Model-Based Testing: Approaches and Notations

Ina Schieferdecker
Axel Rennoch
Alain Vouffo-Feudjio
Competence Center of 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. The entry investigates MBT approaches and the languages and notations used. It is the successor entry to model-based testing by the same authors.

MODEL-BASED TESTING APPROACHES

Model-based testing (MBT) can be used for both static and dynamic tests. A principal positioning of MBT in the test methods taxonomy is shown in Fig. 1. It can be applied for both manual and tool-supported automated testing. For manual testing, MBT provides guidance in performing the tests only, whereby 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 like the system architecture, system interfaces, system components, and their relations is examined. The system model (or parts of it) is interpreted as a set of rules to which the system must correspond, see for example Ref. [1].

More often MBT is used for dynamic testing.[2] Dynamic tests can use active (i.e., intrusive) or passive (i.e., non-intrusive) tests. Active tests provide stimuli to the system under test 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, structural and behavioral information of the system models are derived, completed (if needed), and applied to the system under test. 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. 2 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 exist, on different abstraction levels. 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 the model level—and provides an additional validation possibility: the test model can be used for an early validation of the system model.

As described in the subsequent subsections, there are different variants of MBT processes that make different use of system and/or test models.

System Model Only Approach

A very common approach of MBT is the exclusive use of a system model (Fig. 3), which is often defined in UML, SDL, MSC, or other formal notations like Petri nets, temporal logic. In this approach, not only the test system (or parts of it) is generated from the system model, but the system itself (or parts of it) is also generated. One testing method used in this setting is the so-called on-the-fly testing. As in the case of on-the-fly model checking, instead of generating the system state space fully before deriving tests (which yields scalability issues), the system state space is dynamically explored by using the system model as a test driver.[3] Other methods produce test code directly from the system model, e.g., for integration tests from collaboration diagrams[1] or for unit tests from automata.[4]

An advantage of the system model-driven approach is that only one model is used. Thus, modeling expenditures are reduced and inconsistencies between system and tests are limited. However, it also has a big disadvantage: since

Encyclopedia of Software Engineering DOI: 10.1081/E-ESE-120046902
Copyright © 2011 by Taylor & Francis. All rights reserved.

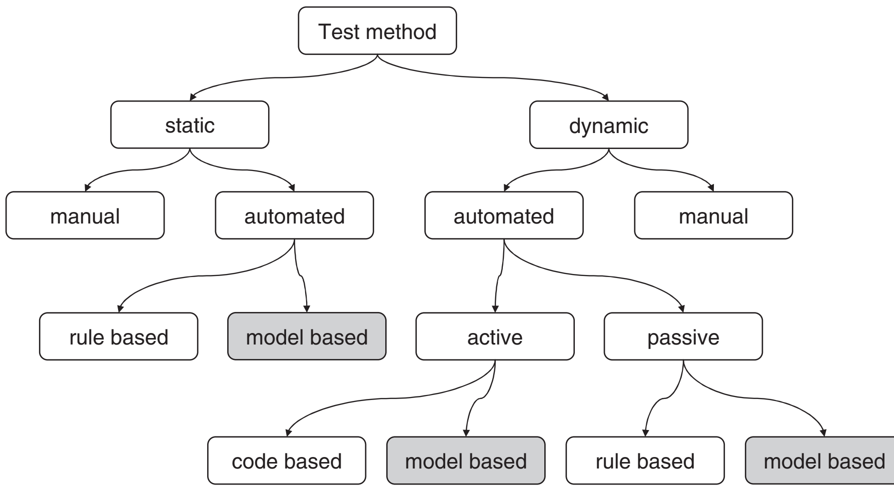

Fig. 1 Test methods taxonomy and model-based testing.

system and test system are produced from the same source, the independence of the system and the test is missing: the tests can only check whether the system model is correctly realized in the system itself but cannot identify errors that are already contained in the system model. The single source issue makes the test of limited power.

Test Model Only Approaches

In a test model only approach (Fig. 4), proprietary or standardized test modeling techniques as described in the "Model-Based Testing Languages and Notations" section are used. Based on the original requirements, the test models are manually developed. Test generation is used to generate the test system, i.e., to automate the test executions, but not to generate the test model itself (variant (b) and (c)). The test model may even be used to generate the system itself (variant (c)), which is however only a theoretical option (see, e.g., Ref. [5]), but is not in practical use.

Variant (b) is of widespread use (in particular in combination with TTCN-3, e.g., Ref. [6]). It offers the advantage of a dedicated model for testing, which is able to represent the tester's needs. It is however not easy to validate such a stand-alone test model with respect to correctness, test coverage. Instead, test model guidelines and rules are checked.

Variant (c) is in analogy to variant (a) a possible option, which however has the same critical single source problem as variant (a). However, if the two principles "test first" and "model based" are to be applied, (c) would be a variant, which though is superseded by variant (e) in the "System and Test Model Combined Approaches" section.

System and Test Model Combined Approaches

Variants (d), (e), and (f) (see the "Independent System and Test Model Approach" section) realize MBT most consistently: models are used both for the system and for the test system. After the development or (partial) generation of these models, they can be further refined.

Variant (d) uses the system model to generate the test model (or parts of it). Variant (e) uses the test model to generate the system model (or parts of it) (Fig. 5).

Most approaches use variant (d), whereby often automata-based methods are used.[7] Variant (e) was publicized by companies like Telelogic/IBM, but was not developed further.

The advantage of these variants is the existence of two models, by which both system and test systems can be modeled and analyzed abstractly in a technology-independent manner. The analysis can take place before

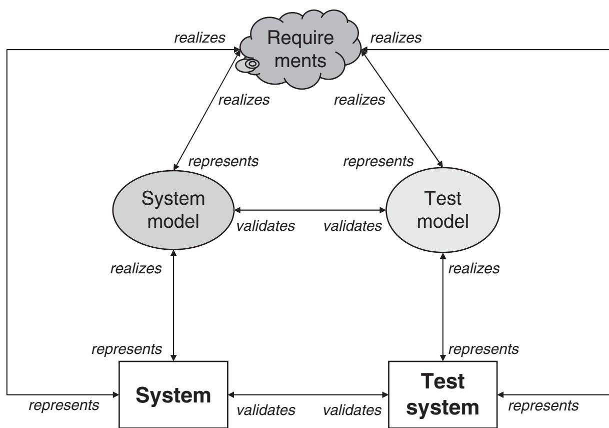

Fig. 2 Duality of system, test system, and their models.

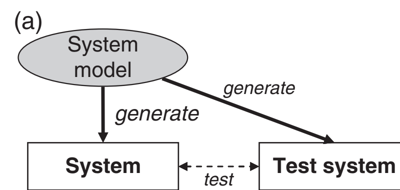

Fig. 3 System model only approach.

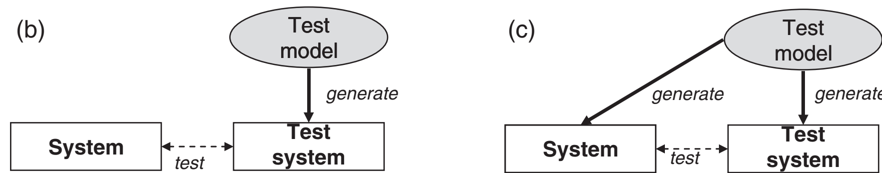

Fig. 4 Test model only approach.

any implementation of the system or the test system. The analysis can address model correctness in itself, the consistency of the model and can also derive coverage metrics that indicate, for example, the test quality.

Furthermore, the models (or parts of them) are used as a basis to generate the other models (or parts of them), which lowers the development efforts for the models but bears the potential problem of error propagation into the other model and the generated system or test system, respectively.

Independent System and Test Model Approach

According to Ref. [8], variant (f) (Fig. 6) is optimal with respect to the independence of the two models: error propagation is reduced as the models are developed independently by system modelers and test modelers. Only those parts of the system model that do not represent testing concerns are reused in the test model. For example, if the dynamic system behavior is to be tested and if the static interface structures have been checked already, the test model can make use of the interface definitions in the system model. Variant (f) has been realized in the context of MDA in Ref. [9].

The advantage of variant (f) comes with increased efforts for test model development. This is however typically paid off by an increased efficiency and quality of the tests and an improved maintenance of the system, test system, and their models.

MODEL-BASED TESTING LANGUAGES AND NOTATIONS

Despite significant, recent advances in software-intensive systems engineering and the availability of sophisticated commercial testing tools, test automation still has much potential for improvement. Automation in quality assurance is often synonymous with test generation from structural and behavioral models and tool-supported test execution. However, automated testing has gained much momentum with the development of the test technology TTCN-3. It is the testing and test control notation developed by ETSI and also adopted by ITU that addresses testing needs of modern communication, software, and embedded systems technologies. The standardized test language has a look and feel similar to a typical programming language. However, in addition to the typical programming constructs, it contains all the important features necessary to specify test procedures and campaigns for functional, conformance, interoperability, load, and scalability tests. So its testing-specific features make it distinct to classical scripting or programming languages, while being technology-independent. Its application in industrial-scale environments in different domains including mobile communication, railway control systems, financial applications, and telematics systems has been proven by the ITEA TT-Medal project only recently. TTCN-3 also got an entry into UML-based development processes via the definition of the UML testing profile (UTP). This enables test specifications with UML concepts, which can automatically be executed on TTCN-3 platforms.

Over the recent decades, state-of-the-art testing and tool support were represented by the "capture & play" paradigm. This approach however has a number of problems that can make test automation quite worthless. First of all, automated test scripts are usually self-contained and do not share any knowledge on the system under test: this causes large error-prone code duplication and time-consuming maintenance. Second, tests are so closely tied to trivial details of the user interface that code changes frequently cause test refactoring. Thus, regression testing is often hard to perform. In fact, automation in quality assurance should touch the complete software life cycle, where it starts with a tool-supported requirements definition, performing frequent consistency checks between related documents and subsequent phases, and results in a lean and highly standardized quality approval. An important step toward this visionary software development process was the introduction and broad acceptance of the MDA as

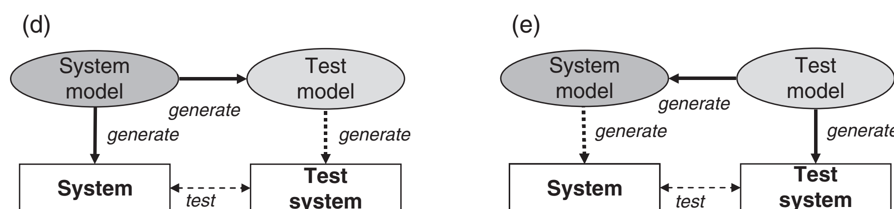

Fig. 5 Combined approaches.

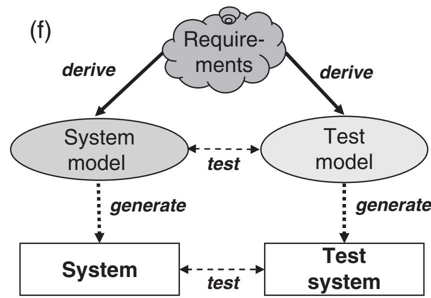

Fig. 6 Independent model approach.

an imperfect but promising guideline to model-driven software development.

The main benefit of MBT is the possibility of using the same model for both specification and testing of the application. This approach intends to use information about the system behavior included in the model for creation of test cases. In this section, we consider model-based test generation tools. We focus on test generators that have one of the formal or semiformal specification languages (like UML, SDL, LOTOS) as an input language. Particular attention is paid to test generators using test generation directives to guide the generation process.

UML Testing Profile

The unified modeling language (UML)[10,11] is a graphical language to support the design and development of complex object-oriented systems. While it is flexible in addressing the major object-oriented concepts, test specification and testing issues are beyond the scope of UML version 1.4. In late 2001, the Object Management Group (OMG) issued a Request for Proposals (RFP) to develop a testing profile for version 2.0 of UML (UML 2.0). A UML profile is a domain-specific extension of UML provided using a standardized extensibility mechanism.

The UTP defines testing concepts, including test context, test case, test component, and verdicts, that are commonly used during testing. Behavioral elements from UML 2.0 can be used to specify the dynamic nature of test cases. These include interactions, state diagrams, and activity diagrams. Additional concepts for behavior include several types of actions (validation actions, log actions, final actions, etc.), defaults for managing unexpected behavior, and arbiters for determining the final verdict for a test case. The definition and handling of test data is supported by wildcards, data pools, data partitions, data selectors, and coding rules. Timers and time zones are also provided to enable specifications of test cases with appropriate timing capabilities.

UTP also contains a stand-alone meta-model as a separate compliance point. This allows non-UML tools to provide an implementation that is consistent with the UML-based profile. The Eclipse TPTP[12] project currently has implemented this stand-alone model as a basis for their test information model.

The UTP provides a coherent set of extensions to UML 2.0 that support effective test specification and modeling for black-box testing. This is a significant enhancement to UML to support the testing portion of the system development life cycle. Meanwhile, the UTP development has come to its finalization and it has become an official standard of the OMG.

Unified Test Modeling Language

The Unified Test Modeling Language (UTML) is a notation developed by FOKUS to support model-driven test engineering. It adopts, refines, and extends concepts of the UTP to provide a modeling approach that supports test development from requirements to test cases. UML profiles automatically inherit all concepts of UML and are therefore just as powerful in terms of expressiveness. However, profiles also inherit the complexity of the UML notation, although such a high level of complexity might not be required in the targeted domain. The consequence is that the modeling process with profiles is sometimes described as unnatural and lacking intuitiveness. UTML is exclusively dedicated to the design of tests using a model-driven development approach. Therefore, it explicitly restricts the scope of modeling and provides an intuitive modeling process to fulfill that purpose. This makes the notation less difficult to learn and should facilitate its adoption by testing experts.

Fig. 7 depicts the test modeling process, which starts with the specification of test objectives (what to test) and leads to the specification of test behavior, i.e., a definition of the actions to perform and observations required for verifying the SUT's correct behavior.

UTML defines five types of test models that can be combined and linked to one another to provide a complete model of a whole test system:

- Test objective models describe what the tests need to verify.
- Test strategy models describe how each of the test objectives is going to be achieved.
- Test architecture models describe test configurations, i.e., the topological context in which test behavior will be executed. This includes the entities involved in the test scenario, the points of stimulation and/or observation and how they are connected to one another.
- Test data models describe the data to be used for sending impulses to the SUT or those for describing the requirements to be met by responses of the SUT to be considered valid.
- Test behavior models describe the series of actions to be performed between the entities building the test architecture to verify the SUT's behavior. This includes actions for:

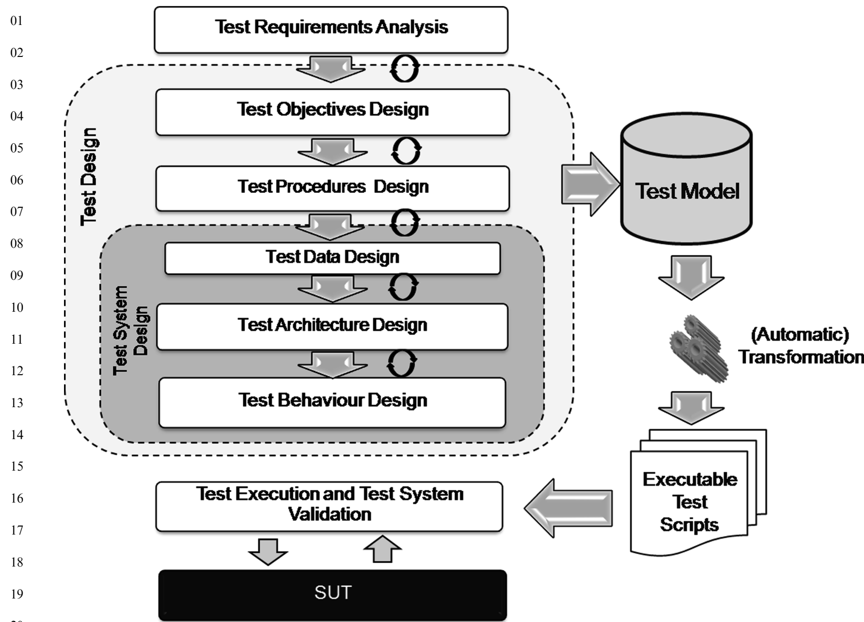

Fig. 7 The UTML approach.

- sending impulses to the SUT: SendDataActions
- receiving responses from the SUT: ReceiveDataEvents
- checking values

For each of those test models, UTML defines a graphical representation, thus providing different views on the test model.

Testing and Test Control Notation, version 3

The testing and test control notation, version 3 (TTCN-3) is a test specification and implementation language to define test procedures for black-box testing. TTCN-3[11,13,14] is being developed from 1998 on by a European Telecommunications Standards Institute (ETSI) experts team as successor language for TTCN. Its development was driven by industry and research in order to obtain a single test notation for black-box testing needs. In comparison to TTCN, TTCN-3 provides several additional concepts like dynamic test configurations, procedure-based communication, and module control part. By means of TTCN-3, a tester is able to specify tests at an abstract level and focus on the definition of the test cases rather than test system adaptation and execution. TTCN-3 enables a systematic and specification-based test development for various kinds of tests, including functional, scalability, load, interoperability, robustness, regression, system, and integration testing.

In spite of the textual TTCN-3 core language (CL), presentation formats can also be taken as front ends of the language. The tabular and graphical formats called tabular presentation format for TTCN-3 (TFT) and graphical presentation for TTCN-3 (GFT) are standardized formats of TTCN-3. Other presentation formats can be added according to the needs of the users. Furthermore, the core language of TTCN-3 provides interfaces to referenced data that are defined in other description languages. For that, types and values such as defined in the Abstract Syntax Notation One (ASN.1) or interface definition language (IDL) can be imported to TTCN-3.

The TTCN-3 core language (CL) is a modular language and is similar to a common programming language. In contrast to common programming languages, TTCN-3 also contains concepts necessary to specify test artifacts like verdicts, timers, data matching, dynamic test configuration, encoding, and synchronous and asynchronous communication. A TTCN-3 test specification typically consists of:

- test data and templates definition,
- function and test case definitions for test behavior, and
- test control definition for the execution of test cases.

Today in several industrial domains TTCN-3 has been used and is accepted as a mature test language and technique for manual test specification and automatic test execution and the target output notation for model-based test generation methods and tools. Nevertheless, it is a standard that is maintained and evolved by an expert team at ETSI that takes into account observation and proposals for correction and improvements and is producing new versions of the multipart standard. The latest version has been published in June 2009. Further technical details and organizational information can be found at the TTCN-3 homepage.[15] For a more in-depth introduction to MBT and a discussion of the applications please see the two other entries on model-based testing.

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. Abdurazik, A.; Offutt, J. *Using UML Collaboration Diagrams for Static Checking and Test Generation Conference*, Springer: Berlin, LNCS 1939, 2000.
2. Briand, L.; Labiche,Y. A UML-based approach to system testing. J Software Syst. Modeling **2002**, *1*, 10–42.
3. Jeron, T.; Morel, P. Test generation derived from model-checking. In 11th International Conference on Computer Aided Verification, Trento, Italy, 1999.
4. Kim, Y.G. et al. Test cases generation from UML state diagrams. IEE Proc. Software **1999**, *146* (4).
5. Beck, K. *Test-Driven Development: By Example*; Addison-Wesley: Boston, MA, 2002.
6. Publically available TTCN-3 test suites, http://www.ttcn-3.org/PublicTTCN3TestSuites.htm (accessed August 2009).
7. Fujiwara, S.; Bochmann, G., et al. Test selection based on finite state models. IEEE Trans. Software Eng. **1991**, *17* (6), 591–603.
8. Utting, M.; Pretschner, A.; Legeard, B. A Taxonomy of Model-based Testing. Working Paper: 04/2006, The University of Waikato, Hamilton, New Zealand, 2006.
9. Busch, M., et al. Model transformers for test generation from system models. In Conquest 2006; Hanser Verlag: Berlin, Germany, 2000.
10. OMG, The Unified Modeling Language, UML 2.1.1 Superstructure Specification, http://www.omg.org/spec/UML/2.3/Superstructure/PDF, 2009 (accessed August 2009).
11. 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.
12. TPTP, Test & Performance Tools Platform, http://www.eclipse.org/tptp (accessed August 2009).
13. Willcock, C.; Deiß, T.; Tobies, S.; Schulz, S.; Keil, S.; Engler, F. *An Introduction to TTCN-3*, 1st Ed.; Wiley & Sons, 2005.
14. Pretschner, A.; Philipps, J. *10 Methodological Issues in Model-based Testing*; Springer: Heidelberg, LNCS 3472, 2005.
15. TTCN-3 Home Page, http://www.TTCN-3.org (accessed August 2009).