Model-Based Testing: Trends

Ina Schieferdecker
Jürgen Großmann
Marc-Florian Wendland
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. This entry examines the use of MBT to solve problems and answer challenges in the field of test automation, test quality improvement, and test efficiency. It reviews the application of MBT in different domains and trends and challenges for the further development of MBT approaches. Subsequently, tools supporting MBT processes are presented and a discussion on the MBT return of investment is given.

MBT TRENDS AND CHALLENGES

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 (MBE) techniques by industrial-grade software engineering companies provides a solid basis to introduce and apply MBT approaches and will hopefully lead to a quite larger acceptance of model-based techniques.

Selected Application Areas for MBT

In the following sections, we provide an overview of how MBT is applied to different industrial domains. These domains include

- Automotive
- Telecommunications
- Automation
- Production engineering
- Enterprise Business Application (EBA)

For each of the domains we consider the state of the art and describe major challenges that have to be typically addressed to successfully implement MBT strategies in the respective industrial domains.

MBT in the automotive domain

Software in the automotive domain is essentially developed to be deployed on embedded devices as Electronic Control Units (ECUs). Therefore, testing automotive software requires a methodology that can match the high level of requirements in that context. MBT is viewed as a potential approach for addressing that task. However, it is still weakly applied for several reasons, including the level of maturity of the tools provided in that domain, the resilience of people to changes, and the challenging nature of the task itself.

Nevertheless, MBT is used for functional and robustness testing in the automotive domain and is continuously gaining popularity. The tests aim at verifying functional models, implementation models, software elements (components, systems, modules), and finally integrated ECUs, i.e., the combination of software and hardware components.

Modeling for MBT in the automotive domain is generally based on the (E)FSM approach, implemented by tools such as MATLAB/Simulink or through UML (Unified Modeling Language) state charts. In some cases, other tools are used for specific parts of the modeling process. For example, the Classification Tree Method (CTM), implemented in tools such as CTE XL, is used for modeling test data.

MBT in the telecommunications domain

The telecommunications domain is historically one of the first industries in which MBT has been applied, mainly for hardware testing. This leads to up to tenfold productivity gains compared to a manual test production process.[1] On the software side, MBT usage has gained less popularity in the telecommunication industry because of discouraging results obtained in real-life case studies.[2] MBT is mainly used in that context for automated generation of conformance and functional tests. Behavior is modeled using (E)FSMs and Message Sequence Charts (MSC) or equivalent approaches, whereas data is modeled as UML equivalence partition classes. For conformance testing, the TTCN-3 (Testing and Test Control Notation version 3[3]) is widely used. However, TTCN-3 is used in telecommunications via its core notation format, which is somewhat different from what one might expect for a UML-like model-based approach. Besides TTCN-3, the UML Testing Profile (UTP) is gaining popularity in the telecommunications domain for MBT.

Most efforts of applying MBT in the telecommunications domain have focused on automated generation of test cases from (annotated) models of the SUT (system under test). Another interesting approach that is gaining popularity consists of using MBT as a means for supporting manual test development. This is viewed as a promising alternative or complementary element to face issues inherent to automated test case generation.

Academic usage. Several case studies describing the application of MBT to the telecommunications domain have been conducted in the academic context. Most of those case studies feature small-size "toy" examples and demonstrate the high potential of MBT in terms of productivity improvements. Belinfante et al.[4] describe an example case study featuring the T- or X-model-based test generation tool. Numerous other such examples exist, but listing them here would go beyond the scope of this entry. A general impression emerging from academic publications on MBT in telecommunications is that their application has been very successful and bears the potential for considerable benefits to the whole software development process.

Industrial usage. While MBT has been keenly adopted by large numbers of academic experts on testing, it has not reached the same level of popularity in the telecommunication industry yet. One of the difficulties faced by MBT in telecommunications is the fact that the combination of concurrent and distributed behavior along with a very high level of configurability of the software systems involved quickly leads to a state space explosion problem. Strategies proposed for addressing that issue (e.g., abstraction and exclusion[5]) weaken what is supposed to be the main argument for MBT, i.e., the ability to automatically generate an optimal number of test cases by analyzing the complete SUT model and not just a part or an abstract representation thereof.

In spite of those challenges, the telecommunications industry is working on new methods for applying MBT. This is illustrated by projects such as the ITEA TT-Medal project and the ITEA D-Mint project in which key players from that domain are involved and combine their ideas with test tool vendors and other testing experts to address those issues and make MBT a reality in the telecommunications industry.

MBT in the automation domain

In the traditional automation products domain, manufacturers are confronted with an increased customer demand for new features like fieldbus connectivity that has driven an increase in product complexity. New features should be added in addition to old goals such as safety and reliability. MBT is seen as a vital part of a process to develop high-quality products with a short time to market. The intention is to have full overnight testing of the complete system functionality from the start of the implementation phase in order to identify faults early in the implementation phase.

In the automation domain, MBT must also be able to deal with non-object-oriented systems. Another issue is the testing of variants. There are many variants of the automation products, e.g., with different consumer connections and different running modes and boundary conditions. The test models have to be enriched with additional information to handle such variants.

Applied methods include model-based statistical testing (MBST) and feature-oriented testing with MATLAB/Simulink.

MBT in the product engineering domain

Currently, the product engineering domain is based on a traditional Waterfall model of the software development cycle. The original software development cycle was customized to be able to create and coordinate the mechatronic systems development. The basic phases of the software development cycle were kept, and similar phases added to manage the hardware development cycle. The result is a development process that merged the hardware and software development for its final testing and validation. Nowadays, testing is carried out manually; human intuition is usually used in the testing phase instead of a detailed methodological process.

Often, UML models are used for documentation of information only. Innovative companies started to complete these UML models and derive test model in order to create test cases systematically. Automatic execution of tests is not required but an automated documentation is.

From a market point of view, it is expected that the application of MBT methodologies to the production engineering development process will bring improvements in the following metrics:

- Improvement of the machine tool RAM (reliability, availability, maintainability) parameters.
- Improvement in the definition of the system specifications.
- Reduction in the development cost and time. Since the early stages of the development, with an initial model,

the engineers will know if the modeled solution is correct, and errors will not go ahead.
- Improvement of the test phase quality and reduction of engineers' dedication in the testing phase (test cases will be systematically executed and documented).

MBT in the EBA domain

The EBA domain is characterized by a broad field of application architectures that ranges from enterprise resource planning (ERP) architectures or enterprise application software (EAS) architectures, to large-scale business-to-customer (B2C) and business-to-business (B2B) application architectures. Its main ambition is to support the business activities of big enterprises by means of information technology. Thus, the explosive emergence of Internet technologies have speeded up the field of application, and extended the pure size and distribution of application to world-spanning infrastructures. The development of EBA was mainly influenced by big software development companies like IBM, Microsoft, Oracle, and SAP, as well as by companies that provide large business application infrastructures like Amazon, Google, etc. The software application itself is typically characterized by its modularity, distribution, heterogeneity, and a common service-based terminology. Thus, in the 1990s, efforts to standardize service interface descriptions, modeling languages, and technologies had already started and led to a large number of mature standards.[6–9]

The development of enterprise applications imposes some unique challenges to software testing due to the heterogeneity, distribution, and complexity of EAS.

- Heterogeneity: EBA aims to provide an infrastructure that crosses the borders of individual companies and enables support for B2B and B2C relationships. Despite the efforts that have recently been spent in standardization and harmonization, interoperability and standard conformance remain a major issue for testing. Thus, MBT approaches especially have to address conformance and interoperability tests.
- Different kind of user interfaces: EBA shows a different kind of graphical user interfaces (GUIs) that—respecting the growing proliferation of Web 2.0 technologies—take over important parts of the overall behavior. GUIs are difficult to test in general and with MBT in particular.
- No separate test environment: due to its pure size it is nearly impossible to provide a separate test bed to test EBA. Thus, testing has to be established and integrated in the productive systems with all the compromises concerning controllability and assessability of such processes.
- Unobservable global states: the global state of a software application is determined not only by its actual local process step or the values of its variables at runtime but also by the stored data that the system is able to access. This data affects the state because it might influence the next step of computation. Moreover, the distribution of SOA components leads to a decomposition of data and behavior. Consequently, the global state of an SOA application is hardly observable anymore. Thus, simple MBT approaches that are based on transparent access to the states of the SUT are not adequate.
- Dynamic system states: applications are highly complex in functionality and data. Thus, the reproducibility of tests is hard to achieve. It is practically impossible to drive large-scale EBA systems back into a defined initial state for testing. Even in a developing phase the necessary efforts are too high. As each test run changes the system state, a rerun of the same test case will most likely find the system in a different condition and thus a testing strategy for MBT has to address dynamic state changes during test execution.
- Challenging test data provision: system state changes heavily depend on input data (transactional data) and system internal data (master data and state variables). EBA test data have complex structure (containing dozens of required fields) and originate from heterogeneous systems out of the control of the testers. Thus, MBT approaches especially have to consider test data provision and the proper handling of data, which are not under the control of the test system.

Attempts to ease the identification and specification of tests for EBA have been carried out by different research projects (COTE,[10] AGEDIS,[11] and ModelPlex[12]) as well as by industrial companies (SAP and T-Systems).[13] A requirements catalog for information systems MBT that can be used as a basis for improving methods as well as a guide for the development of new methods and tools has been presented in Ref. [14]. In Ref. [13], an approach based on UML, U2TP, and TTCN-3 is introduced that helps formalizing the test artifacts and thus supports offshoring processes for test case specification and test execution. In Ref. [15], challenges concerning the problems of providing appropriate test data are listed and in Ref. [16] a model-driven solution to address dynamic discovery and binding for the integration of web services during testing is proposed.

While UML is more dedicated to technical issues, BPML and BPEL provide a more business-oriented perspective to EBA. State-of-the-art BPEL server provides the capabilities to simulate the interactions between different service providers and thus to test BPEL processes outside the productive environment.

Commercial tools that are directly dedicated to support MBT for EBA are rarely available. HP offers the so-called BPT tool suite[17] as an extension to the HP Quality Center Tool Suite but it is not based on standardized notions.

However, most of the attempts mentioned above are based on well-known modeling paradigms like UML or BPML; thus, especially UML-based MBT approaches like Smartesting Test Designer,[18] Matelo,[19] and QTronic[20] are applicable to the EBA domain in principle.

MBT Trends

This section reviews developments in deploying or improving MBT methods. The trends include approaches to improve test processes and to improve test automation.

Test process improvement

Facing the increasingly complex difficulties regarding software development, which require teams and clear structures for work, in the past few years, several software development process models have emerged. These process models coordinate the activities and define the clear structure needed for modern software development. Besides models like the "V-Modell XT," the "rational unified process," or the "extreme programming," there exists a so-called "W-Model."[6]

This model integrates inter alia the phases of requirements analysis, design, and implementation with the phases of test specification, execution, and evaluation.

MBT may be adopted using the principal ideas of this process model. The difference with regular testing is that a model of the system or of the test is used as a basis for testing, which may be executable. These models may already be created out of the requirements and can be used as a basis for a later code generation.

Still, there exists only little experience in development processes that integrate MBT.[21,22] Designing such processes implies certain difficulties:

- Whereas for conventional software development, several standards exist, for MBT no such standards are available that could be used as reference.
- A lack of specific experience on particular techniques, their applicability, and constraints for MBT.
- The variations of the applications and, as a consequence, possible variations of the processes are not sufficiently understood.
- The impact of the variation of the enabling technology on the tested product is not always known and this may affect the underlying process.

Industrial experiences in introducing MBT approaches have been reported in Ref. [23].

Requirements engineering and MBT. Demonstrating test coverage of requirements is important for any black-box-driven testing process. This requires a well-organized testing process, as well as testable requirements. Requirements are often reported as not suitable for testing, because they are, for instance, incomplete or too ambiguous. Ref. [24] describes an approach where requirements specified in the form of UML use cases are the basis for test generation.

Requirements traceability is another active area of research in software engineering. Tracking requirements throughout the software artifacts and development processes ensures that requirements can be respected during development. Ref. [25] describes an approach for the automated production of a traceability matrix. This approach has the following advantages for the software testing process:

- It gives to the generated test cases a clear functional coverage metrics from the viewpoint of the requirements.
- Knowing which requirements are not covered via the MBT process allows one to complete the test suite with some manually designed test cases or to improve the model or test generation strategies in order to fulfill the test objectives.
- It gives valuable feedback on the requirements: some test cases may not be linked with any requirements, possibly showing the lack of expressed requirements.
- Automatically generating the traceability matrix from requirements to test cases implies managing the links between the requirements specification, the model, and the test cases.

Integration of testing into the system development cycle: MDA++. A first step toward a better integration of test development into the system development process has been taken in Ref. [26]. The idea of MDA (model-driven architecture) is that platform-independent models (PIMs) can be automatically transformed into platform-specific models (PSMs), and programming language code can be generated from PSMs. The test software can be modeled and developed in exactly the same way as the functional system software. Abstract testing artifacts are derived and modeled from the existing information in PIMs. These platform-independent test models (PITs) can be transformed to platform-specific test models (PSTs), potentially taking additional information from PSMs. Then, the programming language test code, i.e., the code of the test components of the test system, can be generated from the PSTs.

This approach is depicted in Fig. 1. According to this approach, test components are developed in parallel with the system components as soon as the interfaces and the overall architecture of the system to be developed are defined. However, since the developers still use another language for system modeling than for test models, there is still a gap between the system and test development. Further investigations have thus been done to improve the integration of the test and system development by

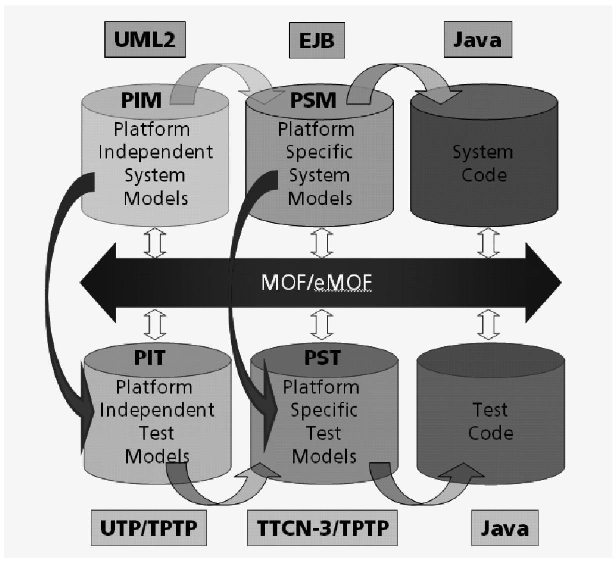
UML2
EJB
Java
PIM
Platform Independent System Models
PSM
Platform Specific System Models
System Code
MOF/eMOF
PIT
Platform Independent Test Models
PST
Platform Specific Test Models
Test Code
UTP/TPTP
TTCN-3/TPTP
Java

Fig. 1 Combining system development and system test according to MDA.

adopting the same specification language for system and test modeling.

Test evolution in relation to system (model) evolution. Testing should be primarily based on requirements and specifications and not on implementations as testing should always be aimed at demonstrating conformance of a product to some predefined specification documents or models. Tools such as Telelogic's DOORS[27] or TNI's Reqtify[28] have become well established for recording and managing requirements. However, these tools do not support semantic relationships between requirements, specifications, design models, and their respective tests. An important question that has yet to be answered is how traceability that is enriched with semantics can be realized between each individual requirement and its respective test artifacts.

Variant and configuration management add another dimension to the requirements management and traceability issues. Variant and configuration management are concerned with product line engineering, which is basically an approach for planning and developing several similar systems concurrently in order to systematically exploit their commonalities. Product line engineering can be subdivided into commonality analysis, which results in a set of common features and system functionality that essentially define the product line, and variability analysis, which defines how each individual product differs from the common core and any of the other products in the same product line. Some of the stated requirements will be attributable to one product, and some other requirements will be attributed to some other product, whereas a number of requirements can be attributed to all products at the same time. Methods that support product line engineering have been subjected to investigation.[29] Product line engineering is always based on a decision model that relates variable options in the requirements with concrete system features of a final product. A decision comprises a domain-specific question that represents the decision to be made, the set of possible answers to the question representing the feasible variants, references to all assets and artifacts including variation points within the product line that are affected by the decision, and a description of the effects on those assets for each possible answer. The decision model is a specification model in the same sense as any other functional or behavioral model. So, requirements management and tracing must be ensured across this dimension as well, and that includes mappings within and across different models and notations along the same lines as is the case for any other "normal" requirement, including testing.

The third dimension comes in the form of system versioning, thus relating specific requirements or specification artifacts to specific versions of one system, or one version of a specific product of a product line. The same requirements and consistency tracing problems are also apparent in that dimension.

A new approach is needed where requirements and test models are semantically related based on a decision model reflecting the product variants, configurations, and versions. So far, research work on product line testing has been done mainly focusing on approaches where tests are being constructed from complete system models resulting after resolving all variation points of a product line. In contrast to that, research should aim at a compositional approach where fixed and variable constraints of a product line are reflected in fixed and variable parts of the tests—the resolution of tests is based on constraint composition techniques—rather than on model composition that encompasses the scalability problem for real systems. This principal idea is depicted in Fig. 2.

Requirements models such as classification trees, use case scenarios, structured text, as well as structural or high-level behavioral models guide the elicitation and capturing of product variants. Every fixed and variable part is associated with a set of system constraints, which are induced for the resulting product once a certain variable part is being selected—constraints for fixed parts are always induced. These constraints can be of local or global nature and are required to hold always (i.e., the constraints are invariants), occasionally (under certain conditions), and temporarily (induced by events in the system). Furthermore, constraints can be defined for various aspects such as structural aspects (interfaces, components, etc.), functional aspects (scenarios, interactions, etc.), timing aspects (responsiveness, continuity, etc.), and alike. Every single constraint is associated with test snippets (being rudimentary test data and/or test behavior) or with complete test cases. Once the set of fixed and variable constraints has been identified, the corresponding set of tests is being identified and composed into complete and

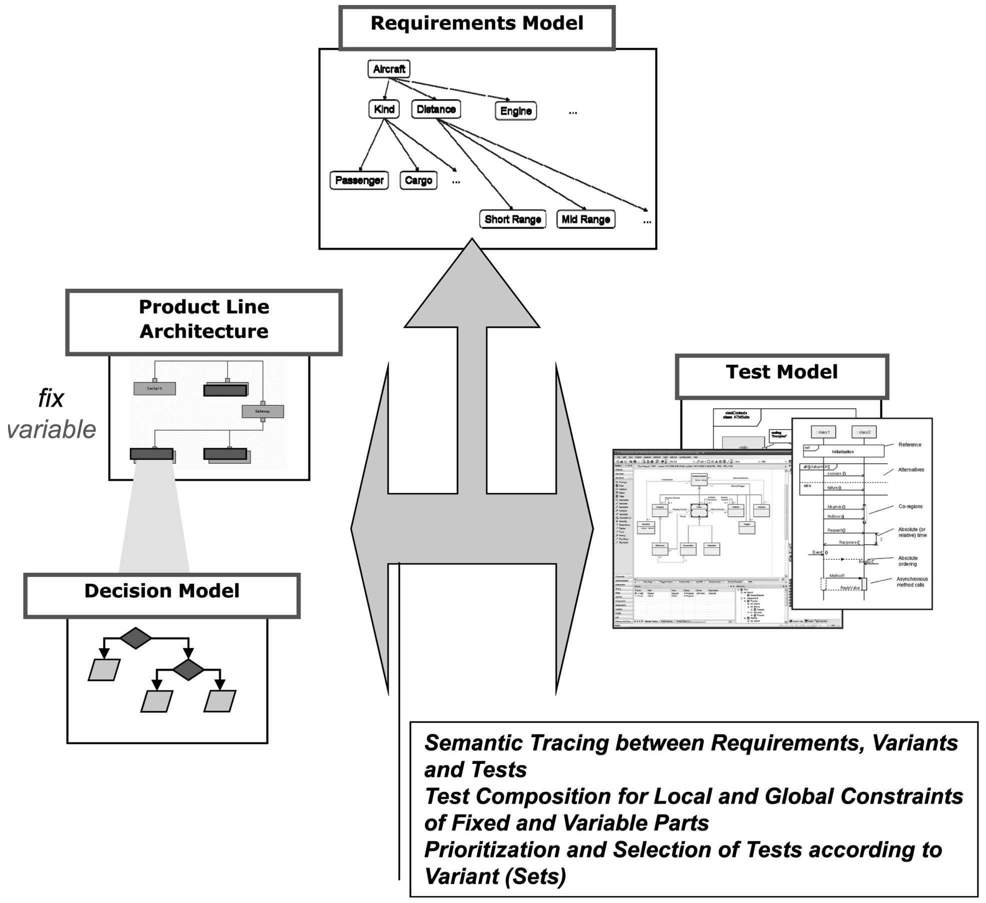
Requirements Model
Aircraft
Kind
Distance
Engine
...
Passenger
Cargo
...
Short Range
Mid Range
...
Product Line Architecture
fix
variable
Decision Model
Test Model
Semantic Tracing between Requirements, Variants and Tests
Test Composition for Local and Global Constraints of Fixed and Variable Parts
Prioritization and Selection of Tests according to Variant (Sets)

Fig. 2 Three-dimensional model of changing requirements through requirements evolution, product line engineering, and versioning.

meaningful test suites. Hence, in contrast to other approaches, one should not construct the resulting overall system model for a product variant, but uses rather a constraint-driven composition and construction of the tests. Such test-first-driven development approach is successfully used for code development (test first paradigm or extreme programming) and should also be successful for product lines.

Test automation improvements

For years, experts have been convinced that test design is the central element to successful and meaningful software testing. In the meantime, however, it has become apparent that testing requires all the typical elements of software engineering: tests are software-based systems themselves and need to be engineered, designed, verified, validated, and executed like any other software-based system. Specialities of test systems involve the ability to control, stimulate, observe, and evaluate the SUT. Although standard development and programming techniques are mostly applicable, specific solutions for test automation respecting their peculiarities, to make testing more effective and efficient, seem reasonable. In the following sections, we review various elements of trends in improving the automation of MBT methods.

Using model transformation techniques. Model transformation provides a key technology to bridge the semantic gap between models. They are used to translate model parts of one model view into another model view on one abstraction level, of one abstraction level into model parts on another abstraction level, or of one modeling technique into another one.

A transformation defines a set of rules used to modify a source model to a target model (Fig. 3). A minor transformation step is a refinement if it describes the relationship between models of the same abstraction level. Transformations can also be used to specify the relations and invariants between the models, which are the base to check the consistency between models and to validate models against each other.

Generally, model transformation can be in three styles:

- Source-driven transformation, where the transformation rule is a simple pattern. The matched elements

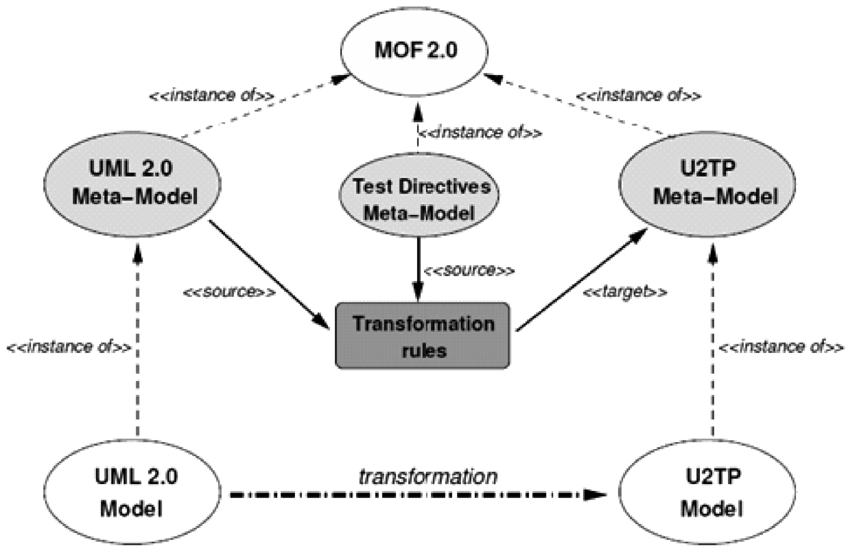

Fig. 3 Test directives for test generation using model transformations.

are transformed to a set of target elements. This style is often used in high-level to low-level transformations (e.g., compilation).

- Target-driven transformation, in which a transformation rule is a complex pattern of source elements. The matched elements are transformed to a simple target pattern. This style is often used for reverse engineering or for performing optimizations.
- Aspect-driven transformation describes the relationship of the semantic concepts, e.g., transforming all imperial measurements to one metric measurement, or replacing one naming system with another naming system.

A tester may develop a test model manually. However, a more ambitious effort than the manual test modeling is to generate the test model automatically. 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.[30] For model-based test generation, source-driven or aspect-driven transformations are used.

The starting point of a test model transformation is an existing system model. However, test-specific requirements directing the test model specification, such as the choice of SUT component, configuration of the test system, selection of test scenarios, specification of timers, invocation of default behavior, setting of test verdicts, definition of time zones, coordination of test components, etc. cannot be derived from the system model. They must be defined explicitly and included in the transformation process.[31]

Test coverage and test selection strategies. In practice, the huge amount of produced test cases in contrast to the limited resources for test execution is a problem that is not new but still requires innovative methods and algorithms to improve test efficiency.

There already exist different approaches on different abstraction levels addressing this topic. Traditionally, test coverage of system requirements is used as a measurement criterion. But it is even not enough to know that a number of tests covered the system model or a list of requirements. The coverage potential of a single test case and the preferred order of test execution during test campaign are needed to optimize the test process and use of resources.

In the context of UML model validation, there exist practical approaches using requirement tracing with tags in the test model.[20] As such models are derived from requirements, it is important to track how different requirements are reflected in the models, on different perspectives, and on different abstraction levels. It is also important to propagate requirements through the test generation and execution processes, such that one can verify what parts of the models and consequently which requirements have been tested and validated.

The requirements can be propagated onto different parts of models to point out a relationship between requirements and model elements. For instance, communication requirements are traced to data models, architectural requirements to architecture models, whereas functional requirements are initially traced to use case models and then to state models.

The approach traces requirements from test cases to models with the use of a Python script, that is, the use of information about the failed requirements obtained from the testing log of Qtronic to trace failed tests back to UML models via requirements. This method shows which requirements failed during testing and to what model elements they are linked. It also enables one to identify which parts of the system model have been covered by a given test set.

Prioritization techniques. System requirements may be changed along system development due to reduction, extensions, adaptations, or other modifications. Hence, it is important to identify those test cases in the test base that are affected by these changes imposing a redefinition of the corresponding test models. A new algorithm has been proposed in Ref. [32] that supports test case priorities based on weights and (feature) potentials (e.g., occurrence, risk, and severity indicators). This enables an overall reduction in the magnitude of testing efforts, while minimizing resources needed for test execution.

The algorithm for the calculation of test case priorities is based on the sum of weight and potential values that have been assigned to the conditions and events in a model that represents the set of test cases in a test campaign. Within the first step all weight values of all possible tests (conditions) have to be calculated by multiplying all single weight values that belong to a single test in the model. In a second step of the algorithm, a factor "F" will be calculated for each test that intends to reflect the relevance of a test case in addition to its test case weight. This factor "F" identifies the portion of condition of a test case together with a consideration of the potential values assigned to test in comparison to the total sum of uncovered conditions and test potentials. The next step is the multiplication of condition weight values and the factor "F" of each test case. The priorities of the test cases are according to the resulting calculated values: the test case with the highest value has the highest priority and so on.

The approach is applicable on models like, e.g., classification trees or system architecture and allows finding an order for test case execution and first results using an extended implementation of, e.g., the classification tree editor CTE XL have shown that most important test cases lead to an early discovering of faults in the main modules.

Test patterns. Patterns are a method for reusing software methods and artifacts at the design, modeling, and implementation level that have been successfully applied in various contexts. Patterns are used to capture experiences, expertise, and facts to improve system quality and shorten the development cycle of software systems. Recent works have proposed a similar approach for the design and implementation of tests systems.[2,5] The approach aims at exploiting knowledge gathered on test development and to combine it with modeling techniques to generate new test solutions. This is particularly interesting with the growing popularity of MBT, which raises the level of abstraction for test design to a degree that it allows reuse of concepts for new solutions.

One application of test patterns on MBT can occur on model-driven manual design of test cases through so-called wizards that can be used to generate elements of test architecture, test data, and test behavior elements based on selected corresponding pattern.

Furthermore, test patterns can be used to guide the test generation process of model-based automated test generation tool by ensuring that the algorithm for traversing the SUT model takes recognized test patterns into account, e.g., for dismissing paths that are known to be inappropriate for certain types of testing.

Statistical testing. MBST is a black-box testing technique that enables the generation of representative tests from the tester's or user's perspective.[33,34] MBST has been extended for risk-based testing[35,36] and applied to safety-critical embedded systems.[37,38] MBST has been used to construct the generic test models and to automatically generate test cases from the concrete test models.

Fig. 4 shows the steps of the MBST approach. In the first stage, a test model is built from the requirements that represent relevant system inputs and usages and the expected system responses. The test automation stage deals with the automated generation, execution, and evaluation of test.

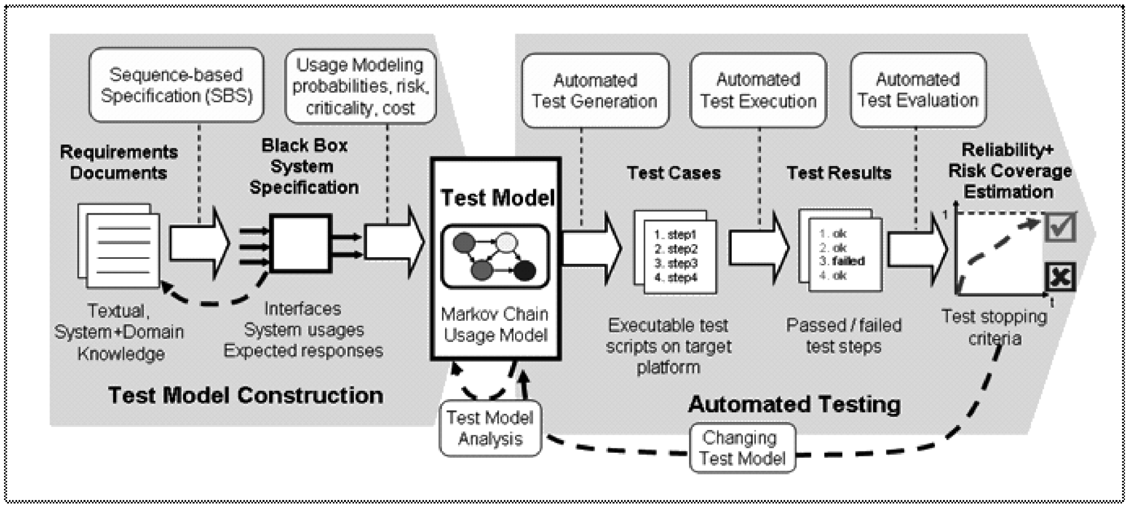
Sequence-based Specification (SBS)
Usage Modeling probabilities, risk, criticality, cost
Automated Test Generation
Automated Test Execution
Automated Test Evaluation
Requirements Documents
Black Box System Specification
Test Model
Test Cases
Test Results
Reliability+ Risk Coverage Estimation
1. step1
2. step2
3. step3
4. step4
1. ok
2. ok
3. failed
4. ok
Textual, System+Domain Knowledge
Interfaces System usages Expected responses
Markov Chain Usage Model
Executable test scripts on target platform
Passed / failed test steps
Test stopping criteria
Test Model Construction
Automated Testing
Test Model Analysis
Changing Test Model

Fig. 4 Steps of model-based statistical testing.

In the test modeling stage, the original system requirements are systematically inspected by the sequence-based specification method to determine the system boundary and the stimuli and responses of the test object. The next step is the sequence enumeration that aims at systematically writing down each possible stimulus sequence, starting with sequences of length one. A sequence of stimuli represents one history of the usage of the SUT. Additionally, every sequence is compared to previously analyzed sequences with regard to equivalence. Two sequences are equivalent if their responses to future stimuli are identical. Illegal and equivalent sequences are not extended in the ongoing sequence enumeration. The enumeration stops if all sequences are illegal or reduced to equivalent sequences.

Certain inconsistencies, missing specifications, unclear requirements, and vague formulations can be found in the requirements document during sequence enumeration. All construction decisions for equivalence, system responses, and requirements coverage are traceable to requirements by following requirements traces attached to the enumerations.

The list of input–output sequences will be mapped to a Mealy machine. The Mealy machine represents the structure of the usage model and describes all possible usages identified from the system requirements. By adding probabilities to the model transitions, the state machine becomes a discrete-time Markov chain (DTMC). The probabilities reflect the expected usage or criticality of system inputs.

Test models have particular states for the initialization (START) and finalization (EXIT) of test cases. A test case is an arbitrary path through the model from START to EXIT. The state START describes the system state at the beginning of a test case. The state EXIT marks the end of a test case and can be reached from all the states where a test case can end.

In the test automation stage, test cases are generated as paths through the test model from the START to the EXIT state. Different strategies for automated test case generation exist:

- **Model coverage tests** make sure that the whole model is covered by test cases. This means that each transition and each state of the test model is tested.
- **Random tests** are randomly generated paths through the test model based on a usage profile, e.g., frequency or criticality.

The transitions of the test model are annotated with scripts for the test runner. Thus, during the generation of a specific test case, the scripts of the transitions lying on the path will be aggregated one after another to build a concrete test case that is executable in the selected test environment.

During test execution, the number of executions with and without failure is counted for each transition. Based on the usage profile and the failure statistics, the reliability of the test object is estimated after each test run. Reliability in MBST is the probability of a no-failure operation.[39] In the case of a criticality profile, safety compliance is estimated instead of reliability.

Evolutionary testing. Evolutionary testing (ET) is a testing method that is used to automatically generate test data by use of evolutionary algorithms. An evolutionary algorithm (EA) is an optimization procedure that has the biological evolution as a model. Individuals are described by their characteristics (i.e., by numerical values that quantify a characteristic). These characteristics are used to measure the suitability of the individuals for a given task. The individuals with the best characteristics are kept for the next iteration; the ones with a bad performance are skipped. Moreover, a set of replacement individuals is generated by slightly modifying the individuals with a good performance. In the course of several runs "the population develops" more and more toward an optimum.

ET uses EAs to generate and optimize test data. The optimization process starts in most cases with randomly generated test data. A fitness function is used to assess the quality of the test data during each optimization step. According to the idea of EA, a set of adequate test data is kept for the next optimization iteration and a set of newly generated test data is introduced. The introduced test data are derived by slightly modifying the fittest results of the previous iteration. Challenges for ET are the specification of useful and operationalizable fitness criteria as well as the definition of appropriate ending conditions for the search.

ET has shown its applicability to many forms of testing, namely, specification testing,[40] structural testing,[41] and execution time testing.[42]

Architecture-driven testing. Architecture-driven testing is a test derivation method that starts from a system model that contains model information on different abstraction levels, the so-called architecture viewpoints.

Looking at the system from different points of view is a way to deal with complexity. Fig. 5 illustrates the viewpoint levels. The top level contains the functional requirements. The logical level shows the system in the form of functional blocks to realize the required functionality without taking into account any technical aspects of their realization, e.g., whether they will be realized in software or hardware, the hardware they are to be running on as single blocks or in combination with others, and the technical resources being available and used in competition with other blocks. Those technical aspects will be taken into account under another viewpoint—technical view. Here, we exactly look at them and assign the functional blocks from the logical view to a realization in or on specific hardware. Another view, the topological view, may pay respect to the aspects where those hardware blocks are

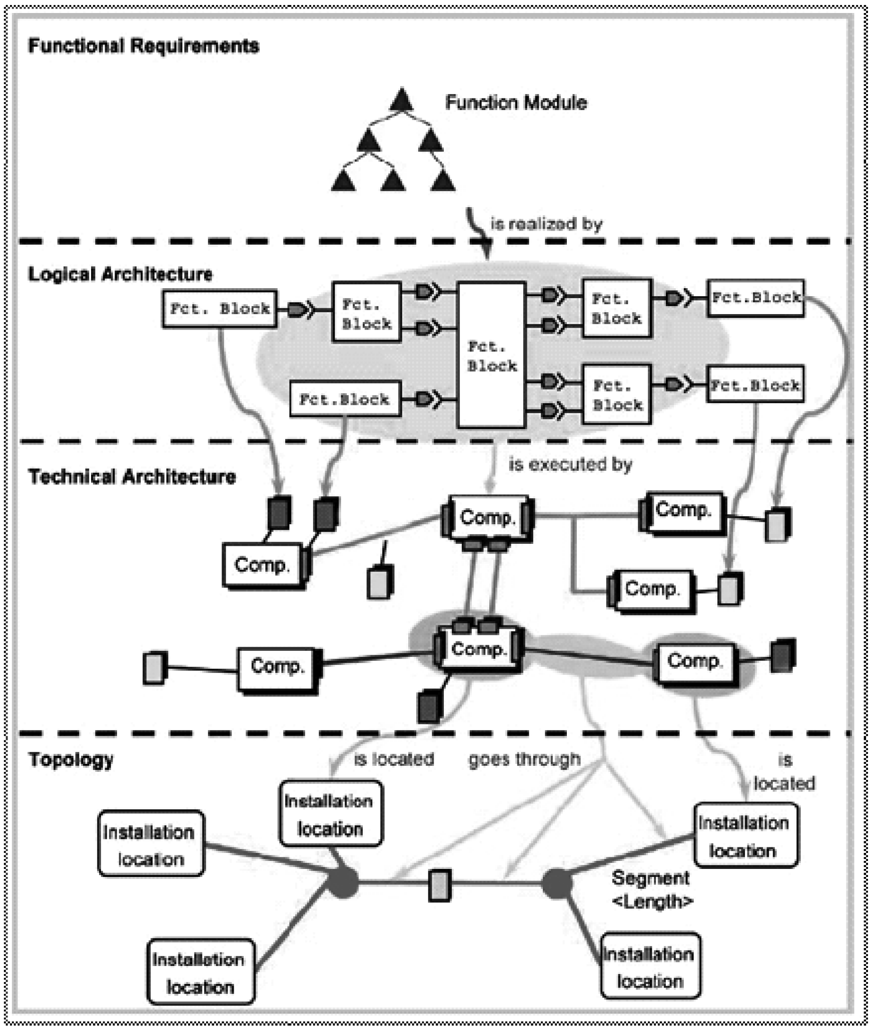

Fig. 5 Function-oriented architecture in the automotive industry.

located in the system, the wire harnesses, and other aspects concerned with the "topology" of the system. Other views may be used and applied as well to get a less complex view on the system by concentrating on a specific aspect of it.

The test generation algorithm starts with the functionality that is given by the functional requirements in a simple form of preconditions, events, and reactions (PER). PERs statements contain references to names of blocks or ports where certain signals are produced or observed. All functional blocks participating in the realization of a specific functional requirement have to be considered. Very important in this approach is also to keep track of relations between elements used within the different views. Within the system design as well as testing phase, it is of relevance to know which requirement is realized by which component under which viewpoint. Within our example of PREEvision we could have links that specify

- which functional requirement is realized by the combined action of which blocks in the logical architectural view,
- which blocks from the logical view are realized or executed on which technical blocks in the technical view, and
- which technical blocks are located where and where does the communication go through.

In order to derive tests, the information about the logical and technical architecture is combined with the information contained in the functional requirements (in form of PERs). The test derivation strategy is based on two steps: 1) decide which functional block (or blocks) should be tested by inspecting the possibility to interfere with other blocks or the possibility to overload and 2) establish which PERs can be applied to the selected functional block in order to validate the functional block in relation with those PERs.

Examples of related functional tests are

1. Validation of PERs when additional ports are stimulated with valid/invalid values. A functional block is tested against a PER while its additional ports (not

regarded by the PER) are simulated too. It is assumed that the functional block should satisfy the functional requirement regardless of what happens with the additional ports.

2. Validation of additional assumed observations. These tests make additional assumptions (while testing a PER) that need to be observed also as reactions. The additional observations are generated based on patterns. For example, the functional blocks with names that are in antithesis (right, left) cannot react at the same time (e.g., if the control of the mirror is set on left then no reaction should be observed on the right side).
3. Monitoring of function blocks that should not have activity when their neighbors are active. These tests simply listen to additional blocks connected to blocks realizing a PER. The additional blocks are not involved in the realization of the PER; therefore, it is assumed that they should not produce any reaction.

MBT Challenges

The status and advances in MBT are not easy to characterize without recognizing the advances in other fields of software engineering and test engineering, for example, improvements in MBE, test automation, and formal test specification. Thus, MBT has a long history and is strongly connected to research activities in the field of protocol testing for telecommunication systems.[43] This directly relates to the application of formal description techniques (FDTs) like Z, SDL,[44,45] and MSCs[46] for the specification of protocols in the early 1990s. FDTs have led to a noticeable paradigm shift in the field of protocol engineering and protocol testing. Research activities focused mainly on test generation methods on the basis of formal models (mostly from Mealy machine models). Currently, relevant algorithms have been developed (the T-Method,[47] the U-Method,[48] the D-Method,[49] the W-Method,[50] and several methods based on UIOs (Unique Inputs/Outputs)).[51–53] However, protocol testing is not subjected to comprehensive research anymore. A good summary of the state of the art of protocol testing is given in Refs. [54, 55].

Deployment challenges

Companies like Motorola started with the application of MBT activities to industrial processes on the basis of the FDT mentioned above. At that time MBE was quite a new idea. Thus, mature tools with standardized interfaces did not exist for MBE as well as for MBT. To be able to start with MBT approaches, the companies had to define their own tool environment, introduce new processes for MBE and MBT, and provide a proper integration with their existing development processes—an almost impossible task. During this time, companies that started with the application of MBT approaches to industrial-grade testing processes reported the following main issues:

- The overall lack of rigorous models for test generation.
- Poor correctness and maturity of the available models.
- Ambiguities of the MBT method: test generation versus test specification.
- Missing mature tools and specification environments.

At the same time, the automotive industry, especially in Germany, developed a slightly different understanding of MBT. Due to the availability of executable software models on the basis of MATLAB/Simulink, the functional properties of embedded systems software could be tested by means of simulation runs. Thus, functional requirements and properties could be tested in the early stages of the development process. To support the testing activities, testing tools[56–58] have been developed that were tailored especially to the needs of the automotive industry. Meanwhile, a couple of tools are commercially available.[59,60] However, similar to the telecommunication industry, the automotive industry used MBT approaches only in a few projects. The reasons have been similar to the ones mentioned above:

- The overall lack of rigorous models for test generation.
- The lack of mature tools and a long-term commercial support for the tools (some of the tests in the automotive industry must be archived for more than 20 years).
- Missing integration (i.e., missing traceability) between the artifacts from different stages of development (requirements engineering, modeling and implementation, testing).
- Missing modularity of models to provide flexible adaption to product lines and software versions.
- Mature test generation algorithms to provide meaningful tests for complex systems.

In recent years, the development and continuing evolution of formal and semiformal description technologies and their standardization have led to a situation where major problems could be solved. Especially the languages and concepts defined by the OMG (Object Management Group) to support their MDE approach[61] have had visible impact on the maturity of modeling tools and the respective infrastructure. Especially the Eclipse project[62] and related technology provider [e.g., for the EMF (Eclipse Modeling Framework) and UML2 Framework] have shown that the OMG concepts can be integrated in a generic tooling platform that is mature enough to yield as a common basis for commercial as well as for academic modeling tools, MBT tools and their related infrastructures (e.g., model repositories).

- Actually there exist a large number of commercial and open source modeling tools that are based on the OMG standards.
- The UML integrates semiformal description technologies to a standardized, self-contained but yet flexible extendable bundle of languages.
- UML artifacts are interchangeable between different tools and tool infrastructures.

Moreover, key technologies for the specification and implementation of tests are formalized, standardized, and provided by commercial tool manufacturers.

- The Test Specification Language TTCN-3 provides a standardized solution for specifying executable tests on different level of abstractions.
- The availability of the UML2 testing profile allows integrating test modeling directly with the UML.

Thus, companies like Motorola switched to yet modern technologies like TTCN-3 and UML 2.x and successfully implemented MBT approaches as part of their MDE approach. For example, the Integrated Dispatch Enhanced Network (iDEN) infrastructure division of Motorola has expanded its MDE usage to 9 out of 12 major network elements such that the use of scenario-based test generation tools yielded an approximately 33% reduction in the effort required to develop test cases.[63]

The market of MDT tools is actually dominated by small-size companies that provide solutions for special fields of application. Even though the maturity of tools has significantly increased and, due to the existence of standardized interfaces, the option to integrate the tools in existing tool environments (e.g., test execution environments and test management environments) are much better than some years ago, MBT is actually not a standard application in industrial-grade processes. We can quote the following reasons that are largely responsible for that:

- The rollout of an MBT approach has serious preconditions: mature processes, approved MBE procedures, skilled test engineers and software developers, etc.
- Today's industrial software development processes feature problems, which in most of the cases are not directly addressed and not solvable by MBT approaches. These include traceability between development artifacts, versioning of development artifacts, and managing supplier chains in system development.

However, IBM, one of the key players for integrated software development solutions, provides MBT solutions[64] that are seamlessly integrated in UML-based development processes. Thus, we can hope for a broader application in the coming years.

Tool challenges

This section reviews challenges for MBT tools that are subsequently grouped into general challenges and specific challenges for test model editors and for test generators.

Tools for MBT can be categorized into two major categories: *model-based test case editors* and *model-based test case generators*. Sometimes a third category is mentioned in this context, namely, *model-based test data generators*. Since MBT and in particular the execution of test cases, created or generated within a model-based approach, has to care for concrete value for the stimuli and expected return values, each major applicable tool integrates a sufficient test data generator to fulfill its purpose, so that data generators are not considered separately.

Model-based test case editors support the manual modeling of abstract or concrete test cases by interpreting a system's specification. Therefore, editors often provide a specific notation, which eases a tester to model the test case. These test cases can be refined automatically, so that they can be executed against the SUT. Editors often provide interfaces or import/export capabilities to other test- or development-specific tools for test management or requirements engineering in order to correlate separately defined information with the test model and test cases. One of the significant disadvantages of model-based editors is that they are not intended to derive test case out of a system's model directly; thus, the identification of test cases is up to a tester's genius or experience. Model-based test case editors focus in particular on the "Test Model Only Approach" and "Independent System and Test Model Approach."

Model-based test case generators are tools that are capable of deriving test cases, test models, or even entire test suites from a system's behavioral model. A key point of these kinds of generators is that the test cases are identified, respectively generated automatically by use of a traversal algorithm, based on configurable coverage criteria like state or requirements coverage. Once the test cases are derived automatically, the generators are similar to the editors. Model-based test case generators focus primarily on the "System Model Only Approach" and "System and Test Model Combined Approach."

We consider the following to be the top challenges for MBT tools:

- Tools (general): seamless integration into software development and maintenance processes
- Editors: user-friendly test model editing, management, and documentation
- Generators: user-controllable scalable test generation

General tool challenges. MBT tools are highly specialized on particular tasks but there are a couple of general challenges that are common. Since the introduction of

MBT usually means to use new utilities or even new IDEs (integrated development environments), it is important to minimize the efforts from the first use, i.e., license costs should be low as well as the efforts to learn and benefit from features. On the other hand, professional support and maintenance is required to guarantee stability for the quality assurances for a longer time in future. For the introduction of the MBT tools it is an advantage if test designers already have experiences with related modeling tools or the target tool platform.

Current MBT tools can be associated with one corresponding modeling language. The language should be appropriate for MBT to be accepted by customers. In this context it is of advantage if the presentation format/syntax has been standardized or at least a standardized terminology and interfaces (application programming interface (API)/GUI) are applied. If the modeling notation has multiple presentation formats (e.g., graphical/textual), switching between them helps to allow giving emphasis to selected aspects and/or different purposes. The tool should support the latest version of the modeling language and be flexible in case of syntax updates or extensions (e.g., import of external-type systems). Domain orientation by allowing, e.g., different profiles would be an advantage too. In case of a universal modeling language there could be a need to specify and check the application of specification guidelines that should be introduced for readability and easier maintenance of existing models.

Test design and development address multiple aspects like configuration, behavior, and data. Due to the different application domains some MBT tools have been specialized only on single problems, e.g., test data partitioning and/or value combinations. Switching between tools that could not exchange model information implies costly manual steps and should be avoided. Thus, it is a challenge for the preferred tool to address all required aspects that arise from different testing types (conformance, interoperability, performance, regression, etc.) and test levels (unit, integration, system, acceptance tests).

Since test development is only one phase in the testing process, MBT tools that primarily address test development should also be connected to the tools concerning other tasks to allow most possible process automation from requirement engineering and management to test execution. Traceability between requirements, system/test models, test definitions, and test observations/verdicts have to be considered to support result analysis and corrections.

For economical reasons customers are looking for means to optimize the test selection and/or ordering. MBT tools should address quantification of test quality using prioritization and/or coverage metrics with respect to requirements. It should be possible, e.g., to avoid or indicate unnecessary redundancy or to apply standardized complexity metrics formula. Finally, it is a challenge for a good MBT tool to generate adequate documentation from the test definition, e.g., for presentation, administration/configuration, and ministration.

Test model editor-specific challenges. Test model editors provide means to model test data, test behavior, as well as individual test cases and complete test suites. In contrast to classic test scripting approaches they aim for a higher degree of abstraction, especially for that kind of development artifacts that are directly modified by the test engineer. Thus, besides the already-mentioned general challenges like integratability, adequateness for the preferred testing levels, support for standardized modeling languages, and maturity, test model editors shall provide user-friendly test model editing, management, and documentation capabilities. This covers support for modeling languages that are directly dedicated to testing, support of different abstraction levels for specification, user-definable test priorities (e.g., risks, probabilities, severity), and simple test management capabilities to order, select, and reuse test specifications.

With the UTP and the TTCN-3 there are dedicated modeling and specification languages for testing, which provide—besides general computational concepts—dedicated test concepts like pattern and templates to express flexible data assessments, timers to specify the temporal upper bound of a test case run, and verdict-assigning statements. Moreover, these languages are standardized, self-documenting to a certain degree, and—in the case of the UTP—integrate seamlessly with existing modeling languages like the UML.

Beside the language-related features, test model editors like the TTworkbench as well as other TTCN-3 tools provide flexible adaption strategies to hide the technical details, which are not relevant for the desired level of abstraction.

To ease test execution, the assessment of test cases, and the identification and removal of errors inside the test specification, a strong relationship between the test specification and test execution environment is eligible. This comprises traceability between test results, test specification, and test documentation as well as proper preparation and user-friendly representation of the test results. On this note, the graphical logging, which is provided by the TTworkbench, is a good example. It reuses concepts and graphical representations that are quite similar to the ones that are used for test specification and is thus self-documenting and easy to understand by the test engineer.

Last but not least, the technical relationship between test model editor and a dedicated test execution environment should not be tight or too strong. The test code generators that are used to obtain test scripts for specific test platforms should be configurable and flexible enough to support different test platforms and test execution languages.

Test generator-specific challenges. Test case generators are tools that are capable of deriving test cases, test models, or even entire test suites from a system's behavioral model. Hence, the key challenge for test generators is that the test generation can be controlled by the tool user by choosing appropriate test generation strategies, by parameterizing the test generation mechanisms, and by combining them so that test cases appropriate to the test plan and test objective can be derived. This includes support for a partial generation of the test model only in order to allow a focused test model generation. It may also include the support for test model optimization strategies by use of, e.g., heuristics to optimize the costs of the required tests resulting from the test model.

Another major challenge is the efficiency and scalability of test generation to cope with system models of industrial size. As test cases basically represent possible or forbidden paths of the system, they are tightly linked to the system state space. Hence, test generation has to cope with the state space explosion problem that can, e.g., be addressed by compositional approaches for test generation, parallelized approaches for test generation, or on-the-fly test generation. These methods have to be applied in order to have performance test generation processes—the test generation time is in particular important for iterative development and test processes where tests have to be generated often.

In addition to these two key challenges and in addition to the general tool challenges given in the "General tool challenges" section, test generator tool should be able to derive test cases from a variety of system model notations and techniques. This challenge results from the fact that various system modeling techniques and dialects are in use. A test generator tool enforcing only selected system model techniques, even if they follow a modeling standard (e.g., certain kinds of UML diagrams only), will be of limited applicability. This challenge also implies that it is typically not enough to generate a test model from a singular, homogeneous system model, but rather to generate a test model from an integrated set of system models (e.g., in UML, from a combination of class models and state machines).

The quantification of the generated test model is another challenge for test case generator tools. The tools have to be in the position to provide coverage and quality metrics for the generated test model as these indicate the quality of the testing that will be applied to the SUT. Coverage has to be related to the test goals used in the definition of the test generation strategies, to the system model from which the test model is derived, and may also be related to the system requirements or the system code (the latter requires traceability including code development or code generation).

Also, the quality of the test model has to be analyzable by the tool user. The test case generator should offer means to analyze the correctness of the test model (e.g., by simulation against the system model), to check for redundancies in the test model (e.g., by detecting duplications or clones), to identify omissions (e.g., by applying coverage analysis methods), to check for the compliance to test model guidelines (e.g., by supporting a customizable set of test modeling guidelines), and to check for the complexity of the test model (e.g., by providing test model-related metrics and their evaluation).

Furthermore, the generated test model needs to be modifiable and extendable by the user. As often not all test aspects can be derived from the system model; certain elements of the test model need to be adapted. This includes the requirement that respective changes in the test model are kept whenever the test model will be regenerated.

Test case generators are often combined with test script generators to support the automated test execution. Therefore, they also face the challenges given for test editors that relate to test script generation and test script execution. Test case generators should likewise be able to generate test scripts in the required technology such as TTCN-3, C/C++, Java, Tcl, Python, etc. While any concrete usage of a test case generator will often use a specific test script technique, different test script techniques are however typically used by the installed test infrastructure for various software systems, product lines, or test kinds.

Further challenges for the generated test scripts relate to their readability, documentation, and traceability (i.e., to the system model and, if possible, to the original system requirements). At the end, the generated test scripts are executed by the testers on the SUT. The test results and traces of the test scripts have to be understood and analyzed to provide appropriate assessments of the SUT including bug reports where needed.

SUMMARY AND OUTLOOK

MBT constitutes a number of technologies, procedures, and approaches with the aim of improving the quality and 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 MBE 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 additionally led to a higher interest on MBT as well.

However, against the background of its application to industrial software development and quality assurance processes, MBT has to prove its potential as a rationalization technology. Thus, the user needs the evidence that—at the latest for a longer term—the investment in MBT returns valuable results.

Consideration of Return on Investment

Concerning the actual return on investment, it is difficult to get concrete numbers, since these are confidential information of the corresponding companies. However, the return on investment has to consider the following factors.

Costs

- MBT tool costs: the costs of acquiring new tools and frameworks in order to implement the MBT approaches in a broader way (even if the existing tools are still used).
- MDE tool costs: the implementation of MBT is strongly coupled with the implementation of MDE processes. Thus, to fully exploit the advantages of MBT, an MDE infrastructure (tools, methodology) is needed.
- Adaption costs to the company's tool and process infrastructure: the MBT methodology and tool platform need to be fine-tuned with respect to the company's development processes, best practices, and domain requirements. Moreover, a fine-tuning for particular projects or at least project categories is necessary.
- Qualification costs: the implementation, maintenance, and integration of MBT procedures require as much higher level of expertise as ordinary test activities. The costs for qualification and training as well as for new experts have to be considered.
- Roll-out costs when changing existing methods, procedures, and best practices.

Gains

- Efficient yet faster procedures to get the test cases for a system under development.
- Better maintenance of test cases.
- Better documentation of the test cases through the use of models (increased transparency).
- Flexible adaption of test cases to new development statuses through the use of automation techniques during test specification (test generation).
- Early discovery of specification errors by means of testing executable software models (simulation models).
- Improvements of the test quality through model-based quality analysis.

Both the quality and quantity of the costs and gains listed above depend on the individual requirements that are defined by the company or the company's application domain. For example, the availability of executable models in the automotive industry is much higher than in any other domain. Thus, the application of MBT approaches that aim for testing software models in simulation is much more valuable for the automotive domain. On the other hand, the telecommunication industry gains more from the OMG standards and thus, the integration of MBT approaches on the basis of standardized technology (the UTP, TTCN-3, etc.) is much easier than in the automotive, medical, or aerospace domain.

Recommendations

MBT is not a self-contained approach. The successful application of MBT is highly dependent on the proper adaption to the existing development infrastructures and processes at the company. One of the great dangers and possible reason for disappointment is too high expectations on the potential of MBT. Often, costs that are connected to the roll out of MBT approaches are underestimated. Especially the requirements regarding the needed infrastructure and the necessary qualification of the personnel are often misclassified.

Studies on the application of MBE/MBT approaches recommend that a roll out of an MBE and/or MBT approach should be introduced by several small pilot projects. These pilot projects usually cover a very small field of expertise in the beginning and aim for simple, realistic, but well-attestable goals (e.g., increased transparency, more efficient (faster) test specification procedures, and better maintenance of test cases). Such a pilot project should be proposed, prepared, executed, and assessed by MBT experts as well as by domain experts to provide the necessary adaption to the company and domain-specific requirements. Pilot project with low-hanging fruit hopefully provides enough motivation to escalate MBT strategies to other projects as well as provides a suitable technological basis for such an escalation.

Experiences with the MBE/MBT technologies have shown that MBT/MBE technologies are actually no commercial-of-the-shelf solution. To successfully implement such technologies they have to be tailored to fit to the company's infrastructure. The tailoring includes adaption to existing tool environment (test management tools, test execution environments, modeling tools, data repositories, etc.) and integration with best practices and existing processes. Efforts and costs of these kinds of integration processes are hard to estimate. Thus, a successfully executed pilot project can help generate valuable results to better estimate such requirements with respect to their necessity, their technical realizability, and the expected costs.

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-Independent 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. Rosaria, S.; Robinson, H. Applying models in your testing process. Inf. Software Technol. September **2000**, *42* (12), 815–824.
2. Hartman, A. Adaptation of model based testing to industry (presentation slides). Agile and Automated Testing Seminar, Tampere University of Technology, Tampere, Finland, August 2006, http://www.cs.tut.fi/tapahtumat/testaus06/alan.pdf (accessed August 2009).
3. TTCN-3 Home Page, http://www.TTCN-3.org (accessed August 2009).
4. Belinfante, A.F.E.; Brinksma, H.; Feenstra, J.; Tretmans, G.J.; de Vries, R.G. Côte de Resyste—Automatic Model-based Testing of Communication Protocols. In Mobile Communications in Perspective—7th Annual CTIT Workshop, 2001. Centre for Telematics and Information Technology, University of Twente, Enschede, The Netherlands, 49–51.
5. El-Far, I.K.; Whittaker, J.A. Model-based software testing. In *Encyclopedia of Software Engineering*, Marciniak, J.J., Ed.; John Wiley & Sons, Inc.: New York, 2001, http://www.geocities.com/model_based_testing/ModelBasedSoftwareTesting.pdf (accessed August 2009).
6. OMG. Business Process Modeling Notation (BPMN) Specification, http://www.bpmn.org (accessed August 2009).
7. OASIS: Web Services Business Process Execution Language Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (accessed April 2010).
8. W3C, Web Services Description Language (WSDL) Specification, March 15 2001, http://www.w3.org/TR/wsdl (accessed April 2010).
9. W3C—WS-CDL Specification—XML Schema Model, http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/#WS-CDL-XSDSchemas (accessed April 2010).
10. COTE: Component Testing using the Unified Modelling Language, http://www.ercim.org/publication/Ercim_News/enw48/jard.html (accessed April 2010).
11. AGEDIS: Automated Generation and Execution of Test Suites for Distributed Component-based Software, http://www.agedis.de/ (accessed April 2010).
12. ModelPlex: MODELling solution for comPLEX software systems, 2006–2009, http://www.modelplex.org/ (accessed April 2010).
13. Braun, A.; Andaloussi, B.S. A test specification method for software interoperability tests in offshore scenarios: A case study. In IEEE International Conference on Global Software Engineering (ICGSE), Florianópolis, Brazil, 2006.
14. Santos-Neto, P.; Resende, R.; Pádua, C. Requirements for information systems model-based testing. In Proceedings of the 2007 ACM Symposium on Applied Computing, Seoul, Korea, March 2007.
15. Wieczorek, S.; Stefanescu, A.; Schieferdecker, I. Test data provision for ERP systems. In Proceedings of ICST'08, IEEE Computer Society: Los Alamitos, CA, 2008.
16. Lohmann, M.; Mariani, L.; Heckel, R. *A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services in Test and Analysis of Web Services*; Springer: Berlin, Heidelberg, 2007.
17. HP: Business Process Testing software, Data sheet, https://h10078.www1.hp.com/cda/hpdc/fetchPDF.do (accessed August 2009).
18. Leirios—SMART TESTING^{TM}: Streamline the test design and improve productivity, 2007, http://www.leirios.com (accessed August 2009).
19. MaTeLo, All4Tec In., http://all4tec.com/ (accessed August 2009).
20. Conformic Qtronic, http://www.conformiq.com/ (accessed August 2009).
21. Bouquet, F.; Debricon, S.; Legeard, B.; Nicolet, J.-D. Extending the unified process with model-based testing. In Workshop on Integration of Model Driven Architecture (MDA) and Verification and Validation (V&V), Genova, Italy, Oct 2006.
22. Engels, G.; Güldali, B.; Lohmann, M. Towards model-driven unit testing. In Workshop on Integration of Model Driven Architecture (MDA) and Verification and Validation (V&V), Genova, Italy, Oct 2006.
23. Ulrich, A. Introducing model-based testing techniques in industrial projects. In MOTES Workshop at Software Engineering, Hamburg, Germany, Mar 2007.
24. Hasling, B.; Goetz, H.; Beetz, K. Model based testing of system requirements using UML use case models. In Proceedings of 1st International Conference on Software Testing, Verification, and Validation, Lillehammer, Norway, Apr 2008.
25. Bernard, E.; Legeard, B. Requirements traceability in the model-based testing process. In MOTES Workshop at Software Engineering, Hamburg, Germany, Mar 2007.
26. Born, M.; Schieferdecker, I.; Kath, O.; Hirai, C. Combining system development and system test in a model-centric approach. In International Workshop on Rapid Integration of Software Engineering Techniques (RISE), Nov 26, 2004, Springer: Luxembourg.
27. Rational DOORS, http://www.telelogic.com/products/doors/index.cfm (accessed August 2009).
28. TNI Reqtify, http://www.tni-software.com/en/produits/reqtify/overview.php (accessed August 2009).
29. Atkinson, C.; Bayer, J.; Bunse, C.; Kamsties, E.; Laitenberger, O.; Laqua, R.; Muthig, D. *Component-Based Product-Line Engineering with the UML*; Addison-Wesley: London, 2001, XXII, 506.

30. Chaparadza, R.; Busch, M. Case study: Derivation of U2TP and TTCN-3 Models from UML2 System Models. In Proceedings of ECMDA 2006, Bilbao, Spain, Sept 2006.
31. Dai, Z.R. An Approach to Model-Driven Testing—Functional and Real-Time Testing with UML 2.0, U2TP and TTCN-3, Promotion, TU Berlin, Fakultät Elektrotechnik und Informatik, Jun 2006.
32. Hoffmann, A.; Rennoch, A.; Schieferdecker, I.; Radziwill, N. A generic approach for modeling test case priorities with applications for test development and execution. In MOTES'09 Workshop at "Informatik 2009," Lübeck, Germany, Sept 2009.
33. Prowell, S.; Trammell, C.; Linger, R.; Poore, J. *Cleanroom Software Engineering: Technology and Process*; Addison-Wesley-Longman: Boston, MA, 1999.
34. Prowell, S. Using Markov chain usage models to test complex systems. Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS), Jan 2005, Maui, Hawaii, 2005.
35. Zimmermann, F.; Kloos, J.; Eschbach, R.; Bauer, T. Risk-based statistical testing: A refinement-based approach to the reliability analysis of safety-critical systems, to appear: EWDC'09, European Workshop on Dependable Systems, Toulouse, France, 2009.
36. Zimmermann, F.; Kloos, J.; Eschbach, R.; Bauer, T. Risiko-basiertes statistisches Testen (in German), 28. TAV Group Meeting of the GI (Gesellschaft für Informatik), Dortmund, Germany, 2009.
37. Kloos, J.; Eschbach, R. Generating system models for a highly configurable train control system using a domain-specific language: A case study. AMOST'09—5th Workshop on Advances in Model Based Testing, Denver, CO, 2009.
38. Bauer, T.; Böhr, F.; Landmann, D.; Beletski, T.; Eschbach, R.; Poore, J. H. From requirements to statistical testing of embedded systems. In Software Engineering for Automotive Systems—SEAS 2007, ICSE Workshops, Minneapolis, MN, 2007.
39. Miller, K.; Morell, L.; Noonan, R.; Park, S.; Nicol, D.; Murril, B.; Voas, J. Estimating the probability of failure when testing reveals no failures. Trans. Software Eng. January **1992**, *18*, 33–43.
40. Tracey, N.; Clark, J.; Mander, K. Automated flaw finding using simulated annealing. In International Symposium on Software Testing and Analysis, Clearwater Beach, FL, 1998; 73–81.
41. Wegener, J.; Baresel, A.; Stahmer, H. Evolutionary test environment for automatic structural testing. Inf. Software Technol. **2001**, *43*, 841–854.
42. Wegener, J.; Grochtmann, M. Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Syst. **1998**, *15*, 275–298.
43. ITU-T Recommendation Z.500, Framework on formal methods in conformance testing. International Telecommunications Union: Geneva, Switzerland, 1997.
44. International Telecommunication Union: Specification and Description Language (SDL), ITU-T Recommendation Z.100, revised, August 2002.
45. International Telecommunication Union: Common Interchange Format for SDL, ITU-T Recommendation Z.106, revised, August 2002.
46. ITU-T Recommendation Z.120, Message Sequence Chart, 2008, http://www.itu.int/itudoc/itu-t/approved/z/index.html. (accessed August 2009).
47. Naito, S.; Tsunoyama, M. Fault detection for sequential machines by transition tours. In Proceedings IEEE Fault Tolerant Computing Conference, Montreal, Canada, 1981.
48. Sabnani, K.; Dahbura, A. A protocol test generation procedure. Computer Netw. ISDN Syst. **1988**, *15*, 285–297.
49. Gonenc, G. A method for the design of fault detection experiments. IEEE Trans. Computer **1970**, *C-19*, 551–558.
50. Chow, T. Testing software designs modeled by finite-state machines. IEEE Trans. Software Eng. **1978**, *SE-4*, 178–187.
51. 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.
52. Shen, Y.N.; Lombardi, F.; Dahbura, A.T. Protocol conformance testing using multiple UIO sequences. In Proceedings of the IFIP Wg6.1 Ninth International Symposium on Protocol Specification, Testing and Verification IX, June 6–9, 1989; Brinksma, E., Scollo, G., Visers, C.A., Eds.; North-Holland Publishing Co.: Amsterdam, The Netherlands, 1989; 131–143.
53. Woojik, C.; Paul, D.A. Improvements on UIO sequence generation and partial UIO sequences. In *Protocol Specification, Testing and Verification*, Linn, Jr., R.J., Uyar, M.U., Eds.; Elsevier: Amsterdam; IFIP: New York, 1992; Vol. 12.
54. Lai, R. A survey of communication protocol testing. J. Syst. Software **2002**, *62* (1), 21–46.
55. König, H. Protocol Engineering Prinzip, Beschreibung und Entwicklung von Kommunikationsprotokollen, B.G. Teubner, Stuttgart Leipzig Wiesbaden, 2004.
56. Lehmann, E. *Time Partition Testing—Systematic Testing of Continuous Behaviour of Embedded Systems*, Dissertation of the Department "Elektrotechnik und Informatik" of the Technical University Berlin, Nov 2003.
57. Conrad, M. *Model-Based Testing of Embedded Software for Automobiles*. Dissertation, Technical University Berlin, Germany, Oct 2004, Deutscher Universitätsverlag: Wiesbaden.
58. Daimler Research results, http://www.systematic-testing.com (accessed August 2009).
59. Reactive Systems, Inc. Reactis, http://www.reactive-systems.com/ (accessed August 2009).
60. PikeTec: Time Partition Testing (TPT) on the websites of the PikeTec Company, http://www.piketec.com/products/tpt.php?lang=en (accessed August 2009).
61. OMG: Model Driven Architecture, http://www.omg.org/mda/ (accessed August 2009).
62. Eclipse IDE, http://www.eclipse.org, release 3.5 (accessed August 2009).
63. Baker, P.; Loh, S.; Weil, F. Model-Driven Engineering in a Large Industrial Context—Motorola Case Study, in Model Driven Engineering Languages and Systems; Springer: Berlin, Heidelberg, 2005; 476–491.
64. IBM Rational Rhapsody TestConductor, http://www.telelogic.com/products/rhapsody/add-on/testconductor.cfm (accessed August 2009).