systems—Software for Railway Control and Protection Systems" specifies procedures and technical requirements for the development of programmable electronic systems for use in railway control and protection applications.[53]

Areas for Test Process Automation

Today, the industrial process for test case development follows an approach of stepwise collection and completion of test-related information. A practical process that has been established especially in the context of the development of standardized test suites may use the following sequence of documents:

- SUT specification
- Requirement catalog
- Test model
- Test purpose definition
- Test case description
- Test report

The document structure, notation, and degree of formalism are varying in the different application domains and depend on the test derivation methods.

An outcome of the test campaign execution are test reports related to the test cases that include generated test trace logs and test result verdicts. They can be used for the purpose of system failure reparation and/or coverage detection.

Both the creation of the documents listed above and the execution and evaluation of the test runs may be according to manual procedures but are also subject for semiautomatic or tool-supported activities. Researchers and tool developers are aiming to increase the degree of automation of the test steps.

The following section provides further details on the nature and characteristics of the major test process phases.

Test purposes and test model creation

The initial test information that provides the target and scope of the tests to be developed is traditionally a structured list of test purposes. It can be retrieved from either the system specification or requirement list. Alternatively, a test model containing test-relevant definitions about configuration, data, and behavior needed may be involved. In the latter case, test purposes may not be retrieved directly from the SUT information but the test model description.

Test case, data, and script generation

Test case descriptions need to contain test configuration, data, and behavior (sequence) definitions. This step is archived due to the refinement of completion of test purposes information. The outcome must be deterministic for an unambiguous interpretation and execution of the final test cases. The notation used for test description needs a clear operational semantic.

Test configurations are usually being defined as one of the first steps within the test development process. They have to reflect the possible access points (ports) of the SUT and the data types used at the involved interfaces. Test sequences will be derived by using the identified test activities reflecting the test purposes or test model behavior events. They need to be completed by appropriate pre- and postamble steps. The challenge of test data selection and combination is mostly a critical step in the automated test definition process since the explosion of test case number may be due to unlimited test data sets. Automated tools help to find appropriate data combination in order to cover data space.

Test case selection

Due to a high number of test cases or a long execution time of parts of the test suite, a selection or ordering according to empirical or calculated priorities of the available test cases is required. Selection criteria are due to, e.g., coverage or economical aspects and both can benefit from automated tool support.

Test execution

An automated procedure for test case execution helps to avoid manual failure and to ensure exact repeatability of the tests. Additionally, automated test execution is essential if time-critical event sequences (e.g., short reactions) or high parallelisms (e.g., loads) are addressed.

Test management and test result analysis

Test verdict management and analysis of identified misbehavior of the SUT is very time-consuming and require very good logging features of the test execution tool, e.g., data view facilities to compare observations with expectation and graphical traces are of great advantages. This includes tool support for the backward association of observation with the test description and/or system definition.

The test report generation is the last step of the test process and may be subject for test automation in order to give a fast and clear overview on extensive test campaigns.

Using Models for Testing

MBT refers to software testing where test cases are derived in whole or in part from a model that describes selected, often structural, functional, sometimes non-functional aspects of a SUT. According to Ref. [54], in recent years, people are using this term for a wide variety of test generation techniques, while the following four main approaches are known as MBT:

- Generation of test input data from domain model
- Generation of test cases from environmental model
- Generation of test cases with oracles from a behavior model
- Generation of test scripts from abstract tests

With this view of MBT, it may be defined as "the automation of the design of black-box tests." It is different from the usual black-box testing that instead of writing tests based on requirements documentation, the model of SUT should be created and used for automatic generation of tests.

Goals for using models for testing

The basic idea of deriving test models from system models is to reuse the information about the system to be developed also for developing the test model as the counterpart to the system. In particular, the following system information can be used for the derivation of test models for black-box testing:

- The structure and configuration of the system to be developed in terms of components, interfaces, connected instances, etc.
- The system behavior externally observable at component ports.
- The type system (in particular user-defined types, e.g., structured types).
- Some concrete data values, e.g., used for selecting between branches in the control flow.

Limitations

System and test models differ in their degree of abstraction and completeness. It is obvious that the stage of information provided by the models is essential for scope and benefits from the model usage. This is important and even more important if the work is dedicated beyond academic case studies but to real industrial application.

In many domains, the system descriptions are formalized only partly or apply an in-house technology platform that is not useful or open for data exchange or interaction with other tools. The technology, notations, and tools for modeling need to be adequate for the system definition and test development process.

Furthermore, the quality of the model has to be taken into account for further usage, in various dimensions, e.g., readability for humans and extent of tool support. This includes the application of guideline and modeling styles.

Role of models

Due to the experiences and limitations with MBT in practice today the models may get different roles that depend on the degree of completeness and target usage. Some sample roles are given in the following list:

- Overview and clarity on system structure and design for handling of complexity and general test planning
- Analysis and understanding of the test target and purposes
- Planning of the test system infrastructure due to the system architecture and configurations
- Planning of the test efforts and estimation of test campaign durations
- Automation of parts of the test development process
- Generation of generic test templates and combinations to be extended and improved
- Abstract test definitions including data and behavior sequences
- Executable test suites to be completed with parameters only

Due to this list it is obvious that models have value in all test development and test campaign execution phases even if the information is focused on system parts only. In the following, we review selected formalisms that are currently used for MBT approaches.

UML. Nowadays, UML is often used for the specification of test suites. Ref. [55] relates TTCN-3[56,57] and the UML 2.0 testing profile. Ref. [58] proposes a Use Interaction Test (UIT) method that allows test generation from UML diagrams (use case and sequence diagrams). The authors claim that the method reuses UML diagrams developed for analysis and design without requiring any additional formalism or ad hoc effort to specify test purposes. However, "choices" that represent a list of specific situations or ranges of input data should be determined for test generation by the UIT method. These choices together with "constraints among choices" can also be considered as implicit test purposes.

MSC. In Refs. [59, 60], Message Sequence Charts (MSCs) are used to specify test purposes. Test purposes can be developed by a test designer or derived fully automatically from an SDL specification. A test developer can also choose to test certain aspects of the system, i.e., certain transitions, processes, or blocks of the specification can be marked as covered so that they are ignored during the test purpose computation. Simple basic MSCs are easy to understand and process automatically. However, understanding MSCs[61] containing new advanced control structures and treatment of data is non-trivial.[62]

Temporal logic. In Ref. [63], a test purpose is expressed as a property in temporal logic. The general idea is to allow automatic test generation from a partial specification. The proposed test generation technique involves a (partial) specification S and a safety or bounded liveness property P. Specification S should be "close enough" to the actual behavior of the Implementation Under Test (IUT). No particular conformance relation between S and the IUT is required at this level. Property P is given through an observer O that can recognize sequences of P. This observer is a parameterized automaton on infinite words. Test cases are automatically generated by traversing the specification in order to find "interesting" execution sequences that are able to show non-satisfiability of P by the IUT.

Test case generation approaches

In this section, various test case generation approaches applied for MBT are presented.

Deductive theorem proving. Theorem proving has been originally used for automated proving of logical formulas. For MBT approaches the system is modeled by a set of logical expressions specifying the system's behavior. For selecting test cases, the model is partitioned into equivalence classes over the valid interpretation of the set of logical expressions describing the SUT. Each class represents certain system behavior and can therefore serve as a test case. The simplest partitioning is done by the disjunctive normal form approach. The logical expressions describing the system's behavior are transformed into the disjunctive normal form. The classification tree method provides a more sophisticated hierarchical partitioning. Also, partitioning heuristics are used supporting the partitioning algorithms, e.g., heuristics based on boundary value analysis.

Constraint logic programming. Constraint programming can be used to select test cases satisfying specific constraints by solving a set of constraints over a set of variables. The system is described by means of constraints. Solving the set of constraints can be done by Boolean solvers or by numerical analysis, like the Gaussian elimination. A solution found by solving the set of constraints formulas can serve as test cases for the corresponding system.

Model checking. Originally, model checking was developed as a technique to check if a property of a specification is valid in a model. Herein, a model of the SUT is provided to the model checker. Within the procedure of proofing if this property is valid in the model, the model checker detects witnesses and counterexamples. A witness is a path where the property is satisfied; a counterexample is a path in the execution of the model where the property is violated. These paths can be used as test cases.

Symbolic execution. Symbolic execution is often used in frameworks for MBT. It can be a means in searching for execution traces in an abstract model. In principle, the program execution is simulated using symbols for variables rather than actual values. Then the program can be executed in a symbolic way. Each execution path represents one possible program execution and can be used as a test case. For that the symbols have to be instantiated by assigning values to the symbols.

MODEL-BASED TESTING APPROACHES

A principal positioning of MBT in the test methods taxonomy is shown in Fig. 4. MBT can be used both for static and dynamic tests. It can be applied both for manual and tool-supported, automated testing. For manual testing, MBT provides guidance in performing the tests only, whereas for automated testing, higher efficiency, coverage, etc. can be obtained, so that a substantial gain can be achieved when combining MBT and test automation.

In static testing, essentially the information from the system model is examined, such as the system architecture, system interfaces, system components and their relations, etc. The system model (or parts of it) is interpreted as a set of rules to which the system must correspond; see, for example, Ref. [64].

More often, however, MBT is used for dynamic testing.[65] Dynamic tests can use active (i.e., intrusive) or passive (i.e., non-intrusive) tests. Active tests provide stimuli to the SUT and observe and analyze the reactions. Passive tests analyze traces of the system execution and compare them against the system model. For active testing, test cases from the data, and structural and behavioral information of the system models are derived, completed (if needed), and applied to the SUT. For passive testing, system invariants and/or conditions, which are given in the system model, are analyzed along the traces (by a forward or backward search).

Fig. 5 represents the relations between system and test system and between their models: the requirements represent—from different perspectives—both the intended system and test system and their models, of which typically several on different abstraction levels exist. On the other hand, system and test system (and their models) realize the requirements. System and test system are dual to each other: while the test system is developed to validate the requirements in the system, the system serves also for the validation of the test system. The same is true on model level—and provides an additional validation possibility: the test model can be used for an early validation of the system model; there are different variants of MBT processes that make different use of system and/or test models (see entry Model-Based Testing-Approaches and Notations, p. xxx).

SUMMARY

MBT constitutes a number of technologies, procedures, and approaches, with the aim to improve the quality and

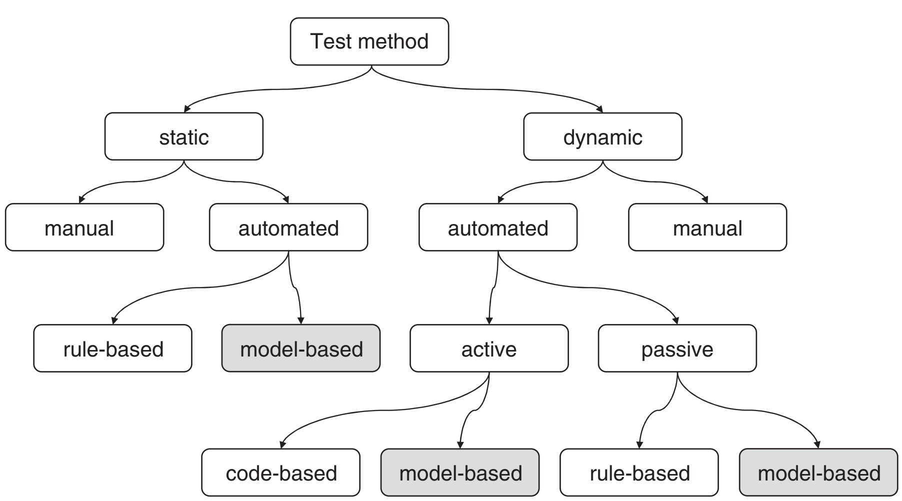

Fig. 4 Test methods taxonomy and model-based testing.

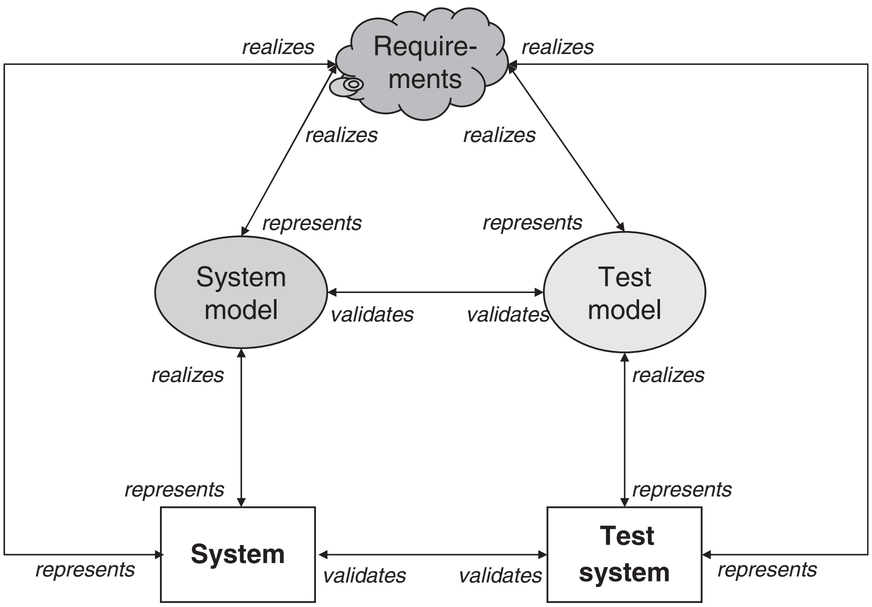

Fig. 5 Duality of system, test system, and their models.

effectiveness of test specification and test execution. Started as a pure academic field of application it has gained significance for industrial domains in recent years. With the, even if slow, adoption of Model-Based Engineering techniques for industrial software engineering processes, the basis to apply MBT approaches is nowadays much better than a few years ago. The availability of yet mature tools has also led to a higher interest on MBT. For a discussion of various approaches to MBT, languages and notations, and applications, please see Model-Based Testing B and Model-Based Testing C.

ABBREVIATIONS

ASM	Annotated SUT Model
ATS	Abstract Test Suite
DTM	Dedicated Test Model
EMF	Eclipse Modeling Framework
ETS	Executable Test Suite
ETSI	European Telecommunication Standards Institute
GUI	Graphical User Interface
IDE	Integrated Development Environment
IDL	Interface Definition Language
IUT	Implementation Under Test
MDA	Model -Driven Architecture
MM	Meta-Model
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform-Independent Model
PIT	Platform-Specific Test Model
PSM	Platform-Specific Model
PST	Platform-Specific Test Model
SUT	System Under Test
TTCN-3	Testing and Test Control Notation
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

REFERENCES

1. The Mathworks. Website of the Matlab/Simulink Modeling Tool, http://www.mathworks.de/products/simulink/ (accessed August 2009).
2. OMG. The Unified Modeling Language, UML 2.1.2 Infrastructure Specification, http://www.omg.org/docs/formal/07-11-04.pdf (accessed August 2009).
3. OMG. The Unified Modeling Language, UML 2.1.1 Superstructure Specification, http://www.omg.org/spec/UML/2.1.1/Superstructure/PDF (accessed August 2009).
4. National Instruments. Introducing LabView, Website of the LabView Infrastructure, http://www.ni.com/labview/ (accessed August 2009).
5. OMG. Business Modeling & Integration DTF, http://bmi.omg.org/ (accessed August 2009).
6. Baker, P.; Loh, S.; Weil, F. Model-Driven Engineering in a Large Industrial Context—Motorola Case Study, in Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, Oct 2–7, 2005; 476–491.
7. Baker, P.; Bristow, P.; Jervis, C.; King, D.; Mitchell, B. Automatic generation of conformance test from message sequence charts. In 3rd SAM Workshop—Telecommunication and Beyond The broader applicability of SDL and MSC, June 2002.
8. Utting, M.; Legeard, B. *Practical Model-based Testing: A Tools Approach*; Morgan-Kaufmann: San Francisco, CA, 2007.
9. Neto, A.D.; Subramanyan, R.; Vieira, M.; Travassos, G.H.; Shull, F. Improving evidence about software technologies—a

look at model-based testing. IEEE Software **2008**, *25* (3), 10–13.
10. Baker, P.; Dai, Z.R.; Grabowski, J.; Haugen, Ø.; Schieferdecker, I.; Williams, C. Model-driven testing: Using the UML testing profile, 1st Ed.; Springer: Berlin, 2007.
11. The V-Modell® XT, http://v-modell.iabg.de/v-modell-xt-html-english/index.html (accessed August 2009).
12. The IBM Rational Unified Process Data Sheet, ftp://ftp.software.ibm.com/software/rational/web/datasheets/RUP_DS.pdf (accessed August 2009).
13. Spillner, A. W-model—test process parallel to the development process. In Conference: Jornada sobre Testeo de Software (JTS), ITI Instituto Tecnológico de Infomática, Universidad Politécnica de Valencia, Spain, Mar 25–26, 2004.
14. Beck, K.; et al. Manifesto for Agile Software Development, http://agilemanifesto.org/ (accessed August 2009).
15. Beck, K. *Test-Driven Development by Example*; Addison Wesley: Boston, MA, 2003.
16. Liggesmeyer, P. *Software-Qualität, Testen: Analysieren und Verifizieren von Software*; Springer-Verlag: Berlin, Heidelberg, 2002.
16. Lai, R. A survey of communication protocol testing. J. Syst. Software **2002**, *62* (1), 21–46.
17. König, H. Protocol Engineering Prinzip, Beschreibung und Entwicklung von Kommunikationsprotokollen, B.G. Teubner, Stuttgart Leipzig Wiesbaden, 2004.
19. Naito, S.; Tsunoyama, M. Fault detection for sequential machines by transition tours. In proceedings of 11th IEEE International Symposium on Fault-Tolerant Computing; IEEE Computer Society Press: Los Alamitos, CA, 1981; 238–243.
20. Sabnani, K.; Dahbura, A. A protocol test generation procedure. Computer Networks ISDN Syst. **1988**, *15*, 285–297.
21. Gonenc, G. A method for the design of fault detection experiments. IEEE Trans. Computer **1970**, *C-19*, 551–558.
22. Chow, T. Testing software designs modeled by finite-state machines. IEEE Trans. Software Eng. **1978**, *SE-r*, 178–187.
23. Aho, A.V.; Dahbura, A.T.; Lee, D.; Uyar, M.U. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese Postman Tuours. In *Protocol Specification, Testing and Verification*; Aggarwal, S., Sabnani, K. K., Eds.; Elsevier: Amsterdam 1988; Vol. 8, 75–86.
24. Shen, Y.N.; Lombardi, F.; Dahbura, A.T. Protocol conformance testing using multiple UIO sequences. In *Protocol Specification, Testing and Verification*; Brinksma, E., Scollo, G., Visers, C.A., Eds.; Elesvier: Amsterdam, IFI: New York, 1990; Vol. 9.
25. Woojik, C.; Paul, D.A. Improvements on UIO sequence generation and Partial UIO sequences. In *Protocol Specification, Testing and Verification*; Linn, R.J., Jr., Uyar, M.U., Eds.; Elsevier: Amsterdam; IFIP: New York, 1992; Vol. 12.
26. de Vries, R.G.; Tretmans, J. Towards formal test purposes. In Formal Approaches to Testing of Software 2001 (FATES), BRICS Notes Series (NS-01-4), BRICS, University of Aarhus, Denmark, 2001; 61–76.
27. Pressman, R. Software Engineering, a practitioner's approach, 3rd Ed.; McGraw-Hill: New York, NY; 1992. ISBN: 0-07-112779-8/0072496681.
28. Sneed, H.M. Measuring the Effectiveness of Software Testing. In Proceedings of SOQUA 2004 and TECOS 2004; Beydeda, S., Gruhn, V., Mayer, J., Reussner, R., Schweiggert, F., Eds.; Lecture Notes in Informatics (LNI). Gesellschaft für Informatik, 2004; Vol. 58.
29. Vega, D.-E.; Schieferdecker, I. Towards quality of TTCN-3 tests. In Proceedings of SAM'06: Fifth Workshop on System Analysis and Modelling, May 31–June 2, 2006, University of Kaiserslautern, Germany, 2006.
30. Zeiss, B.; Neukirchen, H.; Grabowski, J.; Evans, D.; Baker, P. Refactoring and metrics for TTCN-3 test suites. In *System Analysis and Modeling: Language Profiles*; Gotzhein, R., Reed, R., Eds.; Lecture Notes in Computer Science. Springer: Heidelberg, 2006; Vol. 4320, 148–165.
31. Zeiß, B.; Vega, D.; Schieferdecker, I.; Neukirchen, H.; Grabowski, J. Applying the ISO 9126 Quality model to test specifications exemplified for TTCN-3 test specifications. In Software Engineering 2007 (SE). Lecture Notes in Informatics (LNI). Gesellschaft für Informatik, Köllen Verlag: Bonn, 2007.
32. Myers, G.J. *The Art of Software Testing*; John Wiley & Sons: New York, 1979. ISBN 0-471-04328-1.
33. Offutt, J.; Untch, R. Mutation 2000: Uniting the Orthogonal, Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, October 2000; 45–55.
34. Offutt, J. Practical Mutation testing. In Twelfth International Conference on Testing Computer Software, Washington, D.C., June 1995; 99–109.
35. ISTQB, Standard Glossary of Terms used in Software Testing, 2007, http://istqb.org/downloads/glossary-current.pdf (accessed August 2009).
36. IEEE 829-2008. IEEE Standard for Software Test Documentation, http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4459218 (accessed August 2009).
37. BS 7925-2:1998. Software testing. Software component testing, Part 2, http://shop.bsigroup.com/en/ProductDetail/?pid=000000000001448026 (accessed August 2009).
38. IEEE 1028-2008. IEEE Standard for Software Reviews and Audits, http://ieeexplore.ieee.org/xpls/abs_all.jsp?isnumber=4601583&arnumber=4601584 (accessed August 2009).
39. ISO 9646-1:1994. Information technology—Open Systems Interconnection—Conformance testing methodology and framework—Part 1: General concepts, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=17473 (accessed August 2009).
40. ETSI ES 201 873. Testing and Test Control Notation, http://www.ttcn-3.org/StandardSuite.htm (accessed August 2009).
41. IEEE 1061–1998. Software-Quality Metrics Methodology, http://ieeexplore.ieee.org/xpls/abs_all.jsp?tp=&isnumber=16194&arnumber=749159&punumber=6061 (accessed August 2009).
42. ISO 9001:2008. Quality Management Systems—Requirements, http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=46486 (accessed August 2009).
43. ISO/IEC NP 90003. Software Engineering. Guidelines for the application of ISO 9001:2000 to computer software, http://www.iso.org/iso/iso_catalogue/catalogue_

ics/catalogue_detail_ics.htm?csnumber=52373 (accessed August 2009).

44. ISO/IEC 9126-1:2001. Software engineering—Product quality—Part 1: Quality model, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749 (accessed August 2009).
45. IEEE 730–2002. IEEE Standard for Software Quality Assurance Plans, http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1040117 (accessed August 2009).
46. ISO/IEC 14598-1:1999. Information Technology—Software Product Evaluation—Part 1: General Overview, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=24902 (accessed August 2009).
47. ISO/IEC 25051:2006. Software engineering—Software product Quality Requirements and Evaluation (SQuaRE)—Requirements for quality of Commercial Off-The-Shelf (COTS) software product and instructions for testing, http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ ics.htm?csnumber=37457 (accessed August 2009).
48. ISO/IEC 14102:2008. Information technology—Guideline for the evaluation and selection of CASE tools, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43189 (accessed August 2009).
49. IEEE 982.1-2005. IEEE Standard Dictionary of Measures of the Software Aspects of Dependability, http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1634994 (accessed August 2009).
50. IEEE 16085-2006. Systems and software engineering—Life cycle processes—Risk management, http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=04042193 (accessed August, 2009).
51. CSA-396.1.1. Quality Assurance Program for Previously Developed Software Used in Critical Applications, http://www.techstreet.com/cgi-bin/detail?product_id=944623 (accessed August 2009).
52. RTCDO Std 178b. Software Considerations in Airborne Systems and Equipment Certification, http://www.rtca.org/onlinecart/product.cfm?id=341 (accessed August 2009).
53. EN 50128:2001. Railway applications—Communication, signalling and processing systems—Software for railway control and protection systems, http://tcelis.cenelec.be/pls/portal30/CELISPROC.RPT_WEB_PROJECT_D.SHOW?p_arg_names=project_number&p_arg_values=4795# (accessed August 2009).
54. Vincenzi, A.M.R.; Maldonado, J.C.; Delamaro, M.E.; Spoto, E.S.; Wong, W.E. Component-based software: An overview of testing. Component-Based Software Quality **2003**, *2693*, 99–127.
55. Schieferdecker, I.; Dai, Z.R.; Grabowski, J.; Rennoch, A. The UML 2.0 Testing Profile and its Relation to TTCN-3. In *Testing of Communicating Systems*; Hogrefe, D., Wiles, A., Eds.; Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems (TestCom), Sophia Antipolis, France, May 2003. Lecture Notes in Computer Science 2644; Springer-Verlag: Berlin, Heidelberg; 79–94.
56. ETSI ES 201 873-1 (v4.1.1). Methods for testing and specification (MTS); the testing and test control notation version 3; part 1: TTCN-3 core language, 2009.
57. ETSI ES 201 873-3 (v3.2.1). Methods for testing and specification (MTS); the testing and test control notation version 3; part 3: TTCN-3 graphical presentation format (GFT), 2007.
58. Bassanieri, F.; Bertolino, A.; Marchetti, E. The cow suite approach to planning and deriving test suites in UML projects. In Fifth International Conference on the Unified Modeling Language—the Language and its applications UML 2002, Dresden, Germany, LNCS, 383–397, 2002; Vol. 460.
59. Baker, P.; Bristow, P.; Jervis, C.; King, D.; Mitchell, B. Automatic generation of conformance test from message sequence charts. In 3rd SAM Workshop—Telecommunication and Beyond The broader applicability of SDL and MSC, Aberystwyth, U.K., June 2002.
60. Schmitt, M.; Ebner, M.; Grabowski, J. Test generation with autolink and testcomposer. In Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM), Grenoble, France, 2000.
61. ITU-T Recommendation Z.120. Message Sequence Chart, http://www.itu.int/itudoc/itu-t/approved/z/index.html.2008 (accessed August 2009).
62. Jonsson, B.; Padilla, G. An execution semantics for msc-2000. In SDL 2001: Meeting UML: 10th International SDL Forum Copenhagen, Denmark, Proceedings, June 27–29, 2001; Reed, R., Reed, J., Lecture Notes in Computer Science. Springer: Berlin, 2003, Vol. 2078.
63. Fernandez, J.C.; Mounier, L.; Pachon, C. Property oriented test case generation. In Proceedings of FATES (Satellite workshop of ASE), Montreal, Canada, 2003.
64. A. Abdurazik, J. Offutt: Using UML Collaboration Diagrams for Static Checking and Test Generation. Conference, LNCS 1939; Springer: Berlin, 2000.
65. Briand, L.; Labiche, Y. A UML-based approach to system testing. Journal of Software and Systems Modeling **2002**, *1*, 10–42.