Model-Based Testing for Embedded Systems

Book · September 2011

CITATIONS
125

READS
5,295

3 authors:

Justyna Zander
HumanoidWay
52 PUBLICATIONS **1,215** CITATIONS
SEE PROFILE

Ina Kathrin Schieferdecker
Technische Universität Berlin
268 PUBLICATIONS **2,675** CITATIONS
SEE PROFILE

Pieter Mosterman
The MathWorks, Inc
241 PUBLICATIONS **4,780** CITATIONS
SEE PROFILE

1

A Taxonomy of Model-Based Testing for Embedded Systems from Multiple Industry Domains

Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman

CONTENTS

1.1 Introduction

This chapter provides a taxonomy of Model-Based Testing (MBT) based on the approaches that are presented throughout this book as well as in the related literature. The techniques for testing are categorized using a number of dimensions to familiarize the reader with the terminology used throughout the chapters that follow.

In this chapter, after a brief introduction, a general definition of MBT and related work on available MBT surveys is provided. Next, the various test dimensions are presented. Subsequently, an extensive taxonomy is proposed that classifies the MBT process according to the MBT foundation (referred to as MBT basis), definition of various test generation techniques, consideration of test execution methods, and the specification of test evaluation. The taxonomy is an extension of previous work by Zander and Schieferdecker (2009) and it is based on contributions of Utting, Pretschner, and Legeard (2006). A summary concludes the chapter with the purpose of encouraging the reader to further study the contributions of the collected chapters in this book and the specific aspects of MBT that they address in detail.

1.2 Definition of Model-Based Testing

This section provides a brief survey of the selected definitions of MBT available in the literature. Next, certain aspects of MBT are highlighted in the discussion on test dimensions and their categorization is illustrated.

MBT relates to a process of test generation from models of/related to a system under test (SUT) by applying a number of sophisticated methods. The basic idea of MBT is that instead of creating test cases manually, a selected algorithm is generating them automatically from a model. MBT usually comprises the automation of black-box test design (Utting and Legeard 2006), however recently it has been used to automate white-box tests as well. Several authors such as Utting (2005) and Kamga, Hermann, and Joshi (2007) define MBT as testing in which test cases are derived in their entirety or in part from a model that describes some aspects of the SUT based on selected criteria. Utting, Pretschner, and Legeard (2006) elaborate that MBT inherits the complexity of the domain or, more specifically, of the related domain models. Dai (2006) refers to MBT as model-driven testing (MDT) because of the context of the model-driven architecture (MDA) (OMG 2003) in which MBT is proposed.

Advantages of MBT are that it allows tests to be linked directly to the SUT requirements, which renders readability, understandability, and maintainability of tests easier. It helps ensure a repeatable and scientific basis for testing. Furthermore, MBT has been shown to provide good coverage of all the behaviors of the SUT (Utting 2005) and to reduce the effort and cost for testing (Pretschner et al. 2005).

The term *MBT* is widely used today with subtle differences in its meaning. Surveys on different MBT approaches are provided by Broy et al. (2005), Utting, Pretschner, and Legeard (2006), and the D-Mint Project (2008), and Schieferdecker et al. (2011). In the automotive industry, MBT describes all testing activities in the context of Model-Based Design (MBD), as discussed for example, by Conrad, Fey, and Sadeghipour (2004) and Lehmann and Krämer (2008). Rau (2002), Lamberg et al. (2004), and Conrad (2004a, 2004b) define MBT as a test process that encompasses a combination of different test methods that utilize the executable model in MBD as a source of information. As a single testing technique is insufficient to achieve a desired level of test coverage, different test methods are usually combined to complement each other across all the specified test dimensions (e.g., functional and structural testing techniques are frequently applied together). If sufficient test coverage has been achieved on the model level, properly designed test cases can be reused for testing the software created based on or generated from the models within the framework of back-to-back tests as proposed by Wiesbrock, Conrad, and Fey (2002). With this practice, the functional equivalence between the specification, executable model, and code can be verified and validated (Conrad, Fey, and Sadeghipour 2004).

The most generic definition of *MBT* is testing in which the *test specification* is derived in its entirety or in part from both *the system requirements and a model* that describe selected functional and nonfunctional aspects of the SUT.

The test specification can take the form of a model, executable model, script, or computer program code. The resulting test specification is intended to ultimately be *executed* together with the SUT so as to provide the test results. The SUT again can exist in the form of a model, code, or even hardware.

For example, in Conrad (2004b) and Conrad, Fey, and Sadeghipour (2004), no additional test models are created, but the already existing functional system models are utilized for test purposes. In the test approach proposed by Zander-Nowicka (2009), the system models are exploited as well. In addition, however, a *test specification model* (also

called *test case specification*, *test model,* or *test design* in the literature (Pretschner 2003b, Zander et al. 2005, and Dai 2006) is created semi-automatically. Concrete test data variants are then automatically derived from this test specification model.

The application of MBT is as proliferate as the interest in building embedded systems. For example, case studies borrowed from such widely varying domains as medicine, automotive, control engineering, telecommunication, entertainment, or aerospace can be found in this book. MBT then appears as part of specific techniques that are proposed for testing a medical device, the GSM 11.11 protocol for mobile phones, a smartphone graphical user interface (GUI), a steam boiler, smartcard, a robot-control application, a kitchen toaster, automated light control, analog- and mixed-signal electrical circuits, a feeder-box controller of a city lighting system, and other complex software systems.

1.2.1 Test dimensions

Tests can be classified depending on the characteristics of the SUT and the test system. In this book, such SUT features comprise, for example, safety-critical properties, deterministic and nondeterministic behavior, load and performance, analog characteristics, network-related, and user-friendliness qualities. Furthermore, systems that exhibit behavior of a discrete, continuous, or hybrid nature are analyzed in this book. The modeling paradigms for capturing a model of the SUT and tests combine different approaches, such as history-based, functional data flow combined with transition-based semantics. As it is next to impossible for one single classification scheme to successfully apply to such a wide range of attributes, selected dimensions have been introduced in previous work to isolate certain aspects. For example, Neukirchen (2004) aims at testing communication systems and categorizes testing in the dimensions of *test goals*, *test scope*, and *test distribution*. Dai (2006) replaces the test distribution by a dimension describing the different *test development phases*, since she is testing both local and distributed systems. Zander-Nowicka (2009) refers to *test goals*, *test abstraction*, *test execution platforms*, *test reactiveness*, and *test scope* in the context of embedded automotive systems.

In the following, the specifics related to *test goal*, *test scope*, and *test abstraction* (see Figure 1.1) are introduced to provide a basis for a common vocabulary, simplicity, and a better understanding of the concepts discussed in the rest of this book.

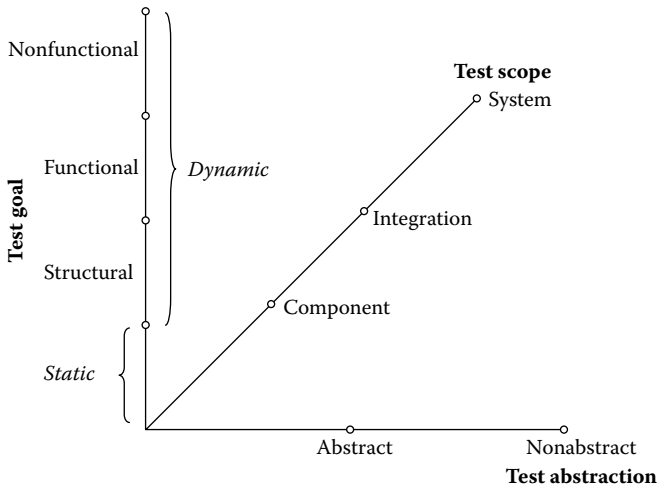

FIGURE 1.1
Selected test dimensions.

1.2.1.1 Test goal

During software development, systems are tested with different purposes (i.e., goals). These goals can be categorized as static testing, also called *review*, and dynamic testing, where the latter is based on test execution and further distinguishes between structural, functional, and nonfunctional testing. After the *review* phase, the test goal is usually to check the functional behavior of the system. Nonfunctional tests appear in later development stages.

- *Static test:* Testing is often defined as the process of finding errors, failures, and faults. Errors in a program can be revealed without execution by just examining its source code (International Software Testing Qualification Board 2006). Similarly, other development artifacts can be reviewed (e.g., requirements, models, or the test specification itself).

- *Structural test:* Structural tests cover the structure of the SUT during test execution (e.g., control or data flow), and so the internal structure of the system (e.g., code or model) must be known. As such, structural tests are also called white-box or glass-box tests (Myers 1979; International Software Testing Qualification Board 2006).

- *Functional test:* Functional testing is concerned with assessing the functional behavior of an SUT against the functional requirements. In contrast to structural tests, functional tests do not require any knowledge about system internals. They are therefore called black-box tests (Beizer 1995). A systematic, planned, executed, and documented procedure is desirable to make them successful. In this category, functional safety tests to determine the safety of a software product are also included.

- *Nonfunctional test*: Similar to functional tests, nonfunctional tests (also called extra-functional tests) are performed against a requirements specification of the system. In contrast to pure functional testing, nonfunctional testing aims at assessing nonfunctional requirements such as reliability, load, and performance. Nonfunctional tests are usually black-box tests. Nevertheless, internal access during test execution is required for retrieving certain information, such as the state of the internal clock.

  For example, during a *robustness test*, the system is tested with invalid input data that are outside the permitted ranges to check whether the system is still safe and operates properly.

1.2.1.2 Test scope

Test scopes describe the granularity of the SUT. Because of the composition of the system, tests at different scopes may reveal different failures (Weyuker 1988; International Software Testing Qualification Board 2006; and D-Mint Project 2008). This leads to the following order in which tests are usually performed:

- *Component:* At the scope of component testing (also referred to as unit testing), the smallest testable component (e.g., a class in an object-oriented implementation or a single electronic control unit [ECU]) is tested in isolation.

- *Integration:* The scope of integration test combines components with each other and tests those as a subsystem, that is, not yet a complete system. It exposes defects in the interfaces and in the interactions between integrated components or subsystems (International Software Testing Qualification Board 2006).

- *System:* In a system test, the complete system, including all subsystems, is tested. Note that a complex embedded system is usually distributed with the single subsystems

connected, for example, via buses using different data types and interfaces through which the system can be accessed for testing (Hetzel 1988).

1.2.1.3 Test abstraction

As far as the abstraction level of the test specification is considered, the higher the abstraction, the better test understandability, readability, and reusability are observed. However, the specified test cases must be executable at the same time. Also, the abstraction level should not affect the test execution in a negative way. An interesting and promising approach to address the effect of abstraction on execution behavior is provided by Mosterman et al. (2009, 2011) and Zander et al. (2011) in the context of complex system development. In their approach, the error introduced by a computational approximation of the execution is accepted as an inherent system artifact as early as the abstract development stages. The benefit of this approach is that it allows eliminating the accidental complexity of the code that makes the abstract design executable while enabling high-level analysis and synthesis methods. A critical enabling element is a high-level declarative specification of the execution logic so that its computational approximation becomes explicit. Because it is explicit and declarative, the approximation can then be consistently preserved throughout the design stages. This approach holds for test development as well. Whenever the abstract test suites are executed, they can be refined with the necessary concrete analysis and synthesis mechanisms.

1.3 Taxonomy of Model-Based Testing

In Utting, Pretschner, and Legeard (2006), a broad taxonomy for MBT is presented. Here, three general *classes* are identified: *model*, *test generation*, and *test execution*. Each of the classes is divided into further *categories*. The model class consists of *subject*, *independence*, *characteristics*, and *paradigm categories*. The test generation class consists of *test selection criteria* and *technology categories*. The test execution class contains execution options.

Zander-Nowicka (2009) completes the overall view with test evaluation as an additional class. Test evaluation refers to comparing the actual SUT outputs with the expected SUT behavior based on a test oracle. Such a test oracle enables a decision to be made as to whether the actual SUT outputs are correct. The test evaluation is divided into two categories: *specification* and *technology*.

Furthermore, in this chapter, the test generation class is extended with an additional category called *result of the generation*. Also, the semantics of the class model is different in this taxonomy than in its previous incarnations. Here, a category called MBT basis indicates what specific element of the software engineering process is the basis for MBT process.

An overview of the resulting MBT taxonomy is illustrated in Figure 1.2. All the categories in the presented taxonomy are decomposed into further elements that influence each other within or between categories. The "A/B/C" notation at the leaves indicates mutually exclusive options.

In the following three subsections, the *categories* and *options* in each of the *classes* of the MBT taxonomy are explained in depth. The descriptions of the most important *options* are endowed with examples of their realization.

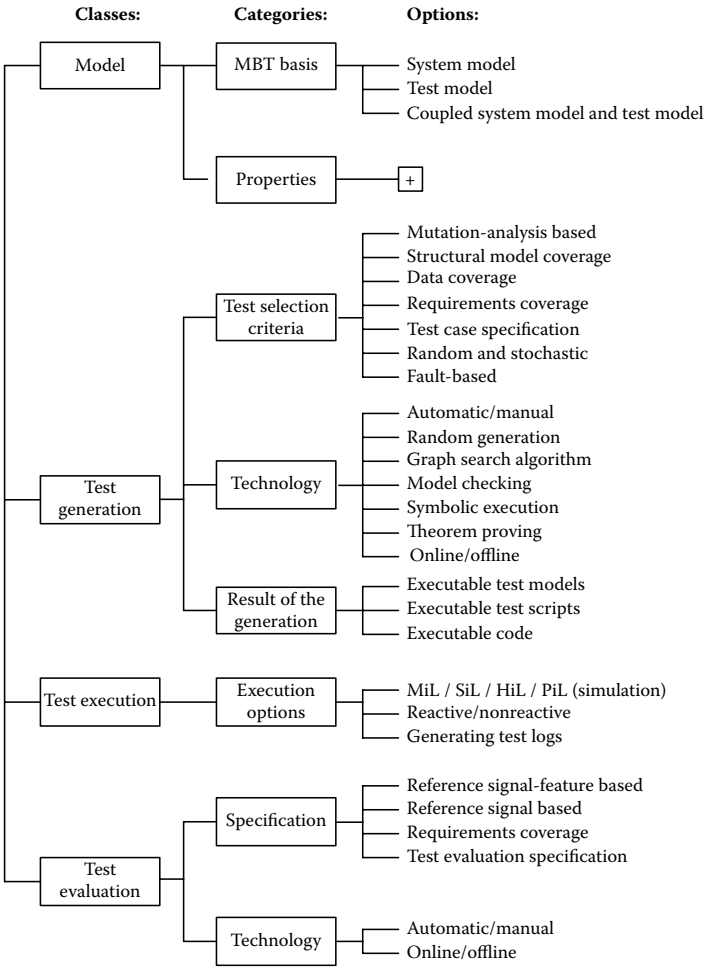

FIGURE 1.2
Overview of the taxonomy for Model-Based Testing.

1.3.1 Model

The models applied in the MBT process can include both system-specific and test-specific development artifacts. Frequently, the software engineering practice for a selected project determines the basis for incorporating the testing into the process and thus, selecting the MBT type. In the following, selected theoretical viewpoints are introduced and join points between them are discussed.

To specify the system and the test development, the methods that are presented in this book employ a broad spectrum of notations such as Finite State Machines (FSM) (e.g., Chapter 2), Unified Modeling Language (UML®) (e.g., state machines, use cases), UML Testing Profile (UTP) (see OMG 2003, 2005), SysML (e.g., Chapter 4), The Model Language (e.g., Chapter 5), Extended FSM, Labeled State Transition System notation, Java (e.g., Chapter 6), Lustre, *SCADE*® (e.g., Chapter 7), B-Notation (e.g., Chapter 8), Communication Sequence Graphs (e.g., Chapter 9), Testing and Test Control Notation, version 3 (TTCN-3) (see ETSI 2007), TTCN-3 embedded (e.g., Chapter 12), Transaction Level Models, Property Specification Language, SystemC (e.g., Chapter 22), Simulink® (e.g., Chapter 12, Chapter 19, or Chapter 20), and so on.

Model-Based Testing basis

In the following, selected options referred to as the MBT basis are listed and their meaning is described.

- *System model*: A system model is an abstract representation of certain aspects of the SUT. A typical application of the system model in the MBT process leverages its behavioral description for derivation of tests. Although this concept has been extensively described in previous work (Conrad 2004a; Utting 2005), another instance of using a system model for testing is the approach called architecture-driven testing (ADT) introduced by Din and Engel (2009). It is a technique to derive tests from *architecture viewpoints*. An architecture viewpoint is a simplified representation of the system model with respect to the structure of the system from a specific perspective. The architecture viewpoints not only concentrate on a particular aspect but also allow for the combination of the aspects, relations, and various models of system components, thereby providing a unifying solution. The perspectives considered in ADT include a functional view, logical view, technical view, and topological view. They enable the identification of test procedures and failures on certain levels of detail that would not be recognized otherwise.

- *Test model:* If the test cases are derived directly from an abstract test model and are decoupled from the system model, then such a test model is considered to constitute the MBT basis. In practice, such a method is rarely applied as it requires substantial effort to introduce a completely new test model. Instead, the coupled system and test model approach is used.

- *Coupled system and test model:* UTP plays an essential role for the alignment of system development methods together with testing. It introduces abstraction as a test artifact and counts as a primary standard in this alignment. UTP is utilized as the test modeling language before test code is generated from a test model. Though, this presupposes that an adequate system model already exists and will be leveraged during the entire test process (Dai 2006). As a result, system models and test models are developed in concert in a coupled process. UTP addresses concepts, such as test suites, test cases, test configuration, test component, and test results, and enables the specification of different types of testing, such as functional, interoperability, scalability, and even load testing.

  Another instantiation of such a coupled technique is introduced in the Model-in-the-Loop for Embedded System Test (MiLEST) approach (Zander-Nowicka 2009) where Simulink system models are coupled with additionally generated Simulink-based test models. MiLEST is a test specification framework that includes reusable test patterns, generic graphical validation functions, test data generators, test control algorithms, and an arbitration mechanism all collected in a dedicated library.

  The application of the same modeling language for both system and test design brings about positive effects as it ensures that the method is more transparent and it does not force the engineers to learn a completely new language.

A more extensive illustration of the challenge to select a proper MBT basis is provided in Chapter 2 of this book.

1.3.2 Test generation

The process of test generation starts from the system requirements, taking into account the test objectives. It is defined in a given test context and results in the creation of test cases. A number of approaches exist depending on the test selection criteria, generation technology, and the expected generation results. They are reviewed next.

1.3.2.1 Test selection criteria

Test selection criteria define the facilities that are used to control the generation of tests. They help specify the tests and do not depend on the SUT code. In the following, the most commonly used criteria are investigated. Clearly, different test methods should be combined to complement one another so as to achieve the best test coverage. Hence, there is no best suitable solution for generating the test specification. Subsequently, the test selection criteria are described in detail.

- *Mutation-analysis based:* Mutation analysis consists of introducing a small syntactic change in the source of a model or program in order to produce a mutant (e.g., replacing one operator by another or altering the value of a constant). Then, the mutant behavior is compared to the original. If a difference can be observed, the mutant is marked as killed. Otherwise, it is called equivalent. The original aim of the mutation analysis is the evaluation of a test data applied in the test case. Thus, it can be applied as a foundational technique for test generation. One of the approaches to mutation analysis is described in Chapter 9 of this book.

- *Structural model coverage criteria:* These exploit the structure of the model to select the test cases. They deal with coverage of the control-flow through the model, based on ideas from the flow of control in computer program code.

  Previous work (Pretschner 2003) has shown how test cases can be generated that satisfy the modified condition/decision coverage (MC/DC) coverage criterion. The idea is to first generate a set of test case specifications that enforce certain variable valuations and then generate test cases for them.

  Similarly, safety test builder (STB) (GeenSoft 2010a) or Reactis Tester (Reactive Systems 2010; Sims and DuVarney 2007) generate test sequences covering a set of Stateflow® test objectives (e.g., transitions, states, junctions, actions, MC/DC coverage) and a set of Simulink test objectives (e.g., Boolean flow, look-up tables, conditional subsystems coverage).

- *Data coverage criteria:* The idea is to decompose the data range into equivalence classes and select one representative value from each class. This partitioning is usually complemented by a boundary value analysis (Kosmatov et al. 2004), where the critical limits of the data ranges or boundaries determined by constraints are selected in addition to the representative values.

  An example is the MATLAB® Automated Testing Tool (MATT 2008) that enables black-box testing of Simulink models and code generated from them by Real-Time Workshop® (Real-Time Workshop 2011). MATT furthermore enables the creation of custom test data for model simulations by setting the types of test data for each input. Additionally, accuracy, constant, minimum, and maximum values can be provided to generate the test data matrix.

  Another realization of this criterion is provided by Classification Tree Editor for Embedded Systems (CTE/ES) implementing the Classification Tree Method (Grochtmann and Grimm 1993; Conrad 2004a). The SUT inputs form the classifications in the roots of the tree. From here, the input ranges are divided into classes according to the equivalence partitioning method. The test cases are specified by selecting leaves of the tree in the combination table. A row in the table specifies a test case. CTE/ES provides a way of finding test cases systematically by decomposing the test scenario design process into steps. Visualization of the test scenario is supported by a GUI.

- *Requirements coverage criteria:* These criteria aim at covering all informal SUT requirements. Traceability of the SUT requirements to the system or test model/code aids in the realization of this criterion. It is targeted by almost every test approach (Zander-Nowicka 2009).

- *Test case definition:* When a test engineer defines a test case specification in some formal notation, the test objectives can be used to determine which tests will be generated by an explicit decision and which set of test objectives should be covered. The notation used to express these objectives may be the same as the notation used for the model (Utting, Pretschner, and Legeard 2006). Notations commonly used for test objectives include FSMs, UTP, regular expressions, temporal logic formulas, constraints, and Markov chains (for expressing intended usage patterns).

  A prominent example of applying this criterion is described by Dai (2006), where the test case specifications are derived from UML models and transformed into executable tests in TTCN-3 by using MDA methods (Zander et al. 2005). The work of Pretschner et al. (2004) is also based on applying this criterion (see symbolic execution).

- *Random and stochastic criteria:* These are mostly applicable to environment models because it is the environment that determines the usage patterns of the SUT. A typical approach is to use a Markov chain to specify the expected SUT usage profile. Another example is to use a statistical usage model in addition to the behavioral model of the SUT (Carter, Lin, and Poore 2008). The statistical model acts as the selection criterion and chooses the paths, while the behavioral model is used to generate the oracle for those paths.

  As an example, Markov Test Logic (MaTeLo) (All4Tec 2010) can generate test suites according to several algorithms. Each of them optimizes the test effort according to objectives such as boundary values, functional coverage, and reliability level. Test cases are generated in XML/HTML format for manual execution or in TTCN-3 for automatic execution (Dulz and Fenhua 2003).

  Another instance, Java Usage Model Builder Library (JUMBL) (Software Quality Research Laboratory 2010) (cf. Chapter 5) can generate test cases as a collection of test cases that cover the model with minimum cost, by random sampling with replacement, by interleaving the events of other test cases, or in order by probability. An interactive test case editor supports creating test cases by hand.

- *Fault-based criteria:* These rely on knowledge of typically occurring faults, often captured in the form of a fault model.

1.3.2.2 Test generation technology

One of the most appealing characteristics of MBT is its potential for automation. The automated generation of test cases usually necessitates the existence of some form of test case specifications.

In the proceeding paragraphs, different technologies applied to test generation are discussed.

- *Automatic/Manual technology:* Automatic test generation refers to the situation where, based on given criteria, the test cases are generated automatically from an information source. Manual test generation refers to the situation where the test cases are produced by hand.

- *Random generation:* Random generation of tests is performed by sampling the input space of a system. It is straightforward to implement, but it takes an undefined period of time to reach a certain satisfying level of model coverage as Gutjahr (1999) reports.

- *Graph search algorithms:* Dedicated graph search algorithms include node or arc coverage algorithms such as the Chinese Postman algorithm that covers each arc at least once. For transition-based models, which use explicit graphs containing nodes and arcs, there are many graph coverage criteria that can be used to control test generation. The commonly used are all nodes, all transitions, all transition pairs, and all cycles. The method is exemplified by Lee and Yannakakis (1994), which specifically addresses structural coverage of FSM models.

- *Model checking:* Model checking is a technology for verifying or falsifying properties of a system. A property typically expresses an unwanted situation. The model checker verifies whether this situation is reachable or not. It can yield counterexamples when a property is falsified. If no counterexample is found, then the property is proven and the situation can never be reached. Such a mechanism is implemented in safety checker blockset (GeenSoft 2010b) or in Embedded*Validator* (BTC Embedded Systems AG 2010).

  The general idea of test case generation with model checkers is to first formulate test case specifications as reachability properties, for example, "eventually, a certain state is reached or a certain transition fires." A model checker then yields traces that reach the given state or that eventually make the transition fire. Wieczorek et al. (2009) present an approach to use Model Checking for the generation of Integration Tests from Choreography Models. Other variants use mutations of models or properties to generate test suites.

- *Symbolic execution:* The idea of symbolic execution is to run an executable model not with single input values but with sets of input values instead (Marre and Arnould 2000). These are represented as constraints. With this practice, symbolic traces are generated. By instantiation of these traces with concrete values, the test cases are derived. Symbolic execution is guided by test case specifications. These are given as explicit constraints and symbolic execution may be performed randomly by respecting these constraints.

  Pretschner (2003) presents an approach to test case generation with symbolic execution built on the foundations of constraint logic programming. Pretschner (2003a and 2003b) concludes that test case generation for both functional and structural test case specifications reduces to finding states in the state space of the SUT model. The aim of symbolic execution of a model is then to find a trace that represents a test case that leads to the specified state.

- *Theorem proving:* Usually theorem provers are employed to check the satisfiability of formulas that directly occur in the models. One variant is similar to the use of model checkers where a theorem prover replaces the model checker.

  For example, one of the techniques applied in Simulink® Design Verifier™ (The MathWorks®, Inc.) uses mathematical procedures to search through the possible execution paths of the model so as to find test cases and counterexamples.

- *Online/offline generation technology:* With online test generation, algorithms can react to the actual outputs of the SUT during the test execution. This idea is exploited for implementing reactive tests as well.

  Offline testing generates test cases before they are run. A set of test cases is generated once and can be executed many times. Also, the test generation and test execution can

be performed on different machines, at different levels of abstractions, and in different environments. If the test generation process is slower than test execution, then there are obvious advantages to minimizing the number of times tests are generated (preferably only once).

1.3.2.3 Result of the generation

Test generation usually results in a set of test cases that form test suites. The test cases are expected to ultimately become executable to allow for observation of meaningful verdicts from the entire validation process. Therefore, in the following, the produced test cases are described from the execution point of view, and so they can be represented in different forms, such as test scripts, test models, or code. These are described next.

- *Executable test models:* Similarly, the created test models (i.e., test designs) should be executable. The execution engine underlying the test modeling semantics is the indicator of the character of the test design and its properties (cf. the discussion given in, e.g., Chapter 11).
- *Executable test scripts:* The test scripts refer to the physical description of a test case (cf., e.g., Chapter 2). They are represented by a test script language that has to then be translated to the executables (cf. TTCN-3 execution).
- *Executable code:* The code is the lowest-level representation of a test case in terms of the technology that is applied to execute the tests (cf. the discussion given in, e.g., Chapter 6). Ultimately, every other form of a test case is transformed to a code in a selected programming language.

1.3.3 Test execution

In the following, for clarity reasons, the analysis of the test execution is limited to the domain of engineered systems. An example application in the automotive domain is recalled in the next paragraphs. Chapters 11, 12, and 19 provide further background and detail to the material in this subsection.

Execution options

In this chapter, execution options refer to the execution of a test. The test execution is managed by so-called test platforms. The purpose of the test platform is to stimulate the test object (i.e., SUT) with inputs and to observe and analyze the outputs of the SUT.

In the automotive domain, the test platform is typically represented by a car with a test driver. The test driver determines the inputs of the SUT by driving scenarios and observes the reaction of the vehicle. Observations are supported by special diagnosis and measurement hardware/software that records the test data during the test drive and that allows the behavior to be analyzed offline. An appropriate test platform must be chosen depending on the test object, the test purpose, and the necessary test environment. In the proceeding paragraphs, the execution options are elaborated more extensively.

- *Model-in-the-Loop (MiL):* The first integration level, MiL, is based on a behavioral model of the system itself. Testing at the MiL level employs a functional model or implementation model of the SUT that is tested in an open loop (i.e., without a plant model) or closed loop (i.e., with a plant model and so without physical hardware) (Schäuffele and Zurawka 2006; Kamga, Herrman, and Joshi 2007; Lehmann and Krämer 2008). The test purpose is prevailingly functional testing in early development phases in simulation environments such as Simulink.

- *Software-in-the-Loop (SiL):* During SiL, the SUT is software tested in a closed-loop or open-loop configuration. The software components under test are typically implemented in C and are either handwritten or generated by code generators based on implementation models. The test purpose in SiL is mainly functional testing (Kamga, Herrmann, and Joshi 2007). If the software is built for a fixed-point architecture, the required scaling is already part of the software.

- *Processor-in-the-Loop (PiL):* In PiL, embedded controllers are integrated into embedded devices with proprietary hardware (i.e., ECU). Testing on the PiL level is similar to SiL tests, but the embedded software runs on a target board with the target processor or on a target processor emulator. Tests on the PiL level are important because they can reveal faults that are caused by the target compiler or by the processor architecture. It is the last integration level that allows debugging during tests in an inexpensive and manageable manner (Lehmann and Krämer 2008). Therefore, the effort spent by PiL testing is worthwhile in most any case.

- *Hardware-in-the-Loop (HiL)*: When testing the embedded system on the HiL level, the software runs on the target ECU. However, the environment around the ECU is still simulated. ECU and environment interact via the digital and analog electrical connectors of the ECU. The objective of testing on the HiL level is to reveal faults in the low-level services of the ECU and in the I/O services (Schäuffele and Zurawka 2006). Additionally, acceptance tests of components delivered by the supplier are executed on the HiL level because the component itself is the integrated ECU (Kamga, Herrmann, and Joshi 2007). HiL testing requires real-time behavior of the environment model to ensure that the communication with the ECU is the same as in the real application.

- *Vehicle:* The ultimate integration level is the vehicle itself. The target ECU operates in the physical vehicle, which can either be a sample or be a vehicle from the production line. However, these tests are expensive, and, therefore, performed only in the late development phases. Moreover, configuration parameters cannot be varied arbitrarily (Lehmann and Krämer 2008), hardware faults are difficult to trigger, and the reaction of the SUT is often difficult to observe because internal signals are no longer accessible (Kamga, Herrmann, and Joshi 2007). For these reasons, the number of in-vehicle tests decreases as MBT increases.

In the following, the execution options from the perspective of test reactiveness are discussed. Reactive testing and the related work on the reactive/nonreactive are reviewed. Some considerations on this subject are covered in more detail in Chapter 15.

- *Reactive/Nonreactive execution:* Reactive tests are tests that apply any signal or data derived from the SUT outputs or test system itself to influence the signals fed into the SUT. As a consequence, the execution of reactive test cases varies depending on the SUT behavior. This contrasts with the nonreactive test execution where the SUT does not influence the test at all.

  Reactive tests can be implemented in, for example, AutomationDesk (dSPACE GmbH 2010a). Such tests react to changes in model variables within one simulation step. Scripts that capture the reactive test behavior execute on the processor of the HiL system in real time and are synchronized with the model execution.

  The Reactive Test Bench (SynaptiCAD 2010) allows for specification of single timing diagram test benches that react to the user's Hardware Description Language (HDL) design files. Markers are placed in the timing diagram so that the SUT activity is

recognized. Markers can also be used to call user-written HDL functions and tasks within a diagram.

Dempster and Stuart (2002) conclude that a dynamic test generator and checker are not only more effective in creating reactive test sequences but also more efficient because errors can be detected immediately as they happen.

- *Generating test logs:* The execution phase can produce test logs on each test run that are then used for further test coverage analysis (cf. e.g., Chapter 17). The test logs contain detailed information on test steps, executed methods, covered requirements, etc.

1.3.4 Test evaluation

The test evaluation, also called the test assessment, is the process that relies on the test oracle. It is a mechanism for analyzing the SUT output and deciding about the test result. The actual SUT results are compared with the expected ones and a verdict is assigned. An oracle may be the existing system, test specification, or an individual's expert knowledge.

1.3.4.1 Specification

Specification of the test assessment algorithms may be based on different foundations depending on the applied criteria. It generally forms a model of sorts or a set of ordered reference signals/data assigned to specific scenarios.

- *Reference signal-based specification:* Test evaluation based on reference signals assesses the SUT behavior comparing the SUT outcomes with the previously specified references.

  An example of such an evaluation approach is realized in MTest (dSPACE GmbH 2010b, Conrad 2004b) or SystemTest™ (MathWorks®, 2010). The reference signals can be defined using a signal editor or they can be obtained as a result of a simulation. Similarly, test results of back-to-back tests can be analyzed with the help of MEval (Wiesbrock, Conrad, and Fey 2002).

- *Reference signal-feature-based specification:* Test evaluation based on features of the reference signal* assesses the SUT behavior by classifying the SUT outcomes into features and comparing the outcome with the previously specified reference values for those features.

  Such an approach to test evaluation is supported in the time partitioning test (TPT) (Lehmann 2003, PikeTec 2010). It is based on the scripting language Python extended with some syntactic test evaluation functions. By the availability of those functions, the test assessment can be flexibly designed and allow for dedicated complex algorithms and filters to be applied to the recorded test signals. A library containing complex evaluation functions is available.

  A similar method is proposed in MiLEST (Zander-Nowicka 2009), where the method for describing the SUT behavior is based on the assessment of particular signal features specified in the requirements. For that purpose, an abstract understanding of a *signal* is defined and then both test case generation and test evaluation are based on this

*A signal feature (also called signal property by Gips and Wiesbrock (2007) and Schieferdecker and Großmann (2007) is a formal description of certain defined attributes of a signal. It is an identifiable, descriptive property of a signal. It can be used to describe particular shapes of individual signals by providing means to address abstract characteristics (e.g., increase, step response characteristics, step, maximum) of a signal.

concept. Numerous signal features are identified, and for all of these, feature extractors, comparators, and feature generators are defined. The test evaluation may be performed online because of the application of those elements that enable active test control and unlock the potential for reactive test generation algorithms.

The division into reference-based and reference signal-feature-based evaluation becomes particularly important when continuous signals are considered.

- *Requirements coverage criteria:* Similar to the case of test data generation, these criteria aim to cover all the informal SUT requirements, but in this case with respect to the expected SUT behavior (i.e., regarding the test evaluation scenarios) specified during the test evaluation phase. Traceability of the SUT requirements to the test model/code provides valuable support in realizing this criterion.

- *Test evaluation definition:* This criterion refers to the specification of the outputs expected from the SUT in response to the test case execution. Early work of Richardson, O'Malley, and Tittle (1998) already describes several approaches to specification-based test selection and extends them based on the concept of test oracle, faults, and failures. When a test engineer defines test scenarios in a certain formal notation, these scenarios can be used to determine how, when, and which tests will be evaluated.

1.3.4.2 Technology

The technology selected to implement the test evaluation specification enables an automatic or manual process, whereas the execution of the test evaluation occurs online or offline. Those options are elaborated next.

- *Automatic/Manual technology:* The *execution option* can be interpreted either from the perspective of the test evaluation definition or its execution. Regarding the specification of the test evaluation, when the expected SUT outputs are defined by hand, then it is a manual test specification process. In contrast, when they are derived automatically (e.g., from the behavioral model), then the test evaluation based on the test oracle occurs automatically. Typically, the expected reference signals/data are defined manually; however, they may be facilitated by parameterized test patterns application.

  The test assessment itself can be performed manually or automatically. Manual specification of the test evaluation is supported in Simulink® Verification and Validation™ (MathWorks® 2010), where predefined assertion blocks can be assigned to test signals defined in a Signal Builder block in Simulink. This practice supports verification of functional requirements during model simulation where the evaluation itself occurs automatically.

- *Online/Offline execution of the test evaluation:* The online (i.e., "on-the-fly") test evaluation happens already during the SUT execution. Online test evaluation enables the concept of test control and test reactiveness to be extended. Offline means that the test evaluation happens after the SUT execution, and so the verdicts are computed after analyzing the execution test logs.

  Watchdogs defined in Conrad and Hötzer (1998) enable online test evaluation. It is also possible when using TTCN-3. TPT means for online test assessment are limited and are used as watchdogs for extracting any necessary information for making test cases reactive (Lehmann and Krämer 2008). The offline evaluation is more sophisticated in TPT. It offers means for more complex evaluations, including operations such as comparisons with external reference data, limit-value monitoring, signal filters, and analyses of state sequences and time conditions.

1.4 Summary

This introductory chapter has extended the Model-Based Testing (MBT) *taxonomy* of previous work. Specifically, test dimensions have been discussed with pertinent aspects such as test goals, test scope, and test abstraction described in detail. Selected classes from the taxonomy have been illustrated, while all categories and options related to the test generation, test execution, and test evaluation have been discussed in detail with examples included where appropriate.

Such perspectives of MBT as cost, benefits, and limitations have not been addressed here. Instead, the chapters that follow provide a detailed discussion as they are in a better position to capture these aspects seeing how they strongly depend on the applied approaches and challenges that have to be resolved. As stated in Chapter 6, most published case studies illustrate that utilizing MBT reduces the overall cost of system and software development. A typical benefit achieves 20%–30% of cost reduction. This benefit may increase up to 90% as indicated by Clarke (1998) more than a decade ago, though that study only pertained to test generation efficiency in the telecommunication domain.

For additional research and practice in the field of MBT, the reader is referred to the surveys provided by Broy et al. (2005), Utting, Pretschner, and Legeard (2006), Zander and Schieferdecker (2009), Shafique and Labiche (2010), as well as every contribution found in this collection.

References

All4Tec, *Markov Test Logic—MaTeLo*, commercial Model-Based Testing tool, http://www.all4tec.net/ [12/01/10].

Beizer, B. (1995). *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. ISBN-10: 0471120944. John Wiley & Sons, Inc., Hoboken, NJ.

Broy, M., Jonsson, B., Katoen, J. -P., Leucker, M., and Pretschner, A. (Editors) (2005). *Model-Based Testing of Reactive Systems*, Editors: no. 3472. In *LNCS*, Springer-Verlag, Heidelberg, Germany.

BTC Embedded Systems AG, Embedded*Validator*, commercial verification tool, http://www.btc-es.de/ [12/01/10].

Carnegie Mellon University, Department of Electrical and Computer Engineering, *Hybrid System Verification Toolbox for MATLAB—CheckMate*, research tool for system verification, http://www.ece.cmu.edu/∼webk/checkmate/ [12/01/10].

Carter, J. M., Lin, L., and Poore, J. H. (2008). Automated Functional Testing of Simulink Control Models. In *Proceedings of the 1st Workshop on Model-based Testing in Practice—MoTip 2008*, Editors: Bauer, T., Eichler, H., Rennoch, A., ISBN: 978-3-8167-7624-6, Fraunhofer IRB Verlag, Berlin, Germany.

Clarke, J. M. (1998). Automated Test Generation from Behavioral Models. In the *Proceedings of the 11th Software Quality Week (QW'98)*, Software Research Inc., San Francisco, CA.

Conrad, M. (2004a). *A Systematic Approach to Testing Automotive Control Software*, Detroit, MI, SAE Technical Paper Series, 2004-21-0039.

Conrad, M. (2004b). *Modell-basierter Test eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien.* PhD thesis. Deutscher Universitätsverlag, Wiesbaden (D). (In German).

Conrad, M., Fey, I., and Sadeghipour, S. (2004). Systematic Model-Based Testing of Embedded Control Software—The MB^{3}T Approach. In *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems*, Edinburgh, United Kingdom.

Conrad, M., and Hötzer, D. (1998). Selective Integration of Formal Methods in the Development of Electronic Control Units. In *Proceedings of the ICFEM* 1998, 144-Electronic Edition, Brisbane Australia, ISBN: 0-8186-9198-0.

Dai, Z. R. (2006). *An Approach to Model-Driven Testing with UML 2.0, U2TP and TTCN-3.* PhD thesis, Technical University Berlin, ISBN: 978-3-8167-7237-8. Fraunhofer IRB Verlag.

Dempster, D., and Stuart, M. (2002). *Verification methodology manual, Techniques for Verifying HDL Designs*, ISBN: 0-9538-4822-1. Teamwork International, Great Britain, Biddles Ltd., Guildford and King's Lynn.

Din, G., and Engel, K. D. (2009). An Approach for Test Derivation from System Architecture Models Applied to Embedded Systems, In *Proceedings of the 2nd Workshop on Model-based Testing in Practice (MoTiP 2009)*, In Conjunction with the 5th European Conference on Model-Driven Architecture (ECMDA 2009), Enschede, The Netherlands, Editors: Bauer, T., Eichler, H., Rennoch, A., Wieczorek, S., CTIT Workshop Proceedings Series WP09-08, ISSN 0929-0672.

D-Mint Project (2008). Deployment of model-based technologies to industrial testing. http://d-mint.org/ [12/01/10].

dSPACE GmbH, *AutomationDesk*, commercial tool for testing, http://www.dspace.com/de/gmb/home/products/sw/expsoft/automdesk.cfm [12/01/2010a].

dSPACE GmbH, *MTest*, commercial MBT tool, http://www.dspaceinc.com/ww/en/inc/home/products/sw/expsoft/mtest.cfm [12/01/10b].

Dulz, W., and Fenhua, Z. (2003). MaTeLo—Statistical Usage Testing by Annotated Sequence Diagrams, Markov Chains and TTCN-3. In *Proceedings of the 3rd International Conference on Quality Software*, Page: 336, ISBN: 0-7695-2015-4. IEEE Computer Society Washington, DC.

ETSI (2007). European Standard. 201 873-1 V3.2.1 (2007-02): *The Testing and Test Control Notation Version 3; Part 1: TTCN-3 Core Language.* European Telecommunications Standards Institute, Sophia-Antipolis, France.

GeenSoft (2010a). *Safety Test Builder*, commercial Model-Based Testing tool, http://www.geensoft.com/en/article/safetytestbuilder/ [12/01/10].

GeenSoft (2010b). *Safety Checker Blockset*, commercial Model-Based Testing tool, http://www.geensoft.com/en/article/safetycheckerblockset_app/ [12/01/10].

Gips C., Wiesbrock, H. -W. (2007). Notation und Verfahren zur automatischen Überprüfung von temporalen Signalabhängigkeiten und -merkmalen für modellbasiert entwickelte Software. In *Proceedings of Model Based Engineering of Embedded Systems III*, Editors: Conrad, M., Giese, H., Rumpe, B., Schätz, B.: TU Braunschweig Report TUBS-SSE 2007-01. (In German).

Grochtmann, M., and Grimm, K. (1993). Classification Trees for Partition Testing. In *Software Testing, Verification & Reliability*, 3, 2, 63–82. Wiley, Hoboken, NJ.

Gutjahr, W. J. (1999). Partition Testing vs. Random Testing: The Influence of Uncertainty. In *IEEE Transactions on Software Engineering*, Volume 25, Issue 5, Pages: 661–674, ISSN: 0098–5589. IEEE Press Piscataway, NJ.

Hetzel, W. C. (1988). *The Complete Guide to Software Testing*. Second edition, ISBN: 0-89435-242-3. QED Information Services, Inc., Wellesley, MA.

International Software Testing Qualification Board (2006). *Standard glossary of terms used in Software Testing*. Version 1.2, produced by the Glossary Working Party, Editor: van Veenendaal E., The Netherlands.

IT Power Consultants, *MEval*, commercial tool for testing, http://www.itpower.de/30-0-Download-MEval-und-SimEx.html [12/01/10].

Kamga, J., Herrmann, J., and Joshi, P. Deliverable (2007). D-MINT automotive case study—Daimler, Deliverable 1.1, *Deployment of model-based technologies to industrial testing*, ITEA2 Project, Germany.

Kosmatov, N., Legeard, B., Peureux, F., and Utting, M. (2004). Boundary Coverage Criteria for Test Generation from Formal Models. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*. ISSN: 1071–9458, ISBN: 0-7695-2215-7, Pages: 139–150. IEEE Computer Society, Washington, DC.

Lamberg, K., Beine, M., Eschmann, M., Otterbach, R., Conrad, M., and Fey, I. (2004). Model-Based Testing of Embedded Automotive Software Using MTest. In *Proceedings of SAE World Congress*, Detroit, MI.

Lee, T., and Yannakakis, M. (1994). Testing Finite-State Machines: State Identification and Verification. In *IEEE Transactions on Computers*, Volume 43, Issue 3, Pages: 306–320, ISSN: 0018–9340. IEEE Computer Society, Washington, DC.

Lehmann, E. (then Bringmann, E.) (2003). *Time Partition Testing, Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen*, PhD thesis, Technical University Berlin. (In German).

Lehmann, E., and Krämer, A. (2008). Model-Based Testing of Automotive Systems. In *Proceedings of IEEE ICST 08*, Lillehammer, Norway.

Marre, B., and Arnould, A. (2000). Test Sequences Generation from LUSTRE Descriptions: GATEL. In *Proceedings of ASE of the 15th IEEE International Conference on Automated Software Engineering*, Pages: 229–237, ISBN: 0-7695-0710-7, Grenoble, France. IEEE Computer Society, Washington, DC.

MathWorks®, Inc., Real-Time Workshop®, http://www.mathworks.com/help/toolbox/rtw/ [12/01/10].

MathWorks®, Inc., *Simulink® Design Verifier*™, commercial Model-Based Testing tool, MathWorks®, Inc., Natick, MA, http://www.mathworks.com/products/sldesignverifier [12/01/10].

MathWorks®, Inc., *Simulink®*, MathWorks®, Inc., Natick, MA, http://www.mathworks.com/products/simulink/ [12/01/10].

MathWorks®, Inc., *Simulink® Verification and Validation*™, commercial model-based verification and validation tool, MathWorks®, Inc., Natick, MA, http://www.mathworks.com/products/simverification/ [12/01/10].

MathWorks®, Inc., *Stateflow®*, MathWorks®, Inc., Natick, MA, http://www.mathworks.com/products/stateflow/ [12/01/10].

MathWorks®, Inc., *SystemTest*™, commercial tool for testing, MathWorks®, Inc., Natick, MA, http://www.mathworks.com/products/systemtest/ [12/01/2010].

MATLAB Automated Testing Tool—MATT. (2008). The University of Montana, research Model-Based Testing prototype, http://www.sstc-online.org/Proceedings/2008/pdfs/JH1987.pdf [12/01/10].

Mosterman, P. J., Zander, J., Hamon, G., and Denckla, B. (2009). Towards Computational Hybrid System Semantics for Time-Based Block Diagrams. In *Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'09)*, Editors: A. Giua, C. Mahulea, M. Silva, and J. Zaytoon, pp. 376–385, Zaragoza, Spain, Plenary paper.

Mosterman, P. J., Zander, J., Hamon, G., and Denckla, B. (2011). A computational model of time for stiff hybrid systems applied to control synthesis, *Control Engineering Practice Journal (CEP)*, 19, Elsevier.

Myers, G. J. (1979). *The Art of Software Testing*. ISBN-10: 0471043281. John Wiley & Sons, Hoboken, NJ.

Neukirchen, H. W. (2004). *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*, PhD thesis, Georg-August-Universiät zu Göttingen.

OMG. (2003). MDA Guide V1.0.1. http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf [12/01/10 TODO].

OMG. (2003). UML 2.0 Superstructure Final Adopted Specification, http://www.omg.org/cgi-bin/doc?ptc/03-08-02.pdf [12/01/10].

OMG. (2005). UML 2.0 Testing Profile. Version 1.0 formal/05-07-07. Object Management Group.

PikeTec, *Time Partitioning Testing—TPT*, commercial Model-Based Testing tool, http://www.piketec.com/products/tpt.php [12/01/2010].

Pretschner, A. (2003). Compositional Generation of MC/DC Integration Test Suites. In *Proceedings TACoS'03*, Pages: 1–11. *Electronic Notes in Theoretical Computer Science* 6.

Pretschner, A. (2003a). Compositional Generation of MC/DC Integration Test Suites. In *Proceedings TACoS'03*, Pages: 1–11. *Electronic Notes in Theoretical Computer Science* 6. http://citeseer.ist.psu.edu/633586.html.

Pretschner, A. (2003b). *Zum modellbasierten funktionalen Test reaktiver Systeme*. PhD thesis. Technical University Munich. (In German).

Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., and Stauner, T. (2005). One Evaluation of Model-based Testing and Its Automation. In *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, Pages: 392–401, ISBN: 1-59593-963-2. ACM New York.

Pretschner, A., Slotosch, O., Aiglstorfer, E., and Kriebel, S. (2004). Model Based Testing for Real—The Inhouse Card Case Study. In *International Journal on Software Tools for Technology Transfer*. Volume 5, Pages: 140–157. Springer-Verlag, Heidelberg, Germany.

Rau, A. (2002). *Model-Based Development of Embedded Automotive Control Systems*, PhD thesis, University of Tübingen.

Reactive Systems, Inc., *Reactis Tester*, commercial Model-Based Testing tool, http://www.reactive-systems.com/tester.msp [12/01/10a].

Reactive Systems, Inc., *Reactis Validator*, commercial validation and verification tool, http://www.reactive-systems.com/reactis/doc/user/user009.html, http://www.reactive-systems.com/validator.msp [12/01/10b].

Richardson, D, O'Malley, O., and Tittle, C. (1998). Approaches to Specification-Based Testing. In *Proceedings of ACM SIGSOFT Software Engineering Notes*, Volume 14, Issue 8, Pages: 86–96, ISSN: 0163–5948. ACM, New York.

Schäuffele, J., and Zurawka, T. (2006). *Automotive Software Engineering*, ISBN: 3528110406. Vieweg.

Schieferdecker, I., and Großmann, J. (2007). Testing Embedded Control Systems with TTCN-3. In *Proceedings Software Technologies for Embedded and Ubiquitous Systems SEUS 2007*, Pages: 125–136, *LNCS* 4761, ISSN: 0302–9743, 1611–3349, ISBN: 978-3-540–75663-7 Santorini Island, Greece. Springer-Verlag, Berlin/Heidelberg.

Schieferdecker, I., Großmann, J., and Wendland, M.-F. (2011). Model-Based Testing: Trends. Encyclopedia of Software Engineering DOI: 10.1081/E-ESE-120044686, Taylor & Francis.

Schieferdecker, I., and Hoffmann, A. (2011). Model-Based Testing. Encyclopedia of Software Engineering DOI: 10.1081/E-ESE-120044686, Taylor & Francis.

Schieferdecker, I., Rennoch, A., and Vouffo-Feudjio, A. (2011). Model-Based Testing: Approaches and Notations. Encyclopedia of Software Engineering DOI: 10.1081/E-ESE-120044686, Taylor & Francis.

Shafique, M., and Labiche, Y. (2010). *A Systematic Review of Model Based Testing Tool Support*, Carleton University, Technical Report, SCE-10-04, http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-10-04.pdf [03/22/11].

Sims S., and DuVarney D. C. (2007). Experience Report: The Reactis Validation Tool. In *Proceedings of the ICFP '07 Conference*, Volume 42, Issue 9, Pages: 137–140, ISSN: 0362–1340. ACM, New York.

Software Quality Research Laboratory, *Java Usage Model Builder Library—JUMBL*, research Model-Based Testing prototype, http://www.cs.utk.edu/sqrl/esp/jumbl.html [12/01/10].

SynaptiCAD, *Waveformer Lite 9.9 Test-Bench* with Reactive Test Bench, commercial tool for testing, http://www.actel.com/documents/reactive_tb_tutorial.pdf [12/01/10].

Utting, M. (2005). Model-Based Testing. In *Proceedings of the Workshop on Verified Software: Theory, Tools, and Experiments VSTTE 2005*.

Utting, M., and Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach.* ISBN-13: 9780123725011. Elsevier Science & Technology Books.

Utting, M., Pretschner, A., and Legeard, B. (2006). *A taxonomy of model-based testing*, ISSN: 1170-487X, The University of Waikato, New Zealand.

Weyuker, E. (1988). The Evaluation of Program-Based Software Test Data Adequacy Criteria. In *Communications of the ACM*, Volume 31, Issue 6, Pages: 668–675, ISSN: 0001-0782. ACM, New York, NY.

Wieczorek, S., Kozyura, V., Roth, A., Leuschel, M., Bendisposto, J., Plagge, D., and Schieferdecker, I. (2009). Applying Model Checking to Generate Model-based Integration Tests from Choreography Models. *21st IFIP Int. Conference on Testing of Communicating Systems (TESTCOM)*, Eindhoven, The Netherlands, ISBN 978-3-642-05030-5.

Wiesbrock, H. -W., Conrad, M., and Fey, I. (2002). Pohlheim: Ein neues automatisiertes Auswerteverfahren für Regressions und Back-to-Back-Tests eingebetteter Regelsysteme. In *Softwaretechnik-Trends*, Volume 22, Issue 3, Pages: 22–27. (In German).

Zander, J., Dai, Z. R., Schieferdecker, I., and Din, G. (2005). From U2TP Models to Executable Tests with TTCN-3—An Approach to Model Driven Testing. In *Proceedings of the IFIP 17th Intern. Conf. on Testing Communicating Systems (TestCom 2005)*, ISBN: 3-540–26054-4, Springer-Verlag, Heidelberg, Germany.

Zander, J., Mosterman, P. J., Hamon, G., and Denckla, B. (2011). On the Structure of Time in Computational Semantics of a Variable-Step Solver for Hybrid Behavior Analysis, 18th World Congress of the International Federation of Automatic Control (IFAC), Milano, Italy.

Zander, J., and Schieferdecker, I. (2009). Model-Based Testing of Embedded Systems Exemplified for the Automotive Domain, Chapter in *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, Editors: Gomes, L., Fernandes, J. M., DOI: 10.4018/978-1-60566-750-8.ch015. Idea Group Inc. (IGI), Hershey, PA, ISBN 1605667501, 9781605667508, pp. 377–412.

Zander-Nowicka, J. (2009). *Model-Based Testing of Embedded Systems in the Automotive Domain*, PhD Thesis, Technical University Berlin, ISBN: 978-3-8167-7974-2. Fraunhofer IRB Verlag, Germany. http://opus.kobv.de/tuberlin/volltexte/2009/2186/pdf/zandernowicka_justyna.pdf.