Model-based Testing of Embedded Systems exemplified for the Automotive Domain

Justyna Zander-Nowicka, Ina Schieferdecker
MOTION, Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
Technische Universität Berlin, FIV, Straße des 17. Juni 135, 10623 Berlin, Germany
{justyna.zander-nowicka, ina.schieferdecker}@fokus.fraunhofer.de

Keywords:
Software Quality, System Quality, Testing, Modeling Languages, Software Design, Testing Issues

Abstract:
The purpose of this chapter is to introduce the testing methods categorized as model-based testing for embedded systems addressing selected problems in the automotive domain.
The main contribution refers to functional black-box testing based on the system and test models. It is contrasted with the test methods currently applied in the industry that form dedicated solutions, usually specialized in a concrete testing context. Model-based test approaches are reviewed and categorized. Weak points are identified and a novel test method realized in MATLAB®/Simulink®/Stateflow® environment is proposed. It is called Model-in-the-Loop for Embedded System Test (MiLEST).
The developed signal-feature – oriented paradigm allows the abstract description of signals and their properties. It addresses the problem of missing reference signal flows and allows for a systematic and automatic test data selection. Processing of both discrete and continuous signals is possible, so that the hybrid behavior of embedded systems can be handled.

1. INTRODUCTION

1.1 Embedded System Context

An *embedded system* [BBK98, LV04] is a system built for dedicated control functions. *Embedded software* [LV04] is the software running on an embedded system. Embedded systems have become increasingly sophisticated and their software content has grown rapidly for the last years. Applications now consist of hundreds of thousands or even more lines of code. The requirements that must be fulfilled while developing embedded software are complex in comparison to the standard software. Embedded systems are often produced in large volumes and the software is difficult to be updated once the product is deployed. Embedded systems interact with real-life environment. Hybrid aspects are often expressed via mathematical formulas. In terms of software development, increased complexity of products, shortened development cycles and higher customer expectations of quality implicate the extreme importance and automation need for software testing. Software development activities in every phase are error prone, so the process of defects detection plays a crucial role. The cost of finding and fixing defects grows exponentially in the development cycle. The software testing problem is complex because of the large number of possible scenarios. The typical testing process is a human-intensive activity and, as such, it is usually unproductive and often inadequately done. Nowadays, testing is one of the weakest points of current development practices. According to the study in [Enc03] 50% of embedded systems development projects are months behind schedule and only 44% of designs meet 20% of functionality and performance expectations. This happens despite the fact that approximately 50% of total development effort is spent on testing [Enc03, Hel+05]. The impact of research into test methodologies that reduce this

effort is therefore very high and strongly desirable [ART05, Hel+05].
Although, a number of valuable efforts in the context of testing already exist, there is still a lot of space to improve the situation. This applies in particular, to the automation potential of the test methods. Also a systematic, appropriately structured, repeatable and consistent test specification is still an aim to be reached. Furthermore, both abstract and concrete views should be supported so as to improve the readability, on the one hand and assure the executability of the resulting test, on the other hand. In the context of this work, further factors become to be crucial. The testing method should address all aspects of a tested system – whether a mix of discrete and continuous signals, time constrained functionality or a complex configuration is considered. In order to establish a controlled and stable testing process with respect to time, budget and software quality, the software testing process must be modeled, measured and analyzed [LV04]. Moreover, the existence of executable system models opens the potentials for model-based testing (MBT). Nowadays MBT is widely used, however with slightly different meanings. In the automotive industry MBT is applied to describe all testing activities in the context of model-based development (MBD) [CFS04, LK08]. It relates to a process of test generation based on a *system under test* (SUT) model by application of a number of sophisticated methods being the automation of black-box test design [UL06]. Surveys on different MBT approaches are given in [BJK+05, Utt05, UL06, UPL06, D-Mint08].
In this chapter additionally, requirements-based testing is considered. Furthermore, a graphical form of a test design will increase the readability. The provided test patterns will considerably reduce the test specifications effort and support their reusability. Then, an abstract and common way of describing both discrete and continuous signals will result in automated test signals generation and their evaluation.

1.2 Automotive Domain

Studies show that the strongest impact of embedded systems on the market has to be expected in the automotive industry. The share of innovative electronics and software in the total value of an automobile is currently estimated to be at least 25%, with an expected increase to 40% in 2010 and up to 50% for the time after 2010 [Hel+05, SZ06]. Prominent examples of such electronic systems are safety facilities, advanced driver assistance systems (ADAS) or adaptive cruise control (ACC). These functionalities are realized by software within electronic control units (ECUs). A modern car has up to 80 ECUs [BBK98, SZ06].
Furthermore, the complexity of car software dramatically increases as it implements formerly mechanically or electronically integrated functions. Yet, the functions are distributed over several ECUs interacting with each other. Studies by [MW04, SZ06, BKP+07, KHJ07] predict that most innovations in a car will come from its electronic embedded systems and their inherent software. Nowadays, *embedded software* [Con04b] makes up 85% [Hel+05] of the value of the entire system.
At the same time, it is demanded to shorten time-to-market for a car by making its software components reliable and safe. Additionally, studies in [Dess01] show that the cost of recalling a car model with a safety critical failure can be more than the cost of thorough testing/verification. Under these circumstances the introduction of quality assurance techniques in the automotive domain becomes obvious and will be followed within this work.

The remainder of this chapter is the following. In Section 2, testing requirements for embedded systems are considered. The test development process, test dimensions and model-based testing (MBT) are discussed. Section 3 categorizes the MBT approaches based on a taxonomy. In Section 4, the state of the art is reviewed so as to introduce the proposed test method in Section 5. Section 6 gives a simple case study and Section 7 enlightens the future trends. The chapter is completed with conclusions.

2. TESTING OF EMBEDDED SYSTEMS

2.1 Background

Testing, an analytic means for assessing the quality of software [Wal01, UL06], is one of the most

important phases during the software development process with regard to quality assurance. It *„can never show the absence of failures“* [Dij72], but it aims at increasing the confidence that a system meets its specified behavior. Testing is an activity performed for improving the product quality by identifying defects and problems. It cannot be undertaken in isolation. Instead, in order to be in any way successful and efficient, it must be embedded in adequate software development process and have interfaces to the respective sub-processes.
An *error* is a human action that produces an incorrect result as defined in [ISTQB06]. *Fault* (also called defect) is a flaw in a component or system that can cause the component or system to fail to perform its required function, e.g., an incorrect statement or data definition. A fault, if encountered during execution, may cause a *failure* of the component or system [ISTQB06]. Failures represent the deviation of the component or system from its expected delivery, service or result [ISTQB06].

2.2 Test Development Process

The fundamental test process according to [BS98, SL05, ISTQB06] comprises (1) planning, (2) specification, (3) execution, (4) recording (i.e., documenting the results), (5) checking for completion, and test closure activities (e.g., rating the final results).
Test planning includes the planning of resources and the laying down of a test strategy: defining the test methods and the coverage criteria to be achieved, the test completion criteria, structuring and prioritizing the tests, and selecting the tool support as well as configuration of the test environment [SL05]. In the test specification the corresponding test cases are specified using the test methods defined by the test plan [SL05]. Test execution means the execution of test cases and test scenarios. Test records serve to make the test execution understandable for people not directly involved (e.g., customer) and prove afterwards, whether and how the planned test strategy was in actual fact executed. Finally, during the test closure step data is collected from completed test activities to consolidate experience, testware, facts, and numbers. The test process is evaluated and a report is provided [ISTQB06].

In addition, [Dai06] considers a process of test development. The test development process, related to steps 2 – 4 of the fundamental test process, can be divided into six phases, which are usually consecutive, but may be iterated: test requirements, test design, test specification, test implementation, test execution, and test evaluation.

The test process aimed at in this work covers with the fundamental one, although only steps 2 – 4 are addressed in further considerations. Compared to [Dai06] the test development process is modified and shortened. It is motivated by the different nature of the considered SUTs. Within traditional software and test development, phases are clearly separated [CH98]. For automotive systems a closer integration of the specification and implementation phases occurs. Hence, after defining the test requirements, the test design phase encompasses the preparation of a test harness. The detailed test specification and test implementation are done within one step as the applied modeling language is executable. Up to this point, the test development process supported in the solution proposal given in Section 5, is very similar to the one defined by [Leh03]. Further on, test execution and test evaluation are performed simultaneously.

2.3 Hybrid Embedded Systems Test and its Dimensions

Tests can be classified in different levels, depending on the characteristics of the SUT and the test system. [Neu04] aims at testing the communication systems and categorizes testing in the dimensions of test goals, test scope, and test distribution. [Dai06] replaces the test distribution by a dimension describing the different test development phases, since she is testing both local and distributed systems. In this chapter embedded systems are regarded as SUTs, thus, the test dimensions are modified as shown in Figure 1.

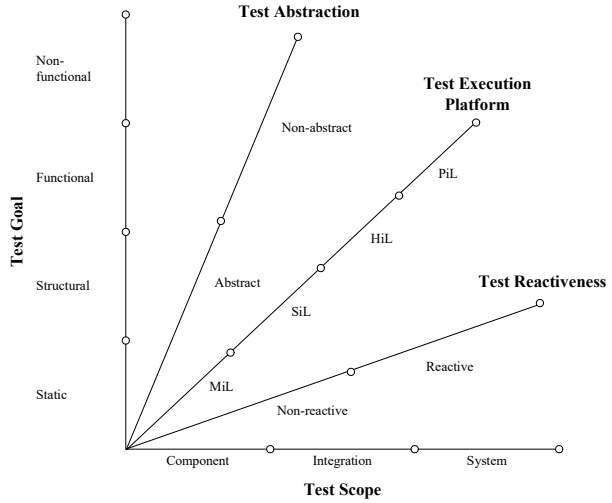

Figure 1: The Five Test Dimensions for Embedded System Test.

Test Goal: During the software development systems are tested with different purposes (i.e., goals). They can be categorized into static testing, also called *review*, and dynamic testing, whereas the latter is distinguished between structural, functional, and non-functional testing. In the automotive, after the *review* phase, the test goal is usually to check the functional behavior of the system. Non-functional tests appear in later development stages.

- *Static Test:* Testing is often defined as the process of finding the errors, failures, and faults. In a program, they can be revealed without execution by just examining its source code [ISTQB06]. Similarly, other development artefacts can be reviewed (e.g., requirements, models or test specification itself). This process is called *static testing*. *Dynamic testing* in contrast, bases on execution.

- *Structural Test:* Structural tests cover the structure of the SUT during test execution (e.g., control or data flow). To achieve this, the internal structure of the system (e.g., code or model) needs to be known. Therefore, structural tests are also called white-box or glass-box tests [Mye79, ISTQB06].

- *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 [Bei95]. In this category functional safety tests are also included. Their purpose is to determine the safety of a software product. They require a systematic, planned, executed, and documented procedure. At present, safety tests are only a small part of software testing in the automotive area. By introduction of safety standards such as IEC 61508 [IEC05] and ISO 26262 [ISO_FS] the meaning of software safety tests will, however, increase considerably within the next few years.

- *Non-functional Test*: Similar to functional tests, non-functional tests are performed against requirements specification of the system. In contrast to pure functional testing, non-functional testing aims at the assessment of non-functional, such as reliability, load or performance requirements. Non-functional tests are usually black-box tests. Nevertheless, for retrieving certain information, e.g., internal clock, internal access during test execution is required.

The focus of this chapter is put on *functional tests*. However some timing and safety aspects are included as well.

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 is observed. However, the specified test cases must be executable at the same time. The *non-abstract* tests are supported by a number of tool providers and they do not scale for larger industrial projects [LK08]. Hence, the abstraction level should not affect the test execution in a negative way.

This chapter develops a conceptual framework for *abstract* test specification; however, simultaneously an *executable* technical framework for a selected platform is built.

Test Execution Platform: 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.

- *Model-in-the-Loop (MiL):* The first integration level, MiL, is based on the model of the system itself. In this platform the SUT is a functional model or implementation model that is tested in an open-loop (i.e., without any plant model in the first place) or closed-loop test with a plant model (i.e., without any physical hardware) [KHJ07, SZ06, LK08]. The test purpose is basically functional testing in early development phases in simulation environments such as MATLAB®/Simulink®/ Stateflow® (ML/SL/SF).

- *Software-in-the-Loop (SiL):* During SiL the SUT is software tested in a closed or open-loop. The software components under test are usually implemented in C and are either hand-written or generated by code generators based on implementation models. The test purpose in SiL is mainly functional testing [KHJ07]. 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 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 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 which allows debugging during tests in a cheap and manageable way [LK08]. Therefore, the effort spent by PiL testing is worthwhile in almost all cases.

- *Hardware-in-the-Loop (HiL)*: When testing the embedded system on HiL level the software runs on the final ECU. However the environment around the ECU is still a simulated one. ECU and environment interact via the digital and analog electrical connectors of the ECU. The objective of testing on HiL level is to reveal faults in the low-level services of the ECU and in the I/O services [SZ06]. Additionally, acceptance tests of components delivered by the supplier are executed on the HiL level because the component itself is the integrated ECU [KHJ07]. 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.

- *Car:* Finally, the last integration level is obviously the car itself, as already mentioned. The final ECU runs in the real car which can either be a sample or a car from the production line. However, these tests, as performed only in late development phases, are expensive and do not allow configuration parameters to be varied arbitrarily [LK08]. Hardware faults are difficult to trigger and the reaction of the SUT is often difficult to observe because internal signals are no longer accessible [KHJ07]. For these reasons, the number of in-car tests decreases while model-based testing gains more attention.

This chapter mainly the system design level so as to start testing as early as possible in the development cycle. Thus, the *MiL* platform is researched in detail. The other platforms are not excluded from the methodological viewpoint. However, the portability between different execution platforms is beyond the scope of this work.

Test Reactiveness: A concept of *test reactiveness* emerges when test cases are dependent on the system behavior. That is, the execution of a test case depends on what the system under test is doing while being tested. In this sense the system under test and the test driver run in a '*closed loop*'. In the following, before the *test reactiveness* will be elaborated in detail, the definition of *open-* and *closed-loop system configuration* will be explicitly distinguished:

- *Open-loop System Configuration*: When testing a component in a so-called open-loop the test object is tested directly without any environment or environmental model. This kind of testing is reasonable if the behavior of the test object is described based on the

interaction directly at its interfaces (I/O ports). This configuration is applicable for SW modules and implementation sub-models, as well as for control systems with discrete I/O.

– *Closed-loop System Configuration*: For feedback control systems and for complex control systems it is necessary to integrate the SUT with a plant model so as to perform closed-loop tests. In early phases where the interaction between SUT and plant model is implemented in software (i.e., without digital or analog I/O, buses etc.) the plant model does not have to ensure real-time constraints. However, when the HiL systems are considered and the communication between the SUT and the plant model is implemented via data buses, the plant model may include real hardware components (i.e., sensors and actuators). This applies especially when the physics of a system is very crucial for the functionality or when it is too complex to be described in a model.

– *Test Reactiveness*: 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. With this practice, the execution of reactive test cases varies depending on the SUT behavior. The test reactiveness as such gives the test system a possibility to immediately react to the incoming behavior by modifying the test according to the predefined deterministic criteria. The precondition for achieving the test reactiveness is an online monitoring of the SUT, though. The advantages can be obtained in a number of test specification steps (e.g., an automatic sequencing of test cases, online prioritizing of the test cases).

For example, assume that the adaptive cruise control (ACC) activation should be tested. It is possible to start the ACC only when a certain velocity level has been reached. Hence, the precondition for a test case is the increase of the velocity from 0 up to the point when the ACC may be activated. If the test system is able to detect this point automatically, the ACC may be tested immediately.

A discussion about possible risks and open questions around reactive tests can be found in [Leh03].

In this work, both *open-* and *closed-loop system configurations* as well as *reactive* and *non-reactive tests* will be regarded.

Test Scope: Finally, the test scope has to be considered. Test scopes describe the granularity of the SUT. Due to the composition of the system, tests at different scopes may reveal different failures [ISTQB06, D-Mint08, Wey88]. Therefore, they are usually performed in the following order:

– Component: At the scope of component testing, the smallest testable component (e.g., a class in an object-oriented implementation or a single ECU) is tested in isolation.

– Integration: The scope of integration test is to combine components with each other and test those not yet as a whole system but as a subsystem (i.e., ACC system composed of a speed controller, a distance controller, switches and several processing units). It exposes defects in the interfaces and in the interactions between integrated components or systems [ISTQB06].

– System: In a system test, the complete system (i.e., a vehicle) consisting of subsystems is tested. A complex embedded system is usually distributed; the single subsystems are connected via buses using different data types and interfaces through which the system can be accessed for testing [Het98].

This chapter encompasses the *component level test*, including both single component and component in-the-loop test, as well as the *integration level test*.

3. MODEL-BASED TESTING TAXONOMY

Model-based testing (MBT) relates to a process of test generation from an SUT model by application of a number of sophisticated methods. MBT is the

automation of black-box test design [UL06]. Several authors [BLL+04, Utt05, CGN+05, FTW06, KHJ07, Tre08] define MBT as testing in which test cases are derived in whole or in part from a model that describes some aspects of the SUT based on selected criteria in different contexts. [Dai06] denotes MBT into model-driven testing (MDT) since she proposes the approach in the context of Model Driven Architecture (MDA). [UPL06] add that MBT inherits the complexity of the domain or, more particularly, of the related domain models.

MBT allows tests to be linked directly to the SUT requirements, makes readability, understandability and maintainability of tests easier. It helps to ensure a repeatable and scientific basis for testing and it may give good coverage of all the behaviors of the SUT [Utt05]. Finally, it is a way to reduce the efforts and cost for testing [PPW+05].

The term *MBT* is widely used today with slightly different meanings. Surveys on different MBT approaches are given in [BJK+05, Utt05, UL06, UPL06, D-Mint08]. In the automotive industry MBT is used to describe all testing activities in the context of MBD [CFS04, LK08]. [Rau02, LBE+04, Con04a, Con04b] define MBT as a test process that encompasses a combination of different test methods which utilize the executable model as a source of information. Thus, the automotive viewpoint on MBT is rather process-oriented. A single testing technique is not enough to provide an expected level of test coverage. Hence, different test methods should be combined to complement each other relating to all the specified test dimensions (e.g., functional and structural testing techniques should be combined). If sufficient test coverage has been achieved on model level, the test cases can be reused for testing the control software generated from the model and the control unit within the framework of back-to-back tests [WCF02]. With this practice, the functional equivalence between executable model, code and ECUs can be verified and validated [CFS04].

For the purpose of the approach presented in Section 5, the following understanding of MBT is used:

Model-based testing is testing in which the *entire test specification* is derived in whole or in part from both *the system requirements and a model* that describe selected functional aspects of the SUT. In this context, the term *entire test specification* covers the *abstract test scenarios* substantiated with the concrete sets of test data and the expected SUT outputs. It is organized in a set of test cases.

Further on, the resulting test specification is *executed* together with the SUT model so as to provide the test results. In [Con04a, CFS04] no additional models are being created for test purposes, but the already existent functional system models are utilized for the test. In the test approach proposed in this chapter the system models are exploited too. In addition, however, a *test specification model* (also called *test case specification*, *test model* or *test design* in the literature [Pre03b, ZDS+05, Dai06]) is created semi-automatically. Concrete test data variants are derived automatically from it.

Moreover, since the MBT approaches have to be integrated into the existing development processes and combined with the existing methods and tools, a good practice is to select a common framework for both system and test definition. By that, MBD and MBT are supported using the same environment.

3.1 Model-based Testing Taxonomy

In [UPL06, UL06] a comprehensive *taxonomy* for MBT identifying its three general *classes*: model, test generation, and test execution is provided. Each of the classes is divided into further *categories*. The model-related ones are subject, independence, characteristics, and paradigm. Further on, the test generation is split into test selection criteria and technology, whereas the test execution partitions into execution options.

In the following work, the *taxonomy* is enriched with an additional *class*, called test evaluation. The test evaluation means comparing the actual SUT outputs with the expected SUT behavior based on a test oracle. Test oracle enables a decision to be made as to whether the actual SUT outputs are correct. It is, apart from the data, a crucial part of a

test case. The test evaluation is divided into two *categories*: specification and technology.
Furthermore, in this chapter only one selected class of the system model is investigated. For clarification purposes, its short description based on the options available in the taxonomy of [UPL06, UL06] will be given. The subject is the model (e.g., SL/SF model) that specifies the intended behavior of the SUT and its environment, often connected via a feedback loop. Regarding the independence level this model can be generally used for both test case and code generation. Indicating the model characteristics, it provides deterministic hybrid behavior constrained by timed events, including continuous functions and various data types. Finally, the modeling paradigm combines a history-based, functional data flow paradigm (e.g., the SL function blocks) with a transition-based notation (e.g., SF charts).

The overview of the resulting, slightly modified and extended MBT *taxonomy* is illustrated in Figure 2. The modification results from the focus of this chapter, which is put on embedded systems. All the categories are split into further instances which influence each other within a given category or between them. The notion of 'A/B/C' at the leaves indicates mutually exclusive options, while the straight lines link further instantiations of a given dimension without exclusion. It is a good practice since, for example, applying more than one test selection criterion and by that, more generation technologies can provide a better test coverage, eventually.

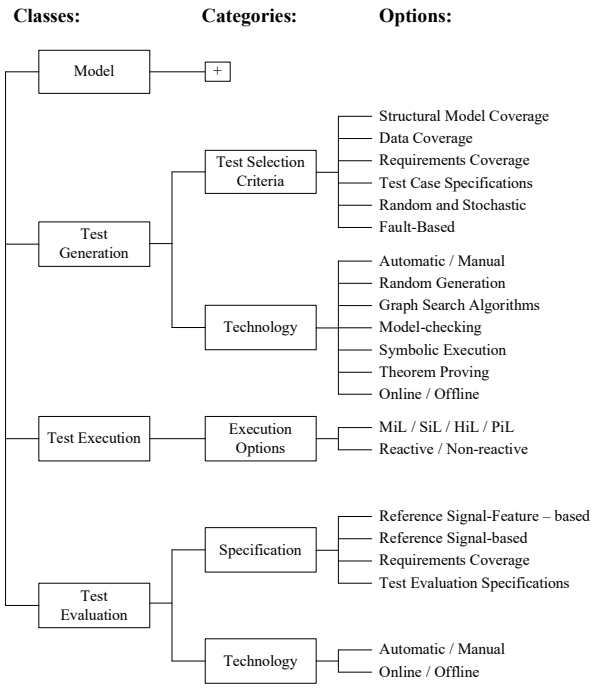

Figure 2: Overview of the Taxonomy for Model-based Testing.

In the next three sections it is referred to the *classes* of the MBT taxonomy and the particular *categories* and *options* are explained in depth. The descriptions of the most important *options* followed in this chapter contain examples of their realization, respectively.

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 leads to the creation of test cases. Depending on the test selection criteria and generation technology a number of approaches exist. They are reviewed below.

Test selection criteria: Test selection criteria define the facilities that are used to control the generation of tests. They help to specify the tests and do not depend on the SUT code [UL06]. In the following, the most commonly-used criteria are investigated. Different test methods should be combined to complement each other so as to achieve the best test coverage. Hence, there is no

best suitable criterion for generating the test specification.

- *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 control-flow through code.
  In [Pre03] it is 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) [STB] or Reactis Tester [ReactT, SD07] generate test sequences covering a set of SF test objectives (e.g., transitions, states, junctions, actions, MC/DC coverage) and a set of SL test objectives (e.g., boolean flow, look-up tables, conditional subsystems coverage).

- *Data coverage criteria:* The idea is to split the data range into equivalence classes and select one representative from each class. This partitioning is usually complemented by the boundary value analysis [KLP+04], where the critical limits of the data ranges or boundaries determined by constraints are additionally selected.
  An example is the MATLAB Automated Testing Tool (MATT) [MATT] enabling black-box testing of SL models and code generated from it by Real-Time Workshop®. It generally enables the creation of custom test data for model simulations by setting their types for each input. Further on, 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) [CTE] implementing the Classification Tree Method (CTM) [GG93, Con04a]. The SUT inputs form the classifications in the roots of the tree. Then, 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 line in the table specifies a test case. CTE/ES provides a way of finding test cases systematically. It breaks the test scenario design process down into steps. Additionally, the test scenario is visualized in a graphical user interface (GUI).

- *Requirements coverage criteria:* These aim to cover all the informal SUT requirements. Traceability of the SUT requirements to the system or test model/code can support the realization of this criterion. It is targeted by almost every test approach.

- *Test case specifications:* When the test engineer defines a test case specification in some formal notation, these can be used to determine which tests will be generated. It is explicitly decided 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 [UPL06]. Notations commonly used for test objectives include Finite State Machines (FSMs), UML Testing Profile (UTP) [UTP], regular expressions, temporal logic formulas, constraints, and Markov chains (for expressing intended usage patterns).
  A prominent example of applying this criterion is described in [Dai06], where the test case specifications are retrieved from UML® models and transformed into executable tests in Testing and Test Control Notation, version 3 (TTCN-3) [ETSI07] by using Model Driven Architecture (MDA) [MDA] methods [ZDS+05]. The work of [Pre03, Pre04] is also based on this criterion (see symbolic execution in the next paragraph).

- *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 [CLP08]. 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.

Exemplifying, Markov Test Logic (MaTeLo) [MaTL] can generate test suites according to several algorithms. Each of them optimizes the test effort according to the 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 [DF03].
Another instance, Java Usage Model Builder Library (JUMBL) [JUMB] can generate test cases either as a collection of test cases which cover the model with the minimum cost or by random sampling with replacement, or in order by probability, or by interleaving the events of other test cases. There is also an interactive test case editor for creating test cases by hand.

- *Fault-based criteria:* These rely on knowledge of typically occurring faults, often designed in the form of a fault model (cf. [Aic05]).

Test generation technology: One of the most appealing characteristics of model-based testing is its potential for automation. The automated generation of test cases usually necessitates the existence of kind of test case specifications [UPL06].

- *Automatic/Manual technology:* Automatic test generation refers to the situation when the test cases are generated automatically from the information source based on the given criteria (cf. [APJ+03]). Manual test generation refers to the situation when the test cases are produced by hand.

- *Random generation:* Random generation of tests is done by sampling the input space of a system. It is easy to implement, but it takes a long time to reach a certain satisfying level of model coverage as [Gut99] reports.

- *Graph search algorithms:* Dedicated graph search algorithms include node or arc coverage algorithms such as the Chinese Postman algorithm, which 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 [LY94], additionally based on 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 [AB02]. It can yield counter examples when a property is not satisfied. If no counter example is found, then the property is proven and the situation has never been reached. Such a mechanism is implemented in CheckMate [ChM, SRK+00], Safety Checker Blockset (SCB) [SCB] or in Embedded*Validator* [EmbV].
The general idea of test case generation with model checkers is to first formulate test case specifications as reachability properties, for instance, “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. 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 [MA00]. 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 done randomly by respecting these constraints.
In [Pre03b] an approach to test case generation with symbolic execution on the backgrounds of Constraint Logic Programming (CLP), initially transformed from the AutoFocus models [AuFo], is provided. [Pre03b, Pre04] concludes that test case generation for both functional and structural test case specifications limits to finding states in the model’s state space. Then, the aim of symbolic execution of a model is then to find a trace representing a test case that leads to the specified state.

- *Theorem proving:* Usually theorem provers are used 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.
The technique applied in Simulink® Design Verifier™ (SL DV) [SLDV] uses mathematical procedures to search through the possible execution paths of the model so as to find test cases and counter examples.

– *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 used for implementing the reactive tests too.

Offline testing means that test cases are generated 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, levels of abstractions or in different environments. Finally, if the test generation process is slower than test execution, then there are obvious advantages to doing the test generation phase only once.

3.2 Test Execution

The test execution options in the context of this chapter have been already described. Hence, in the following only reactive testing and the related work on the reactive/non-reactive option is reviewed.

Execution options: Execution options refer to the execution of a test.

– *Reactive/Non-reactive 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. Then the execution of reactive test cases varies depending on the SUT behavior, in contrast to the non-reactive test execution, where the SUT does not influence the test at all.
Reactive tests can be implemented within AutomationDesk [AutD]. Such tests react to changes in model variables within one simulation step. The scripts run on the processor of the HiL system in real time, synchronously to the model.
The Reactive Test Bench [WTB] 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.
[DS02] conclude that a dynamic test generator and checker are more effective in creating reactive test sequences. They are also more efficient because errors can be detected as they happen. Resigning from the reactive testing methods, a simulation may run for a few hours only to find out during the post-process checking that an error occurred a few minutes after the simulation start.
In [JJR05], in addition to checking the conformance of the implementation under test (IUT), the goal of the test case is to guide the parallel execution towards satisfaction of a test purpose. Due to that feature, the test execution can be seen as a game between two programs: the test case and the IUT. The test case wins if it succeeds in realizing one of the scenarios specified by the test purpose; the IUT wins if the execution cannot realize any test objective. The game may be played offline or online [JJR05].

3.3 Test Evaluation

The test evaluation, also called the test assessment, is the process that exploits the test oracle. It is a mechanism for analyzing the SUT output and deciding about the test result. As already discussed before, 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 specialized knowledge. The test evaluation is treated explicitly in this chapter since herewith a new concept for the test evaluation is proposed.

Specification: Specification of the test assessment algorithms may be based on different foundations that cover some criteria. It usually forms a kind of model or a set of ordered reference signals/data assigned to specific scenarios. Considering continuous signals the division into reference-based

and reference signal-feature – based evaluation becomes particularly important:

- *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 the MTest [MTest, Con04a] or SystemTest™ [STest]. 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 [MEval, WCF02].
- *Reference signal-feature – based specification:* Test evaluation based on reference signal feature assesses the SUT behavior comparing the SUT outcomes partitioned into features with the previously specified reference values for those features.
  Such an approach to test evaluation is supported in the Time Partitioning Test (TPT) [TPT, Leh03, LKK+06]. It is based on the script language Python extended with some syntactic test evaluation functions. By that, the test assessment can be flexibly designed and allows for dedicated complex algorithms and filters to be applied to the recorded test signals. A library containing complex evaluation functions is available.

- *Requirements coverage criteria:* Similar to the case of test data generation, they aim to cover all the informal SUT requirements, but this time with respect to the expected SUT behavior (i.e., regarding the test evaluation scenarios) specified there. Traceability of the SUT requirements to the test model/code can support the realization of this criterion.
- *Test evaluation specifications:* This criterion refers to the specification of the outputs expected from the SUT after the test case execution. Already authors of [ROT98] describe several approaches to specification-based test selection and build them up on the concept of test oracle, faults and failures. When the test engineer defines test scenarios in some formal notation, these can be used to determine how, when and which tests will be evaluated.

Technology: The technology of the test assessment specification enable an automatic or manual process, whereas the execution of the test evaluation occurs online or offline.

- *Automatic/Manual technology:* The *option* can be understood twofold, 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. Usually, the expected reference signals/data are defined manually; however, they may be facilitated by parameterized test patterns application.
  The activity of test assessment itself can be done manually or automatically.
  Manual specification of the test evaluation means is supported in Simulink® Verification and Validation™ (SL VV) [SLVV], where the predefined assertion blocks can be assigned to the test signals defined in the Signal Builder block in SL. With this practice, functional requirements can be verified during model simulation. The evaluation itself then occurs automatically.
  The tests developed in SystemTest exercise MATLAB (ML) algorithms and SL models. The tool includes predefined test elements to build and maintain standard test routines. Test cases, including test assessment, can be specified manually at a low abstraction level. A set of automatic evaluation means exists and the comparison of obtained signals with the reference ones is done 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 the opposite. Hence, the test evaluation happens after the SUT execution.

Watchdogs defined in [CH98] enable online test evaluation. It is also possible when using TTCN-3. TPT [Leh03] means for online test assessment are limited and are used as watchdogs for extracting any necessary information for making test cases reactive. The offline evaluation is more sophisticated in TPT.

4. ANALYSIS OF THE EXISTING APPROACHES

Established test tools from, e.g., dSPACE GmbH [dSP], Vector Informatik GmbH [VecI], MBTech Group [MBG] etc. are highly specialized for the automotive domain and usually come together with a test scripting approach which is directly integrated to the respective test device. All these test definitions pertain to a particular test device and by that not portable to other platforms and not exchangeable.

Recently, the application of model-based specifications in development enables more effective and automated process reaching a higher level of abstraction.
Thereby, model-based testing and platform-independent approaches have been developed such as CTE/ES [Con04a], MTest, and TPT [Leh03]. As already mentioned CTE/ES supports the CTM with partition tests according to structural or data-oriented differences of the system to be tested. It also enables the definition of sequences of test steps in combination with the signal flows and their changes along the test. Because of its ease of use, graphical presentation of the test structure and the ability to generate all possible combination of tests, it is widely used in the automotive domain. Integrated with the MTest, test execution, test evaluation, and test management become possible. After the execution, SUT output signals can be compared with previously obtained reference signals. MTest has, however, only limited means to express test behaviors which go beyond simple sequences, but are typical for control systems. The test evaluation bases only on the reference signals which are often not yet available at the early development phase yet and the process of test development is fully manual.
TPT addresses some of these problems. It uses an automaton-based approach to model the test behavior and associates with the states pre- and postconditions on the properties of the tested system (including the continuous signals) and on the timing. In addition, a dedicated run-time environment enables the execution of the tests. The test evaluation is based on a more sophisticated concept of signal feature. However, the specification of the evaluation happens in Python language, without any graphical support. TPT is a dedicated test technology for embedded systems controlled by and acting on continuous signals, but the entire test process is manual and difficult to learn.

In the following, numerous test approaches are analyzed. Firstly, several, randomly selected academic achievements on testing embedded systems are considered, in general. Then, the test methods applied in the industry are compared.

4.1 Analysis of the Academic Achievements

The approach, of which the realization is called Testing-UPPAAL [MLN03], presents a framework, a set of algorithms, and a tool for the testing of real-time systems based on symbolic techniques used in the UPPAAL model checker. The timed automata network model is extended to a test specification. This one is used to generate test primitives and to check the correctness of system responses. Then, the retrieved timed traces are applied so as to derive a test verdict. Here, online manipulation of test data is an advantage and this concept is partially reused in MiLEST. After all, the state-space explosion problem experienced by many offline test generation tools is reduced since only a limited part of the state space needs to be stored at any point in time. The algorithms use symbolic techniques derived from model checking to efficiently represent and operate on infinite state sets. The implementation of the concept shows that the performance of the computation mechanisms is fast enough for many realistic real-time systems [MLN03]. However, the approach does not deal with the hybrid nature of the system at all.

Similar as in MiLEST the authors of [BKB05] consider that a given test case must address a specific goal, which is related to a specific

requirement. The proposed approach computes one test case for one specific requirement. This strategy avoids handling the whole specification at once, which reduces the computation complexity. However, here again, the authors focus on testing the timing constraints only, leaving the hybrid behavior testing open.

The same holds for approach presented in [Tre08]. Input-output conformance (*ioco*) theory uses labelled transition systems (LTS) as models for specifications, implementations, and test generation source. *Ioco* defines conformance between implementations and specifications. Similarly as in MiLEST a completeness theorem checking the soundness and exhaustiveness of the *ioco* test method is available. [BB04] extends *ioco* towards testing real-time systems (i.e., *timed-ioco*). It uses timed-LTS and is based on an operational interpretation of the notion of *quiescence*. [FTW06] on the other hand, extends *ioco* towards testing based on symbolic execution (i.e., *sioco*). *Sioco* uses symbolic transition systems with an explicit notion of data and data-dependent control flow. The introduction of symbolism avoids the state-space explosion during test generation.

The authors of [CLP08] use two distinct, but complementary, concepts of sequence-based specification (SBS) and statistical testing. The system model and the test model for test case generation are distinguished, similar as in MiLEST. The system model is the black-box specification of the software system resulting from the SBS process. The test model is the usage model that models the environment producing stimuli for the software system as a result of a stochastic process. The framework proposed in this approach automatically creates Markov chain test models from specifications of the control model (i.e., SF design). The test cases with an oracle are built and statistical results are analyzed. Here, the formerly mentioned JUMBL methods are applied [Pro03]. Statistics are used as a means for planning the tests and isolating errors with propagating characteristics. The main shortcoming of this work is that mainly SF models are analyzed, leaving the considerable part of continuous behavior open (i.e., realized in SL design). This is not sufficient for testing the entire functionality of the system.

In contrast, the authors of [PHPS03] present an approach to generating test cases for hybrid systems automatically. These test cases can be used both for validating models and verifying the respective systems. This method seems to be promising, although as a source of test information two types of system models are used: a hybrid one and its abstracted version in the form of a discrete one. This practice may be very difficult when dealing with the continuous behavior described purely in SL.

The authors of [PPW+05] evaluate the efficiency of different MBT techniques. They apply the automotive network controller case study to assess different test suites in terms of error detection, model coverage, and implementation coverage. Here, the comparison between manually or automatically generated test suites both with and without models, at random or with dedicated functional test selection criteria is aimed at. As a result, the test suites retrieved from models, both automatically and manually, detect significantly more requirements errors than handcrafted test suites derived only from the requirements. The number of detected programming errors does not depend on the use of models. Automatically generated tests find as many errors as those defined manually. A sixfold increase in the number of model-based tests leads to an 11% increase [PPW+05] in detected errors.

Algorithmic testbench generation (ATG) technology [Ole07], though commercially available, is an interesting approach since here the test specification is based on the rule sets. These rule sets show that the high-level testing activities can be performed as a series of lower-level actions. By that, an abstraction level is introduced. This hierarchical concept is also used in MiLEST while designing the test system. ATG supports some aspects of test reactiveness, similar to MiLEST, and includes metrics for measuring the quality of the generated testbench specification. Finally, it reveals cost and time reduction while increasing the quality of the SUT as claimed in [Ole07].

4.2 Comparison of the Industrial Test Approaches

Considering the test approaches introduced in the appendix to this chapter, several diversities may be observed. Embedded*Validator* [EmbV, BBS04] uses model checking as test generation technology and thus, is limited to discrete model sectors. The actual evaluation method offers a basic set of constraints for extracting discrete properties, not addressing continuous signals. Only a few temporal constraints are checked. However, the mentioned properties of the model deal with the concept of signal features, whereas the basic verification patterns contribute to the test patterns and their reusability within the technique proposed in this chapter.

MEval [MEval] is especially powerful for back-to-back-tests and for regression tests, since even very complex reference signals are already available in this case. The option is excluded from further consideration.
MTest [MTest] with its CTE/ES [CTE] gives a good background for partitioning of test inputs into equivalence classes. The data coverage and test case specifications criteria are reused in MiLEST to some extent. Similarly as in SystemTest, the test evaluation is based only on reference signal-based specification, which constitutes a low abstraction level, thus it is not adopted for further development.

Reactis Tester [ReactT], T-VEC [TVec] or the method of [Pre03] present approaches for computing test sequences based on structural model coverage. It is searched for tests that satisfy MC/DC criteria. Their value is that the test suites are generated for units (functions, transitions) but also for the entire system or on integration level. The methods seem to be very promising due to their scope and automation grade, they cover only the structural testing criteria, though.

In Reactis Validator [ReactV, SD07] only two predefined validation patterns are available. Hence, a systematic test specification is not possible. This gap is bridged in MiLEST that provides *assertion – precondition* pairs. They enable the test evaluation functions to be related with the test data generation.

For SL DV [SLDV] a similar argumentation applies, although another test generation technology is used. An advantage of these three solutions is their possibility to cover both functional and structural test goals, at least to some extent.
SL VV [SLVV] gives the possibility of implementing a test specification directly next to the actual test object, but the standard evaluation functions cover only a very limited functionality range, a test management application is missing and test data must be created fully manually. A similar test assessment method, called ‘watchdog’ and ‘observer’, has been introduced by [CH98, DCB04], respectively.

TPT [TPT] is platform-independent and can be used at several embedded software development stages, which is not directly supported with MiLEST, although extensions are possible. It is the only tool from the list in Table 1 that enables reactive testing and signal-feature – based specification of the test evaluation algorithms. These concepts are reused and extended in the solution proposed in this chapter.

An extended classification of the test approaches with respect to the MBT taxonomy is provided in Table 1.

Table 1: Classification of the Selected Test Approaches based on the MBT Taxonomy.

MBT Categories, Options / *Selected Test Tools*	**Test Generation**		**Test Execution**	**Test Evaluation**	
	Test Selection Criteria	*Technology*	*Execution Options*	*Specification*	*Technology*
Embedded*Validator* [EmbV]	- does not apply	- automatic generation - model checking	- MiL, SiL - non-reactive	- requirements coverage	- manual specification - does not apply
MEval [MEval]	- does not apply since here back-to-back regression tests are considered	- does not apply	- MiL, SiL, PiL, HiL - non-reactive	- reference signals-based	- manual specification - offline evaluation
MTest with CTE/ES [MTest, CTE]	- data coverage - requirements coverage - test case specification - offline generation	- manual generation	- MiL, SiL, PiL, HiL - non-reactive	- reference signals-based	- manual specification - offline evaluation
Reactis Tester [ReactT]	- structural model coverage - offline generation	- automatic generation - model checking	- MiL, SiL, HiL - non-reactive	- test evaluation specifications	- automatic specification - offline evaluation
Reactis Validator [ReactV]	- structural model coverage - requirements coverage - offline generation	- automatic generation - model checking	- MiL, SiL - non-reactive	- test evaluation specifications	- manual specification - online evaluation
Simulink® Verification and Validation™ [SLVV]	- does not apply	- manual generation	- MiL - non-reactive	- requirements coverage	- manual specification - online evaluation
Simulink® Design Verifier™ [SLDV]	- structural model coverage - offline generation	- automatic generation - theorem proving	- MiL, SiL - non-reactive	- requirements coverage - test evaluation specifications	- manual specification - online evaluation
SystemTest™ [STest]	- data coverage - offline generation	- automatic generation	- MiL, SiL, HiL - non-reactive	- reference signals-based	- manual specification - offline evaluation
TPT [TPT]	- data coverage - requirements coverage - test case specification - offline and online generation	- manual generation	- MiL, SiL, PiL, HiL - reactive	- reference signal-feature – based	- manual specification - online and offline evaluation
T-VEC [TVec]	- structural model coverage - data coverage - requirements specification - offline generation	- automatic generation	- MiL, SiL - non-reactive	- test evaluation specifications [ROT98]	- automatic specification - does not apply

4.3 Analysis Summary

The main *shortcomings* and *problems* within the existing test solutions are the following:

- Automatic generation of test data is based almost only on structural test criteria or state-based models (e.g., SF charts), thus it is not systematic enough.
- For functional testing only manual test data specification is supported, which makes the test development process long and costly.
- The test evaluation is based mainly on the comparison of the SUT outputs with the entire reference signal flows. This demands a so-called *golden device* to produce such references and makes the test evaluation not flexible enough.
- Only a few test patterns exist. They are not structured and not categorized.
- The entire test development process is still almost only manual.
- Abstraction level is very low while developing the test design or selecting the test data variants.

Finally, none of the reviewed test approaches overcomes all the shortcomings given above at once.

In this chapter, only one selected approach will be provided as a proposal for a concrete MBT realization. This is the method developed by the authors of the chapter in [Zan08] and serves only as an illustration of the test concepts from the practical viewpoint.

Based on the recognized problems and the criteria that have been proven to be advantageous in the reviewed related work, the first shape of this proposal may be outlined. MiLEST deals with the following problems:

- Systematic and automatic test data generation process is supported. Here, not only a considerable reduction of manual efforts is advantageous, but also a systematic selection of test data for testing functional requirements including such system characteristics as hybrid, time-constrained behavior is achieved. By that, the method is cheaper and more comprehensive than the existing ones.
- The test evaluation is done based on the concept of signal feature, overcoming the problem of missing reference signals. These are not demanded for the test assessment any more.
- A catalog of classified and categorized test patterns is provided, which eases the application of the methodology and structures the knowledge on the test system being built.
- Some of the steps within the test development process are fully automated, which represents an improvement in the context of the efforts put on testing.
- A test framework enabling the specification of a hierarchical test system on different abstraction levels is provided. This gives the possibility to navigate through the test system easily and understand its contents immediately from several viewpoints.

A brief description of the MiLEST method is given below, whereas a report on its main contributions in relation to the related work will be discussed in the next chapters in depth.
The application of the same modeling language for both system and test design brings positive effects. It ensures that the method is relatively clear and it does not force the engineers to learn a completely new language. Thus, MiLEST is a SL add-on exploiting all the advantages of SL/SF application. It is a test specification framework, including reusable test patterns, generic graphical validation functions (VFs), test data generators, test control algorithms, and an arbitration mechanism collected in a dedicated library. Additionally, transformation functions in the form of ML scripts are available so as to automate the test specification process. For running the tests, no additional tool is necessary. The test method handles continuous and discrete signals as well as timing constraints.

5. SOLUTION PROPOSAL

5.1 The MiLEST Approach

Model-in-the-Loop for Embedded System Test (MiLEST) is a method that addresses the goals of a test approach given in Section 1.1 and refers to the selected test dimensions from Section 2.2.

The main innovation of MiLEST is *assuring the quality of embedded system by means of testing at early levels of their development*.
MiLEST is realized in ML/SL/SF. Although technical test extensions of this environment such as Simulink® Design Verifier™ [SLDV], SystemTest™ [STest], or MTest [MTest] already exist, they all encompass the limitations identified above. MiLEST introduces a more efficient approach to test automatically on model-level based on the so called signal-feature – oriented paradigm, specifically suited for functional testing of embedded systems.

5.2 The Test Development Process in MiLEST

The starting point of the MiLEST approach is to design a test specification model. Since at the early stage of system development reference signals are not available, a new method for describing the required SUT behavior is given. Based on a number of so called *signal features*, a novel, abstract understanding of a *signal* is defined.

A signal feature (SigF), called also signal property in [GW07, SG07, GSW08], is a formal description of certain predefined attributes of a signal. In other words, 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 of a signal. Giving some examples – *step response characteristics*, *step*, *minimum* etc. are considerable SigFs [ZSM06, MP07, GW07, SG07, GSW08, ZXS08].

Graphical instances of SigFs are given in Figure 3. The signal presented on the diagram is fragmented in time according to its descriptive properties resulting in: *decrease*, *constant*, *increase*, *local maximum*, *decrease* and *response*, respectively. This forms the backgrounds of the solution presented in this work.

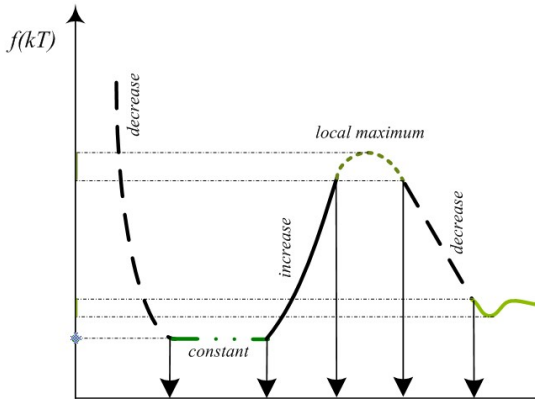

Figure 3: A Descriptive Approach to Signal Feature.

A feature can be predicated by other features, logical connectives or timing relations. These can be defined either between features within one signal or throughout more signals, e.g.:

- within(A_1)A_2	- if *SigF* A_1 occurs, *SigF* A_2 occurs at least once at the time when *SigF* A_1 is active
- after(y ms)A&B	- *SigF A* and *SigF B* occur together after *y* miliseconds
- during(A)B	- if *SigF A* occurs, *SigF B* occurs continuously during the activation time of *SigF A*
- A||¬C	- *SigF A* or negated event *C* occur
- A=v	- a set of *SigFs A* (e.g., maximum) which values are equal to *v*.

Further on, generic test data patterns are retrieved automatically out of marked portions of the test specification. The test data generator concretizes the test data. Its functionality has similarities to the classification tree method [GG93] and aims at a systematic signal production. The SUT input partitions and boundaries are used to find the meaningful representatives. Additionally, the SUT outputs are considered too. Hence, instead of searching for a scenario that fulfills the test objective it is assumed that this has been already achieved by defining the test specification. Further on, the method enables to deploy a searching strategy for finding different variants of such scenarios and the time points when they should start/stop.

The MiLEST test development process is depicted in Figure 4. The assumption resulting from the MBD paradigm is that the SUT model is available.

Also, the input/output interfaces have to be clearly defined and accessible.

Then, pattern for the generation of test harness model can be applied to the SUT model as denoted by *step I*. With this practice, an abstract frame for test specification is obtained. This is done automatically with a MiLEST transformation function.

Further on the test specification and test implementation phase is done in *step II*, where the test definition in MiLEST is concretized based on the test requirements. These requirements, called also test objectives, are usually available in a textual form, often hierarchical, starting from high level, down to concrete technical specification. These can be classified as a set of the abstract test scenarios. These test scenarios can be described by a conditional form which relates incoming SUT stimulation to the resulting SUT behavior by IF–THEN rules (cf. 1.1). The task of the test engineer is to manually refine the test specification based on the concept of validation functions patterns which include the test scenarios. Afterwards in *step III*, the structures for test stimuli and concrete test signals are generated. This step occurs automatically due to the application of the transformations. The test control design can be added automatically too. In that case *step IV* would be omitted. However, if the advantages of the test reactiveness are targeted, it should be refined manually. Finally in the test execution and test evaluation phase in *step V*, the tests (i.e., test cases) may be executed and the test results in the form of verdicts are obtained.

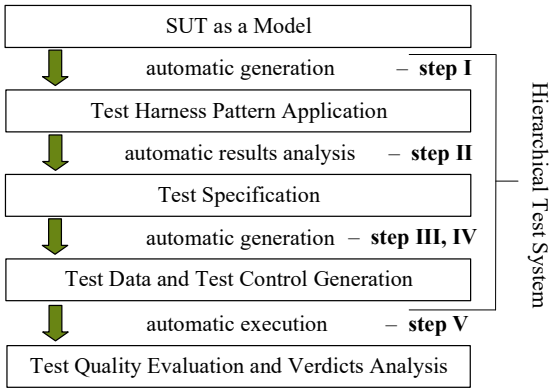

Figure 4: The MiLEST Test Development Process.

In Figure 5, a generic pattern of the test harness used in MiLEST is presented. The test data (i.e., test signals produced along the test cases), on the left, are generated within the test data generator. The test specification, on the right, is constructed by analyzing the SUT functionality requirements and deriving the test objectives out of them. It includes the abstract test scenarios, test evaluation algorithms, test oracle and an arbitration mechanism [ZSM06, ZMS07a, ZSM07b, MP07, Zan07, ZXS08].

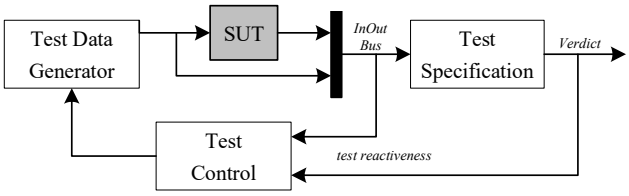

Figure 5: The MiLEST Test Harness Pattern.

The test specification is built applying the test patterns available in the MiLEST library. It is developed in such a way that it includes the design of a test evaluation as well – opposite to a common practice in the embedded systems domain, where the test evaluation design is considered last. Afterward, based on the already constructed test model, the test data generators are retrieved. These are embedded in a dedicated test data structure and are derived out of the test design automatically. The generation of test signals variants, their management and combination within a test case is also supported, similarly as the synchronization of the obtained test stimuli. Finally, the SUT model fed with the previously created test data is executed and the evaluation unit supplies verdicts on the fly.

The first step in the test development process is to identify the test requirements. For that purpose a high level pattern within the test specification unit is applied. The number of test requirements can be chosen in the graphical user interface (GUI) that updates the design and adjusts the structural changes of the test model (e.g., it adjusts the number of inputs in the arbitration unit). The situation is illustrated in Figure 6.

a)

b)

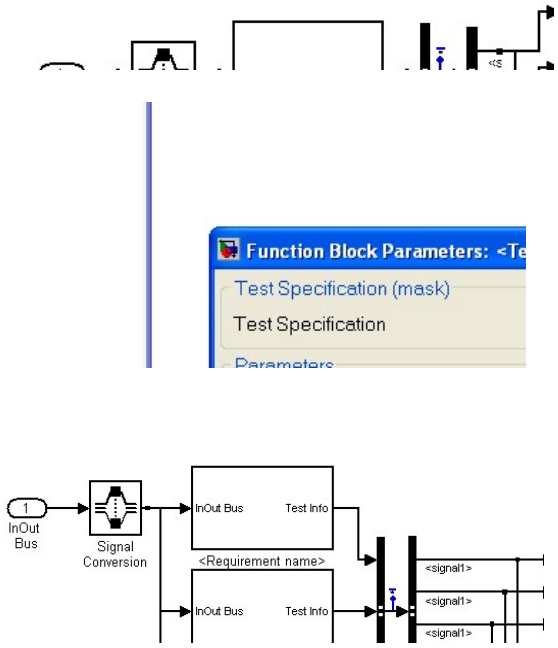

Figure 6: A Pattern for the Test Requirement Specification.
a) Instantiation for One Test Requirement.
b) Instantiation for Three Test Requirements.

Further on, validation functions (VFs) [ZSM06, MP07] are introduced to define the test scenarios, test evaluation and test oracle in a systematic way. VFs serve to evaluate the execution status of a test case by assessing the SUT observations and/or additional characteristics/parameters of the SUT. A VF is created for any single requirement following the generic conditional rule:

(1.1) *IF preconditions set THEN assertions set*

A single informal requirement may imply multiple VFs. If this is the case, the arbitration algorithm accumulates the results of the combined IF-THEN rules and delivers a common verdict. Predefined *verdict* values are pass, fail, none and error. Retrieval of the local verdicts for a single VF is also possible.

A preconditions set consists of at least one extractor for signals' feature or temporally and logically related signal features, a comparator for every single extractor and one unit for preconditions synchronization (PS).
An assertions set is similar, it includes however at least one unit for preconditions and assertions synchronization (PAS), instead of a PS.

VFs are able to continuously update the verdicts for a test scenario already during test execution. They are defined to be independent of the currently applied test data. Thereby, they can set the verdict for all possible test data vectors and activate themselves (i.e., their assertions) only if the predefined conditions are fulfilled.
An abstract pattern for a VF (shown in Figure 7) consists of a preconditions block which activates the assertions block, where the comparison of actual and expected signal values occurs. The activation and by that, the actual evaluation proceeds only if the preconditions are fulfilled.

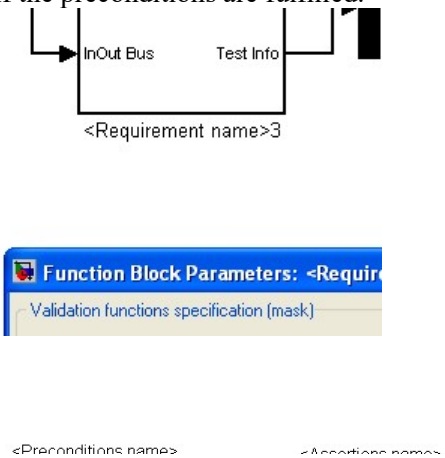

Figure 7: Structure of a VF – a Pattern and its GUI.

The easiest assertion blocks checking time independent features are built following the schema presented in Figure 8.

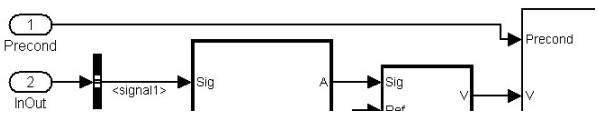

Figure 8: Assertion Block – a Pattern.

They include a SigF extraction part, a block comparing the actual values with the expected ones and a PAS synchronizer. Optionally, some signal deviations within a permitted tolerance range are allowed. Further schemas of preconditions and assertions blocks for triggered features are discussed in [MP07] in detail.

A further step in the MiLEST test development process is the derivation of the corresponding structures for test data sets and the concretization of the signal variants. The entire step related to test data generation occurs fully automatically as a result of the application of transformations. Similarly as on the test specification side, the test requirements level for the test data is generated. This is possible due to the knowledge gained from the previous phase. The pattern applied in this step is shown in Figure 9.

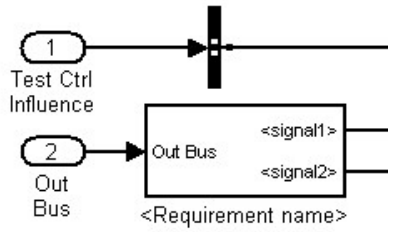

Figure 9: Test Requirement Level within the Test Data Unit – a Pattern.

Moreover, concrete SigFs on predefined signals are produced afterwards. The test signals are generated following a conditional rule in the form (see 1.2):

(1.2) *IF preconditions set THEN generations set*

Knowing the SigFs appearing in the preconditions of a VF, the test data can be constructed from them. The preconditions typically depend on the SUT inputs; however they may also be related to the SUT outputs at some points. Every time a *SigF extractor* occurs for the assertion activation, a corresponding *SigF generator* may be applied for the test data creation. Giving a very simple example – for detection of a given *signal value* in a precondition of a VF, a signal crossing this value during a default time is required. Apart from the feature generation, the SUT output signals may be checked for some constraints, if necessary (cf. Figure 10). The feature generation is activated by a Stateflow (SF) diagram sequencing the features in time according to the default temporal constraints (i.e., *after(time1)*). A switch is needed for each SUT input to handle the dependencies between generated signals. *Initialization & Stabilization* block enables to reset the obtained signal so that there are no influences of one test case on another.

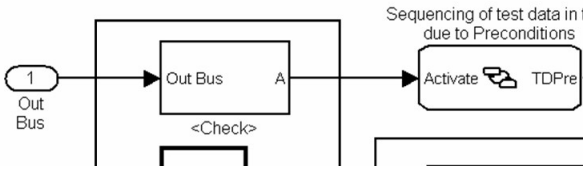

Figure 10: Structure of the Test Data Set – a Pattern.

The patterns in Figure 10 and the concrete *feature generators* are obtained as a result of the automatic transformations. The general principle of the transformation is that if a given *feature or feature dependency extraction* is detected in the source (i.e. preconditions part of a VF), then the action to generate the target (i.e. *feature generator* in the test data structure) is performed. A set of transformation rules has been implemented. Afterwards, the concrete test data variants are constructed based on the generators obtained from the transformations. The assumption and necessary condition for applying the variants generation method is the definition of the *signal ranges* and *partition points* on all the stimuli signals according to the requirements or engineer's experience. Equivalence partitioning and boundary value analysis are used in different combinations to produce then concrete variants for the stimuli.

When a test involves multiple signals, the combination of different signals and their variants has to be computed. Several combination strategies are known to construct the test cases – *minimal combination, one factor at a time* and *n-wise combination* [LBE+04, GOA05]. Combination strategies are the selection methods where test cases are identified by combining values of different test data parameters according to some predefined criteria.

A similar sequencing algorithm as for the test data applies for ordering the test cases on a higher hierarchy level while dealing with a number of requirements. This aspect is called test control. A traditional understanding of the control makes it responsible for the order of test cases over time [ETSI07]. It may invoke, change or stop the test case execution due to the influence of the verdict values coming from the test evaluation. Thus, the test cases are sequenced according to the previously specified criteria (e.g., *pass* verdict).

The test patterns used for realizing a test specification proposed in this work are collected in a library and can be applied during the test design phase.

After the test specification has been completed, the resulting test design can be executed in SL. Additionally, a report is generated including the applied test data, their variants, test cases, test results and the calculated quality metrics.

The *options* that MiLEST covers with respect to the MBT taxonomy are listed in Table 2.

Table 2: Classification of MiLEST with respect to the MBT Taxonomy.

Test Generation Selection Criteria and Technology	*Test Execution Options*	*Test Evaluatio Specification and Technolog*
- data coverage - requirements coverage - test case specifications - automatic generation - offline generation	- MiL level - reactive	- reference sig based - requirements - output data c - test evaluatic specifications

5.3 MiLEST Effects and Benefits

MiLEST is a SL add-on exploiting all the advantages of SL/SF application. It is a comprehensive test modeling framework enabling to build a full test specification for embedded systems based on the ready-to-use test patterns. These are generic graphical VFs, test data generators, test control algorithms and an arbitration mechanism collected in a dedicated library. Additionally, transformation functions in the form of ML scripts are available so as to automate the test specification process. For running the tests, no additional tool is necessary. The application of the same modeling language for both system and test design brings positive effects. It ensures that the method is relatively clear and it does not force the engineers to learn a completely new language. The test method handles continuous and discrete signals as well as timing constraints.

The MiLEST *signal feature* approach provides the essential benefit of describing signal flows, their relation to other signal flows and/or to discrete events on an abstract, logical level. This prevents not only the user from error on too technical specifics, but also enables test specification in early development stages. The absence of concrete reference signals is compensated by a logical description of the signal flow characteristics.

This is a fundamental contribution of MiLEST, based on which test case generation and test evaluation have been developed. Numerous signals' features have been identified; feature extractors, comparators and feature generators have been realized in the MiLEST library. The MiLEST library allows to generate and to analyze both discrete and continuous signals. The test evaluation can be performed offline but also online, which enables an active test control, opens perspectives for dynamic test generation algorithms and provides extensions of reactive testing. Also, new ways for first diagnosis activities and failure management are possible.

In addition, MiLEST automates the systematic test data selection and generation, providing a better test coverage (e.g., in terms of requirements coverage or test purpose coverage) than manually created test cases.

Furthermore, the automated test evaluation reveals a considerable progress, in contrast to the low level of abstraction and the manual assessment means used in existing approaches. The tester can retrieve immediately the test results and is assured about the correct interpretation of the SUT reactions along the tests.

The signal evaluation used in the automated test evaluation can even run independently of the SUT stimulation. The signal evaluation is not based on the direct comparison of SUT signals and previously generated concrete reference signals. Instead, it offers an abstract way for requirements on signal characteristics. The signal evaluation is particularly robust and can be used in other contexts than testing e.g. for online monitoring.

The test specification enables to trace the SUT requirements. The way how it is defined gives the possibility to trace root faults by associating local test verdicts to them, which is a central element in fault management of embedded systems.

Finally, the MiLEST test quality metrics reveal the strengths of the approach by providing high test coverage in different dimensions and good analysis capabilities. The MiLEST projects demonstrated a quality gain of at least 20% [ZMS08].

6. CASE STUDY

In the following, the functionality of a car door is tested with a focus on MiL and SiL level. It has been modeled in ML/SL/SF. The software embedded in a car door controls the position of the window pane and flashing of the light located in the mirror. In the following, one functional requirement for the SUT (see Figure 11) will be described only because of simplicity.

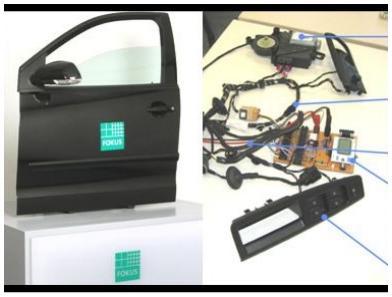

Figure 11: SUT – Car Door.

In Figure 12, the MiLEST test harness including test data generation and test specification units is presented.

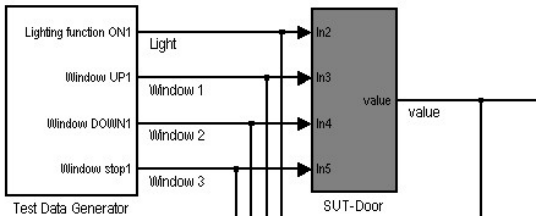

Figure 12: Test Harness for the Door.

In Figure 13, details of the test specification are highlighted so as to give an insight into the test system that is built in a hierarchical way.

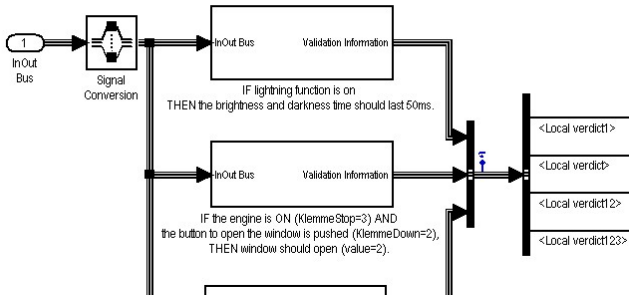

Figure 13: Test Specification View – Selected Test Requirements.

For the requirement *If the flashing is activated, the brightness and darkness times should equal to 50 ms.'*, which is defined in Figure 13 in the first block at the top, the following IF-THEN rules are obtained:

```
IF flashing function is on THEN the brightness
time should last 50 ms.
IF flashing function is on THEN the darkness
time should last 50 ms.
```

These are modeled in the validation functions (VFs) (see Figure 14) so as to consequently enable a systematic and automatic generation of test cases and an automatic test evaluation during the test execution.

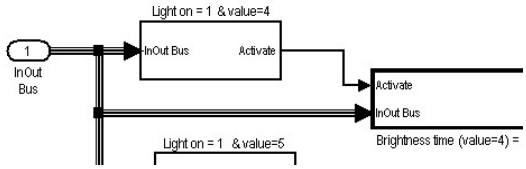

Figure 14: Test Specification View – Selected Validation Functions.

As a result two local verdicts are obtained for this particular test requirement, one for each VF. These are presented in Figure 15. The flows of both signals indicate that the test cases passed. This means that the timing of the lightening control is correct.

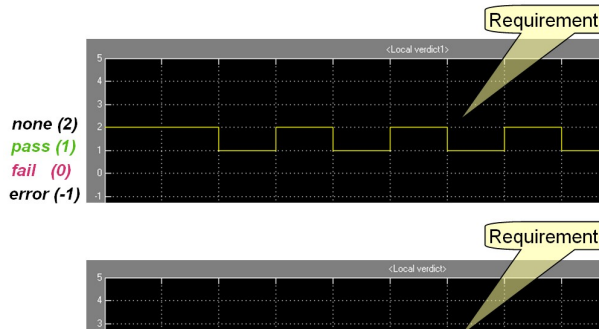

Figure 15: Test Results – two Local Verdicts.

Additionally, the scenario can be run in connection with the real software on the microprocessor of the door, as the test model has been connected to the real SUT via a serial cable. Then, the flashing is observed directly on the hardware as well.

This demonstrates the application of MiLEST on the MiL and SiL level.

7. FUTURE TRENDS AND CONCLUSIONS

Applying MBT methods (e.g., MiLEST) the test engineer needs considerably less effort in the context of test generation and test execution, concentrating on the test logic and the corresponding test coverage. By that, the costs of the entire process of software development are definitely cut.

There is still plenty of work concerning future achievements of MBT. For example, in MiLEST the conditional rules utilized within the test specification an automatic transformation of the incorrect functions (e.g., *IF constrained output THEN constrained input*) into the reverse, based on the transposition rule [CC00] could be easily realized.

Then, the issue of handling the SUT outputs existing in the preconditions of a VF (e.g., *IF constrained input AND constrained output THEN constrained output*) has been only partially solved. It is checked if the test data generator has been able to produce the meaningful signals out of such a specification. If this is not the case, a manual refinement is needed in the Initialization/Stabilization block at the test case level. An automatic way of relating the functional test cases and test sequences to each other could be struggled for, though.

The test stimuli generation algorithms can be still refined as not all the signal features are included in the realization of the engine. Also, they could be enriched with different extensions applying, in particular, the constraint solver and implicit partitions.

Further work regards the negative testing as well. Here, the test data generation algorithm can be extended so as to produce the invalid input values or exceptions. However, the test engineer's responsibility would be to define what kind of system behavior is expected in such a case.

An interesting possibility pointed out by [MP07] would be to take advantage of the reactiveness path for optimizing the generated test data iteratively. In that case, the algorithm could search for the SUT inputs leading to a fail automatically (cf. evolutionary algorithms [WW06]).

Moreover, since software testing alone cannot prove that a system does not contain any defects or that it does have a certain property (i.e., that the signal fulfils a certain signal feature), the proposed VFs could be a basis for developing a verification mechanism based on formal methods in a strict functional context [LY94, ABH+97, BBS04, DCB04]. Thus, the perspective of mathematically proving the correctness of a system design remains open for further research within the proposed quality assurance (QA) framework.

Besides the test quality metrics [ZMS08, Zan08], also other criteria may be used to assess the proposed test method, the following being only some of them:

- the efficiency of faults detection – aimed to be as high as many VFs are designed under the assumption that the corresponding test data are generated and applied properly
- the percentage of test cases / test design / test elements reusability
- time and effort required for the test engineer to specify the test design, which is relatively low only for persons knowing the technologies behind the concepts
- the percentage of the effective improvement of the test stimuli variants generation in contrast to the manual construction.

Regarding the realization of MiLEST prototype, several GUIs (e.g., for transformation functions, for quality metrics application, for variants generation options or for the test execution) would definitely help the user to apply the method faster and more intuitive.

Furthermore, the support of different test execution platforms for the proposed method has not been sufficiently explored yet. Interesting research questions concern the extent to which the concrete test cases could be reused on various execution levels. Consequently, real-time properties on the run-time execution level [NLL03] in the sense of scheduler, priorities or threads have not been considered.

Current research work aims at designing a new platform independent test specification language, one of the branches called TTCN-3 continuous [SBG06, BKL07, SG07, GSW08]. Its fundamental idea is to obtain a domain specific test language, that is executable and that unifies tests of communicating, software-based systems in all of the automotive subdomains (e.g., telematics, power train, body electronics etc.) integrating the test infrastructure as well as the definition and documentation of tests. It should keep the whole development and test process efficient and manageable. It must address the subjects of test exchange, autonomy of infrastructure, methods and platforms, and the reuse of tests.

TTCN-3 has the potential to serve as a testing middleware, indeed. It provides concepts of local and distributed testing. A test solution based on this language can be adapted to concrete testing environments. However, while the testing of discrete controls is well understood and available in TTCN-3, concepts for specification-based testing of continuous controls and for the relation between discrete and the continuous system parts are still under ongoing research [SBG06, SG07, GSW08].

This option becomes interesting especially in the context of a new paradigm – AUTomotive Open System Architecture (AUTOSAR) that has been observed as an automotive development trend for the last few years. Traditional TTCN-3 is already in use to test discrete interactions within this architecture. The remaining hybrid or continuous behavior could be tested with TTCN-3 embedded.

Another graphical test specification language being already in the development stage is UML Testing Profile for Embedded Systems (UTPes) [DM_D07, D-Mint08]. Its backgrounds root from UTP, TPT and Model-in-the-Loop for Embedded System Test, abbreviated as MiLEST (the approach proposed in this chapter). These are coordinated and synchronized with the concepts of TTCN-3 embedded too.

Apart from the tools commonly known in the automotive industry, further approaches exist and are applied for testing the embedded systems.

Investigation in the context of transformations from UTPes into executable MiLEST test models and further on, into TTCN-3 for embedded systems would be an interesting approach enabling to exchange the test specification without loosing the detail information since all three approaches base on a similar concept of signal feature. This would give the advantage of having a comprehensive test strategy applicable for all the development platforms.

REFERENCES:

[AB02] Ammann, P., Black, P. E.: Model Checkers. In *Software Testing*, National Institute of Standards and Technology, Technical Report NIST-IR 6777, 2002.

[ABH+97] Amon, T., Borriello, G., Hu, T., Liu, J.: Symbolic Timing Verification of Timing Diagrams Using Presburger Formulas. In *Proceedings of the 34th Annual Conference on Design Automation*, Pages: 226 – 231, ISBN: 0-89791-920-3. ACM New York, NY, U.S.A., 1997.

[Aic05] Aichernig, B. K.: On the Value of Fault Injection on the Modeling Level, In *Proceedings of the Overture Workshop*, N. Plat and P. G. Larsen (Eds.), Newcastle Upon Tyne, UK, 2005.

[APJ+03] Paiva, A. C. R., Faria, J. C. P., Vidal, R. M.: Specification-Based Testing of User Interfaces, In *Interactive Systems, Design, Specification and Verification*, 2003.

[Art05] *The Artist Roadmap for Research and Development. Embedded Systems Design.* LNCS, Volume 3436, Editors: Bouyssounouse, B., Sifakis, J., ISBN: 978-3-540-25107-1. 2005.

[AuFo] Munich University of Technology, Department of Computer Science, AutoFocus, Research Tool for System Modelling, http://autofocus.in.tum.de/ [04/20/2008].

[AutD] dSpace GmbH, AutomationDesk, Commercial Tool for Testing, http://www.dSpace.de/Goto?Releases [04/20/2008].

[BB04] Brandan Briones, L., Brinksma, E.: *A test generation framework for quiescent real-time systems.* Internal Report 48975. University of Twente, 2004.

[BBK98] Broy, M., Von der Beeck, M., Krüger, I.: *Softbed: Problemanalyse Für Ein Großverbundprojekt "Systemtechnik Automobil – Software Für Eingebettete Systeme".* Ausarbeitung für das BMBF. 1998 (In German).

[BBS04] Bienmüller, T., Brockmeyer, U., Sandmann, G.: Automatic Validation of Simulink/Stateflow Models, Formal Verification of Safety-Critical Requirements, Stuttgart. 2004.

[Bei95] Beizer, B.: *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. ISBN-10: 0471120944. John Wiley & Sons, Inc, 1995.

[BJK+05] *Model-Based Testing of Reactive Systems*, Editors: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A., No. 3472. In LNCS, Springer-Verlag, 2005.

[BKB05] Bodeveix, J.-P., Koné, O., Bouaziz, R.: Test Method for Embedded Real-Time Systems. In *Proceedings of the European Workshop on Dependable Software Intensive Embedded Systems*, ERCIM, Pages: 1 – 10. Porto, 2005.

[BKL07] Bräuer, J., Kleinwechter, H., Leicher, A.: µttcn – An Approach to Continuous Signals in TTCN-3. In *Proceedings of Software Engineering* (Workshops), Editors: Bleek, W.-G., Schwentner, H., Züllighoven, H., Series LNI, Volume 106, Pages: 55 – 64, ISBN: 978-3-88579-200-0. GI, 2007.

[BKP+07] Broy, M., Krüger, I. H., Pretschner, A., Salzmann, C.: Engineering Automotive Software. In *Proceedings of the IEEE*, Volume 95, Number 2, Pages: 356 – 373. 2007.

[BLL+04] E. Bernard, B. Legeard, X. Luck, and F. Peureux, Generation of Test Sequences from Formal Specifications: GSM 11-11 Standard Case Study, *Software Testing, Verification and Reliability*, Vol. 34(10), Pp. 915-948, 2004

[BS98] BS 7925-2:1998, *Software Testing. Software Component Testing*, British Standards Institution, ISBN: 0580295567. 1998.

[CC00] Copi, I. M., Cohen, C.: *Introduction to Logic*, 11th Ed. Upper Saddle River, NJ: Prentice-Hall, 2000.

[CFS04] Conrad, M., Fey, I., Sadeghipour, S.: Systematic Model-Based Testing of Embedded Control Software – The MB3T Approach. In *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems*, Edinburgh, United Kingdom. 2004.

[CGN+05] Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, Microsoft Research, MSR-TR-2005-59, May, 2005.

[CH98] Conrad, M., Hötzer, D.: Selective Integration of Formal Methods in the Development of Electronic Control Units. In *Proceedings of the ICFEM* 1998, 144-Electronic Edition. 1998.

[ChM] 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/ [05/21/2008].

[CLP08] Carter, J. M., Lin, L., Poore, J. H.: 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, Berlin, Germany. Fraunhofer IRB Verlag, 2008.

[Con04a] Conrad, M.: *Modell-Basierter Test Eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien*. PhD Thesis. Deutscher Universitätsverlag, Wiesbaden (D), 2004 (In German).

[Con04b] Conrad, M.: A Systematic Approach to Testing Automotive Control Software, Detroit U.S.A., *SAE Technical Paper* Series, 2004-21-0039. 2004.

[CTE] Razorcat Development GmbH, Classification Tree Editor for Embedded Systems – CTE/ES, Commercial Tool for Testing, http://www.razorcatdevelopment.de/ [04/20/08].

[Dai06] Dai, Z. R.: *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, 2006.

[DCB04] Dajani-Brown, S., Cofer, D., Bouali, A.: formal Verification of an Avionics Sensor Voter Using SCADE. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Volume 3253/2004, LNCS, Pages: 5 – 20, ISBN: 978-3-540-23167-7, ISSN: 0302-9743 1611-3349. Springer-Verlag Berlin/Heidelberg, 2004.

[Dess01] DESS – Software Development Process for Real-Time Embedded Software Systems, *The DESS Methodology*. Deliverable D.1, Version 01 – Public, Editors: Van Baelen, S., Gorinsek, J., Wills, A. 2001.

[DF03] Dulz, W., Fenhua, Z.: 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, U.S.A., 2003.

[Dij72] Dijkstra, E. W.: Notes on Structured Programming. In Structured Programming, Volume 8 of *A.P.I.C. Studies In Data Processing*, Part 1, Editor: Hoare C. A. R., Pages: 1 – 82. Academic Press, London/New York, 1972.

[DM_D07] D-MINT Consortium. Deployment of Model-Based Technologies to Industrial Testing, Milestone 1, Deliverable 2.1 – *Test Modeling, Test Generation and Test Execution with Model-Based Testing*. 2007.

[D-Mint08] D-MINT Project – Deployment of Model-Based Technologies to Industrial Testing. 2008. http://d-mint.org/ [05/09/08].

[DS02] Dempster, D., Stuart, M.: *Verification Methodology Manual, Techniques for Verifying HDL Designs*, ISBN: 0-9538-4822-1. Teamwork International, June 2002.

[dSP] dSpace GmbH, http://www.dSpace.de/ww/en/gmb/home.cfm [04/22/08].

[EmbV] OSC – Embedded Systems AG, Embedded*Validator*, Commercial Verification Tool, http://www.osc-es.de [04/20/08].

[Enc03] Encontre, V.: *Testing Embedded Systems: Do You Have The Guts for It?*. IBM, 2003. http://www-128.ibm.com/developerworks/rational/library/459.Html [04/18/2008].

[ETSI07] ETSI European Standard (ES) 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. 2007.

[FTW06] Frantzen, L., Tretmans, J., Willemse T. A. C., A Symbolic Framework for Model-Based Testing, In *Formal Approaches to Software Testing and Runtime Verification*, ISSN: 0302-9743 (Print) 1611-3349 (online), Volume 4262/2006. Springer Berlin / Heidelberg, 2006.

[GG93] Grochtmann, M., Grimm, K.: Classification Trees for Partition Testing. In *Software Testing, Verification & Reliability*, Volume 3, Number 2, Pages: 63 – 82. Wiley, 1993.

[GOA05] Grindal, M., Offutt, J., Andler, S. F.: Combination Testing Strategies: A Survey. In *Software Testing, Verification and Reliability*. Volume 15, Issue 3, Pages: 167 – 199. John Wiley & Sons, Ltd., 2005.

[GSW08] Großmann, J., Schieferdecker, I., Wiesbrock, H. W.: Modeling Property Based Stream Templates with TTCN-3. In *Proceedings of the IFIP 20th Intern. Conf. on Testing Communicating Systems* (Testcom 2008), Tokyo, Japan. 2008.

[Gut99] Gutjahr, W. J.: 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, 1999.

[GW07] Gips C., Wiesbrock H.-W. 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, 2007 (In German).

[Hel+05] Helmerich, A., Koch, N. and Mandel, L., Braun, P., Dornbusch, P., Gruler, A., Keil, P., Leisibach, R., Romberg, J., Schätz, B., Wild, T. Wimmel, G.: *Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area*, Final Report for the European Commission, Brussels Belgium, 2005.

[Het98] Hetzel, W. C.: *The Complete Guide to Software Testing*. Second Edition, ISBN: 0-89435-242-3. QED Information Services, Inc, 1988.

[IEC05] Functional Safety and IEC 61508, 2005. http://www.iec.ch/zone/fsafety/fsafety_entry.htm [05/09/08].

[ISO_FS] ISO/NP PAS 26262, Road Vehicles - Functional Safety, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43464 [05/09/08].

[ISTQB06] International Software Testing Qualification Board. *Standard Glossary of Terms Used In Software Testing*. Version 1.2, Produced By The 'Glossary Working Party', Editor: Van Veenendaal E. 2006.

[JJR05] Jeannet, B., Jéron, Rusu, V., Zinovieva, E.: Symbolic Test Selection Based on Approximate Analysis. In *Proceedings of TACAS 2005*. Editors: Halbwachs, N., Zuck, L., LNCS 3440, Pages: 349 – 364, Berlin, Heidelberg. Springer-Verlag, 2005.

[JUMB] Software Quality Research Laboratory, Java Usage Model Builder Library – JUMBL, Research Model-Based Testing Prototype, http://www.cs.utk.edu/sqrl/esp/Jumbl.html [04/20/08].

[KHJ07] Kamga, J., Herrmann, J., Joshi, P.: Deliverable: *D-MINT Automotive Case Study – Daimler AG*, Deliverable 1.1, Deployment of Model-Based Technologies to Industrial Testing, Itea2 Project, 2007.

[KLP+04] Kosmatov, N., Legeard, B., Peureux, F., Utting, M.: 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, 2004.

[LBE+04] Lamberg, K., Beine, M., Eschmann, M., Otterbach, R., Conrad, M., Fey, I.: Model-Based Testing of Embedded Automotive Software Using MTest. *In Proceedings of SAE World Congress*, Detroit, US, 2004.

[Leh03] Lehmann, E. (then Bringmann, E.): *Time Partition Testing, Systematischer Test des Kontinuierlichen Verhaltens von Eingebetteten Systemen*, PhD Thesis, Technical University Berlin, 2003 (In German).

[LK08] Lehmann, E., Krämer, A.: Model-Based Testing of Automotive Systems. *In Proceedings of IEEE ICST 08*, Lillehammer, Norway. 2008.

[LKK+06] Lehmann, E., Krämer, A., Lang, T., Weiss, S., Klaproth, Ch., Ruhe, J., Ziech, Ch.: *Time Partition Testing Manual*, Version 2.4, 2006.

[LV04] Lazić, Lj., Velašević, D.: Applying Simulation and Design of Experiments to the Embedded Software Testing Process. In *Software Testing, Verification & Reliability*, Volume 14, Issue 4, Pages: 257 – 282, ISSN: 0960-0833. John Wiley and Sons Ltd. Chichester, UK, 2004.

[LY94] Lee, T., Yannakakis, M.: 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, 1994.

[MA00] Marre, B., Arnould, A.: 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, 2000.

[MaTL] All4Tec, Markov Test Logic – Matelo, Commercial Model-Based Testing Tool, http://www.all4tec.net/ [04/20/08].

[MATT] The University of Montana, MATLAB Automated Testing Tool – MATT, Research Model-Based Testing Prototype, http://www.cs.umt.edu/rtsl/matt/ [04/20/08].

[MBG] MBTech Group, http://www.mbtech-group.com/cz/electronics_solutions/test_engineering/provetechta_overview.html [04/22/08].

[MDA] OMG: MDA Guide V1.0.1, June, 2003. http://www.omg.org/docs/omg/03-06-01.pdf [05/09/08].

[MEval] IT Power Consultants, MEval, Commercial Tool for Testing, http://www.itpower.de/meval.html [04/20/2008].

[MLN03] Mikucionis, M., Larsen, K. G, Nielsen, B.: *Online on-The-Fly Testing of Real-Time Systems*, ISSN 0909-0878. BRICS Report Series, RS-03-49, Denmark, 2003.

[MP07] Marrero Pérez, A.: *Simulink Test Design for Hybrid Embedded Systems*, Diploma Thesis, Technical University Berlin, January 2007.

[MTest] dSpace GmbH, MTest, Commercial MBT Tool, http://www.dspaceinc.com/ww/en/inc/home/products/Sw/expsoft/mtest.cfm [04/20/2008].

[MW04] Maxton, G. P., Wormald, J.: *Time for a Model Change: Re-Engineering the Global Automotive Industry*, ISBN: 978-0521837156. Cambridge University Press, 2004.

[Mye79] Myers, G. J.: *The Art of Software Testing*. ISBN-10: 0471043281. John Wiley & Sons, 1979.

[Neu04] Neukirchen, H. W.: Languages, *Tools and Patterns for the Specification of Distributed Real-Time Tests*, PhD Thesis, Georg-August-Universiät zu Göttingen, 2004. http://webdoc.sub.gwdg.de/Diss/2004/Neukirchen/Index.html [05/09/08].

[NLL03] Nicol, D. M., Liu, J., Liljenstam, M., Guanhua, Y.: Simulation of Large Scale Networks Using SSF. In *Proceedings of the 35th Conference on Winter Simulation: Driving Innovation*, Volume 1, Pages: 650 – 657, Vol.1, New Orleans, Louisiana. ACM, 2003.

[Ole07] Olen, M.: *The New Wave in Functional Verification: Algorithmic Testbench Technology*, White Paper, Mentor Graphics Corporation. 2007. http://www.edadesignline.com/showarticle.jhtml?Articleid=197800043 [08/08/08].

[PHPS03] Philipps, J., Hahn, G., Pretschner, A., Stauner, T.: Prototype-Based Tests for Hybrid Reactive Systems. In *Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping*, Pages: 78 – 84, ISSN: 1074-6005, ISBN: 0-7695-1943-1. IEEE Computer Society, Washington, DC, U.S.A., 2003.

[PPW+05] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., Stauner, T.: One Evaluation of Model-Based Testing and its Automation. In *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, U.S.A., Pages: 392 – 401, ISBN: 1-59593-963-2. ACM New York, NY, USA, 2005.

[Pre03] Pretschner, A.: Compositional Generation of MC/DC Integration Test Suites. *In Proceedings TACOS'03*, Pages: 1 – 11. Electronic Notes in Theoretical Computer Science 6, 2003. http://Citeseer.Ist.Psu.Edu/633586.Html [05/09/08].

[Pre03b] Pretschner, A.: *Zum Modellbasierten Funktionalen Test Reaktiver Systeme*. PhD Thesis. Technical University Munich, 2003 (In German).

[Pre04] Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S.: 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, 2004.

[Pro03] Prowell, S. J.: JUMBL: A Tool for Model-Based Statistical Testing. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, Volume 9, ISBN: 0-7695-1874-5. IEEE Computer Society Washington, DC, 2003.

[Rau02] Rau, A.: *Model-Based Development of Embedded Automotive Control Systems*, PhD Thesis, University of Tübingen, 2002.

[ReactT] Reactive Systems, Inc., Reactis Tester, Commercial Model-Based Testing Tool, http://www.reactive-systems.com/tester.msp [04/20/08].

[ReactV] 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 [07/03/08].

[ROT98] Richardson, D, O'malley, O., Tittle, C.: 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, NY, 1998.

[SBG06] Schieferdecker, I., Bringmann, E., Grossmann, J.: Continuous TTCN-3: Testing of Embedded Control Systems. In *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems*, International Conference on Software Engineering, ISBN: 1-59593-402-2, Shanghai, China. ACM New York Press, 2006.

[SCB] TNI-Software, Safety Checker Blockset, Commercial Model-Based Testing Tool, http://www.tni-software.com/en/produits/safetychecker blockset/index.php [04/20/08].

[SD07] Sims S., Duvarney D. C.: 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, NY, U.S.A., 2007.

[SG07] Schieferdecker, I., Großmann, J.: 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, 2007.

[SL05] Spillner, A., Linz, T.: *Basiswissen Softwaretest, Aus- und Weiterbildung zum Certified Tester Foundation Level nach ASQF- und ISTQB-Standard*. ISBN: 3-89864-358-1. dpunkt.Verlag GmbH Heidelberg, 2005 (In German).

[SLDV] The MathWorks™, Inc., Simulink® Design Verifier™, Commercial Model-Based Testing Tool, http://www.mathworks.com/products/sldesignverifier [04/20/2008].

[SLVV] The MathWorks™, Inc., Simulink® Verification and Validation™, Commercial Model-Based Verification and Validation Tool, http://www.mathworks.com/products/simverification/ [04/20/2008].

[SRK+00] Silva, B. I., Richeson K., Krogh B., Chutinan A.: Modeling and Verifying Hybrid Dynamic Systems Using Checkmate. In *Proceedings of the 4th International Conference on Automation of Mixed Processes* (ADPM 2000), Pages: 237 – 242. 2000.

[STB] TNI-Software, Safety Test Builder, Commercial Model-Based Testing Tool, http://www.tni-software.com/en/produits/safetytestbuilder/index.php [04/20/08].

[STest] The MathWorks™, Inc., SystemTest™, Commercial Tool for Testing, http://www.mathworks.com/products/systemtest/ [04/20/2008].

[SZ06] Schäuffele, J., Zurawka, T.: *Automotive Software Engineering*, ISBN: 3528110406. Vieweg, 2006.

[TPT] PikeTec, Time Partitioning Testing – TPT, Commercial Model-Based Testing Tool, http://www.piketec.com/products/tpt.php [04/20/2008].

[Tre08] Tretmans, J.: Model Based Testing with Labelled Transition Systems, In *Formal Methods and Testing*, Volume 4949/2008, Pages: 1 – 38, ISBN 978-3-540-78916-1, LNCS. Springer Berlin / Heidelberg, 2008.

[TVec] T-VEC Technologies, Inc., Test Vector Tester for Simulink – T-VEC, Commercial Model-Based Testing Tool, http://www.t-vec.com/solutions/simulink.php [07/03/08].

[UL06] Utting M., Legeard B. *Practical Model-Based Testing: A Tools Approach*. ISBN-13: 9780123725011. Elsevier Science & Technology Books, 2006.

[UPL06] Utting M., Pretschner A., Legeard B. *A Taxonomy of Model-Based Testing*, ISSN: 1170-487x, 2006.

[UTP] OMG: *UML 2.0 Testing Profile*. Version 1.0 formal/05-07-07. Object Management Group, 2005.

[Utt05] Utting, M.: Model-Based Testing. In *Proceedings of the Workshop on Verified Software: Theory, Tools, and Experiments,* VSTTE 2005. 2005.

[VecI] Vector Informatik GmbH, http://www.vector-worldwide.com/ [04/22/08].

[Wal01] Wallmüller E. *Software- Qualitätsmanagement in der Praxis*. ISBN-10: 3446213678. Hanser Verlag, 2001 (In German).

[WCF02] Wiesbrock, H.-W., Conrad M., Fey, I.: Pohlheim, Ein Neues Automatisiertes Auswerteverfahren für Regressions und Back-To-Back-Tests Eingebetteter Regelsysteme, In *Softwaretechnik-Trends*, Volume 22, Issue 3, Pages: 22 – 27. 2002 (In German).

[Wey88] Weyuker, E.: 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, 1988.

[WTB] Synapticad, Waveformer Lite 9.9 Test-Bench with Reactive Test Bench, Commercial Tool for Testing, http://www.actel.com/documents/reactive_tb_tutorial.pdf [04/20/08].

[WW06] Wappler, S., Wegener, J.: Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed Genetic Programming. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, Pages: 1925 – 1932, ISBN: 1-59593-186-4 Seattle. Washington, U.S.A. ACM, 2006.

[Zan07] Zander-Nowicka, J.: Reactive Testing and Test Control of Hybrid Embedded Software. In *Proceedings of the 5th Workshop on System Testing and Validation (STV 2007)*, In Conjunction with ICSSEA 2007, Editors: Garbajosa, J., Boegh, J., Rodriguez-Dapena, P., Rennoch, A., Pages: 45 – 62, ISBN: 978-3-8167-7475-4, Paris, France. Fraunhofer IRB Verlag, 2007.

[Zan07] Zander-Nowicka, J.: *Model-based Testing of Embedded Systems in the Automotive Domain*, PhD Thesis, Technical University Berlin, to appear 2008.

[ZDS+05] Zander, J., Dai, Z. R., Schieferdecker, I., Din, G.: 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, Montreal, Canada. Springer-Verlag, 2005.

[ZMS07a] Zander-Nowicka, J., Marrero Pérez, A., Schieferdecker, I.: From Functional Requirements through Test Evaluation Design to Automatic Test Data Retrieval – A Concept for Testing of Software Dedicated for Hybrid Embedded Systems. In *Proceedings of the IEEE 2007 World Congress in Computer Science, Computer Engineering, & Applied Computing; SERP 2007*, Editors: Arabnia, H. R., Reza, H., Volume II, Pages: 347 – 353, ISBN: 1-60132-019-1, Las Vegas, NV, U.S.A. CSREA Press, 2007.

[ZMS07b] Zander-Nowicka, J., Marrero Pérez, A., Schieferdecker, I., Dai, Z. R.: Test Design Patterns for Embedded Systems. In *Business Process Engineering. Conquest-Tagungsband 2007 – Proceedings of the 10th International Conference on Quality Engineering in Software Technology*, Editors: Schieferdecker, I., Goericke, S., ISBN: 3898644898, Potsdam, Germany. dpunkt.Verlag GmbH, 2007.

[ZMS08] Zander-Nowicka, J., Mosterman, J. P., Schieferdecker, I.: Quality of Test Specification By Application of Patterns, In *Proceedings of the 2nd International Workshop on Software Patterns and Quality (SPAQu 2008)*; In conjunction with 15th Conference on Pattern Languages of Programs (PLOP 2008), Co-located with OOPSLA 2008, Nashville, TN, U.S.A. 2008.

[ZSM06] Zander-Nowicka, J., Schieferdecker, I., Marrero Pérez, A.: Automotive Validation Functions for On-line Test Evaluation of Hybrid Real-Time Systems. In *Proceedings of the IEEE 41st Anniversary of The Systems Readiness Technology Conference (Autotestcon 2006)*, IEEE Catalog Number: 06ch37750c, ISBN: 1-4244-0052-X, ISSN: 1088-7725, Anaheim, CA, U.S.A. IEEE, 2006.

[ZXS08] Zander-Nowicka, J., Xiong, X., Schieferdecker, I.: Systematic Test Data Generation for Embedded Software. In *Proceedings of the IEEE 2008 World Congress In Computer Science, Computer Engineering, & Applied Computing; The 2008 International Conference on Software Engineering Research and Practice (SERP 2008)*, Editors: Arabnia H. R., Reza H., Volume I, Pages: 164 – 170, ISBN: 1-60132-086-8, Las Vegas, NV, U.S.A. CSREA Press, 2008.