AUTOMOTIVE VALIDATION FUNCTIONS FOR ON-LINE TEST EVALUATION OF HYBRID REAL-TIME SYSTEMS

Justyna Zander-Nowicka[1],
Ina Schieferdecker,
Abel Marrero Pérez
Fraunhofer Fokus, MOTION
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
{zander-nowicka,
schieferdecker,
perez}@fokus.fraunhofer.de

Abstract - The aim of this paper is to present the means of black-box on-line test evaluation for hybrid real-time systems. The described procedures can be used for the model-based testing process so as to improve its effectiveness. In particular, intelligent automotive validation functions are considered, which are divided into different types depending on the nature of the evaluated issue. All provided definitions are specified on the meta-model level according to the Model Driven Architecture (MDA) trends. The application of the concepts given in this paper contributes to the continuous on-line test analysis and is exemplified by a component of an Adaptive Cruise Control model in Matlab/Simulink/Stateflow ®.

1. INTRODUCTION

Model-based black box testing is an evolving technique for generating a suite of test cases from the system design. It helps to ensure a repeatable and scientific basis for system testing, gives good coverage of all its behaviors and allows tests to be linked directly to the requirements [21]. Model-based testing gained a lot of momentum in the last years due to its automation potential [7], [8]. The generated test designs are provided in a graphical way, executable and they include an oracle component which assigns verdicts to each test [3].

Considering testing of embedded reactive systems, the attention is put on their hybrid nature, complexity and time constraints. Thus, the test evaluation mechanisms become to be more complicated than for standard software. Such aspects like assessment of signals quality in time, duration or frequency of particular actions and catching of test purposes from the specification contribute to the evaluation of test execution results.

The way, how the test evaluation means (i.e. validation functions) presented in this work, are differentiated relates to the identification of requirements during the development of the investigated systems, where mixed continuous and discrete signals appear, time becomes to be the relevant aspect and specification coverage must be reached. However, we focus on testing of the functional requirements [10] by now.

Further on, the on-line evaluation of the results for a given black-box test campaign is reached. The presented approach defines libraries of validation functions that are able to continuously update the verdicts for a test case. Scalability of the test is supported by libraries extension mechanism. Systems may be analyzed either using the predefined validation functions or by building new ones on the given templates. Additionally, we base on the concept of "model-in-the-loop". It serves for testing at the early stages of the development line, where only system designs exist. We foresee also the need to transfer our evaluation means for application on other test levels (i.e. "hardware-in-the-loop").

1.1 Model Driven Architecture Context

Typical verification method used for testing of hybrid reactive systems is their simulation. This process is essential to verify the functionality of the system, but it is not exhaustive due to the lack

[1] This work has been partially supported by the German National Academic Foundation, http://www.studienstiftung.de/.

of particular test scenarios [14]. Applying sophisticated test mechanisms to understand the system behavior during simulation improves development effectiveness and assures a systematic test process. One of the promising testing options is to use automatically created test models based on requirements and system design [2], [7], [8], [23].
Such an automatic and systematic generation of test campaigns can be achieved by application of the model driven testing[2] approach [8], [22]. Thus, we aim to test systems along the Model Driven Architecture (MDA) [2], [16].
The approach shown in the following is based on the work described in [23], where a system meta-model representing hybrid behavior and its test meta-model are both defined as Meta Object Facility (MOF) [17] models. We embed our work in the context of MDA as we plan to enable automatic transformation of both requirements and system models into test models in the future.

1.2 Paper Outline

The paper is structured as follows. After the introduction, Section 2 gives an overview on the related work to the considered subject. Automotive validation functions are presented in Section 3. Section 4 provides examples of system and test models for one component of an adaptive cruise control in Simulink/Stateflow [12], [15]. In particular, the application and contents of the validation functions are shown. Also the functionality behind the meta-classes is depicted and algorithms are described. In Section 5, the results are discussed and conclusions are drawn. Finally, future work challenges are outlined.

2. RELATED WORK

In the following, available test methodologies, at least partially supporting on-line test evaluation are introduced.
An approach used mainly for testing temporal safety critical requirements is presented in Safety Checker Blockset [14] and EmbeddedValidator [1]. It deals with a set of properties being the combination of model's variables that are connected to a proof operator (i.e. “NeverTogether”, “NeverAfter”, “A_implies_B”). A test model typically expresses an unwanted situation (i.e. the cruise control should never be active while the driver is braking). The verification process assesses if this situation is reachable or not and on this base establishes the test outcome. Only few temporal constraints are checked and the methodology needs further detailed research.
Another option described in [20] is based on examining the values of different signals present in the system under test (SUT). The common procedures are checking whether any of the elements of the signal at its input is nonzero or does not exceed specified limits during model execution. If it is the case, an assertion arises. This method covers relatively simple issues, without considering complex features of continuous signals.
Reactis Validator [4] provides a test framework to graphically express assertions that check SUT for potential errors, and user-defined targets that monitor system behavior in order to detect the presence of certain desirable test cases. If a failure occurs a test execution sequence is delivered and it leads to the place where it happens. However, no predefined validation elements enabling a systematic test design are available.
In the research presented in [6] watchdogs' applicability is foreseen and exemplified. The watchdog is used for a single requirement and logs the violation of the respective model property in form of a Boolean output variable. Our validation functions are similar to this approach, although we aim to structure and systematize our solution.
The Time Partition Testing [11] method and its implementation use a special assessment language to write the test evaluation mechanisms and to assign the verdicts, however the process is performed offline.
The verdict concept used for assessing the test outcomes occurs in such standards like UML 2.0 Testing Profile (U2TP) [18] or Testing and Test Control Notation – version 3 (TTCN-3) [9], [13]. In TTCN-3 local verdicts contribute to a global verdict which is calculated from the local ones based on the overwriting rules. Verdicts are evaluated on the fly, however continuous signals can be generated and analyzed only to some extent by now [19]. Such a verdict assessment corresponds to our approach.
We aim to combine the test process advantages of the already mentioned approaches. We obtain a systematic method capable of dealing with reactive tests that check complex properties and requirements of automotive systems. We provide a test framework, test templates, generic, graphical validation functions and other test components collected in a Simulink library.

[2] We enhance MDA with a separate development line for testing artefacts so as to define testing in the context of MDA

3. VALIDATION FUNCTIONS MEANING

Our focus is to evaluate both discrete and continuous signals quality on-line during a single model simulation. Our approach aims to evaluate the SUT signals by predicting their identifiable features or flow. We compare them with the expected values applying sophisticated validation functions. A validation function corresponds to the commonly known watchdogs [6] or assertions with pre- and postconditions [4], but it is additionally structured in a systematic way. We deliver Automotive Validation Functions as we focus our research on the problems specific to the automotive domain. Such functions are independent of test data. They can set the verdict for all possible test data vectors and activate themselves only if some predefined conditions are fulfilled.

Another advantage of the proposed concept is to use the same development language for both – system design and test design so as to integrate testing and system modeling. We also enable easier covering of change requests throughout the development phases and finally we want to evolutionary improve both processes parallel during the project time.

Automotive Validation Functions (AVFs)[3] serve to evaluate the execution status of a *test case* by assessing the *SUT* observations and/or additional characteristics/parameters of the *SUT*.
Verdict is the assessment of the *SUT* correctness yielded by a *test case* [18]. It is used to report failures in the test system.
AVFs contribute to the overall concept of Test Behavior[4].

3.1 Structure behind the Automotive Validation Function

We distinguish different types of *AVFs* depending on the signal features that they test. Thus, we consider characteristics which:

a) address the proper signal values or any derivations (also when filtered or transformed),
b) determine a single value or dependence on a single value,
c) determine value dependence (i.e. linearity) between two signals,
d) demand temporal dependence,
e) demand a specific signal flow (i.e. continuity).

A further analysis of signals to be tested leads to the considerations about their local and global features. We define the first as testable at every simulation step, whereas the latter ones apply over a certain time/signal interval and are available after a certain simulation step.
The detailed division of signal features that we identified to be relevant by now, is introduced in Table 1.

Table 1. Division of Signals Features versus Signals Characteristics

	Local	Global
a)	- Signal values comparison (i.e. raw signal, filtered signal, transformed signal values comparison)	- Response delay
b)	- Single value dependence check (using <=,<, =, !=, >, >=)	- Detect max, detect min, detect inflection (i.e. step response maximal overshoot)
c)	- Signals functional dependence check (i.e. if increase detected then decrease expected), - Linearity detection	
d)		- Signal value at a certain time check (i.e. consumption, emission) - Time check by a single value = time stamp check (i.e. step response settling time) - Time interval between two events (i.e. step response rise time)
e)	- Signal specific flow recognition (i.e. detect decrease, detect increase, detect constant, detect a predefined function flow), - Continuity check - Continuous derivative check	

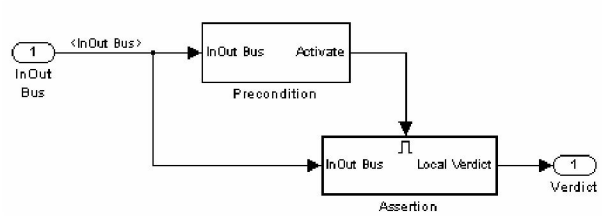
By creating the *AVFs* we consider requirements as well as system models to be our source of information. We split a requirement using a set of generic rules first:

$$IF \quad precondition\_1 \quad AND.... \quad AND \quad precondition\_m$$
$$THEN \quad assertion\_1 \quad AND.... \quad AND \quad assertion\_n,$$

[3] Words written in cursive represent the *meta-classes* of the test meta-models introduced in [23].
[4] We define Test Behavior as the interactions between and inside Test Data, SUT and Validation Functions.

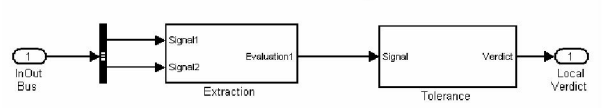
An *AVF* represents such a single elementary rule. The system model delivers the signals for observation and control.
An abstract template for a Validation Function (shown in Figure 1) consists of a preconditions block which activates the assertion block where the comparison of actual- and expected-signal values happens.

Figure 1. Structure of *AVF* – a Template



Preconditions and assertions blocks are built following the schema shown in Figure 2.

Figure 2. Preconditions or Assertions Blocks Contents – Template



They include an extraction part and a tolerance checking block. In the extraction part a proper precondition or assertion is formulated whereas in the tolerance block the permitted tolerance of the signal variation is provided. A general rule governing the tolerance check is:

$$\left|obtained\_signal(time)\right| \leq \varepsilon,$$

where:

obtained_signal(time) - is a signal delivered by the extraction block at every simulation step,
ε - is the permissible tolerance.

If testing of one requirement demands more rules combinations, we implement all of them and conclude with a verdict according to the arbitration algorithm.

3.2 Arbitration Process

AVFs update the *verdict* of a *test case*. Predefined *verdict* values are pass, fail, inconclusive, none and error. Pass indicates that the test behavior gives evidence for correctness of the functional aspects of the *SUT* for that specific test case. Fail describes that the purpose of the test case has been violated. An error *verdict* shall be used to indicate errors (exceptions) within the test system itself. Inconclusive is used for cases where neither a pass nor a fail can be given. When the preconditions of the *AVF* are not fulfilled or before another verdict can be set a none *verdict* is delivered.
The *verdict* of a *test case* is calculated by the *arbiter*. *Arbiter* is a property of a test case to evaluate test results and to assign the overall *verdict* of a test case [18]. We provide a special Simulink/Stateflow arbitration block to play the role of an arbiter in our approach.
During test execution, each *AVF* responsible for *verdict* assignment, continuously reports *verdicts* to the *arbiter.* The *arbiter* produces the final *verdict* from those intermediate *verdicts.*
There is a default arbitration algorithm. It is used while delivering the verdict for a single requirement evaluation or for the entire test case. V*erdicts* are ordered according to the rule: none < pass < inconclusive < fail < error. A meta-class of an *arbiter* contains method for updating the final verdict.

4. EXAMPLE ON THE BASE OF SIMULINK AUTOMOTIVE MODELS

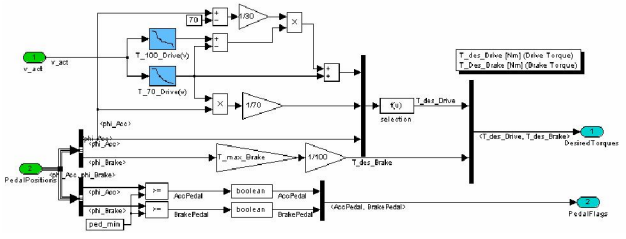
The Adaptive Cruise Control (ACC) [5], [6] system controls the speed of the vehicle while maintaining a safe distance from the preceding vehicle. The activated system monitors the road in front of the vehicle. If there is no vehicle ahead (so called target), the system regulates the vehicle speed. As soon as a preceding vehicle is recognized, the distance control function is activated which makes sure that the vehicle follows the vehicle in front at a safe distance.

In this paper, a simplified component PedalInterpretation (Figure 3) of the ACC is used for illustration purposes. This subsystem can be employed as a preprocessing component for various vehicle control systems. The pedal interpretation subsystem interprets the current, normalized positions of accelerator and brake pedal (phi_Acc, phi_Brake) by using the actual vehicle speed (v_act) as desired torques for driving and brake (T_des_Drive, T_des_Brake) [5], [6]. Furthermore, two flags (AccPedal, BrakePedal) are calculated which indicate whether or not the pedals are considered to be pressed. The software requirements are summarized in Table 2 [5], [6].

Table 2. Software Requirements Specification for PedalInterpretation (excerpt) [5], [6]

ID	Description
SRPI-01	**Recognition of pedal activation** If the accelerator or brake pedal is depressed more than a certain threshold value, this is indicated with a pedal-specific binary signal.
SRPI-01.1	**Recognition of brake pedal activation** If the brake pedal is depressed more than a threshold value ped_min, the BrakePedal flag should be set to the value 1, otherwise to 0.
SRPI-01.2	**Recognition of accelerator pedal activation** If the accelerator pedal is depressed more than a threshold value ped_min, the AccPedal flag should be set to the value 1, otherwise to 0.
SRPI-02	**Interpretation of pedal positions** Normalized pedal positions for the accelerator and brake pedal should be interpreted as desired torques. This should take both comfort and consumption aspects into account.
SRPI-02.1	**Interpretation of brake pedal position** Normalized brake pedal position should be interpreted as desired brake torque T_des_Brake [Nm]. The desired brake torque is determined when actual pedal position is set to maximal brake torque T_max_Brake.
SRPI-02.2	**Interpretation of accelerator pedal position** Normalized accelerator pedal position should be interpreted as desired driving torque T_des_Drive [Nm]. The desired driving torque is scaled in the non-negative range of velocity in such a way that the higher the velocity is given, the lower driving torque is obtained.* * A direct interpretation of pedal position as motor torque would cause the undesired jump of engine torque while changing the gear maintaining the same pedal position.

Figure 3. PedalInterpretation Design [5], [6]

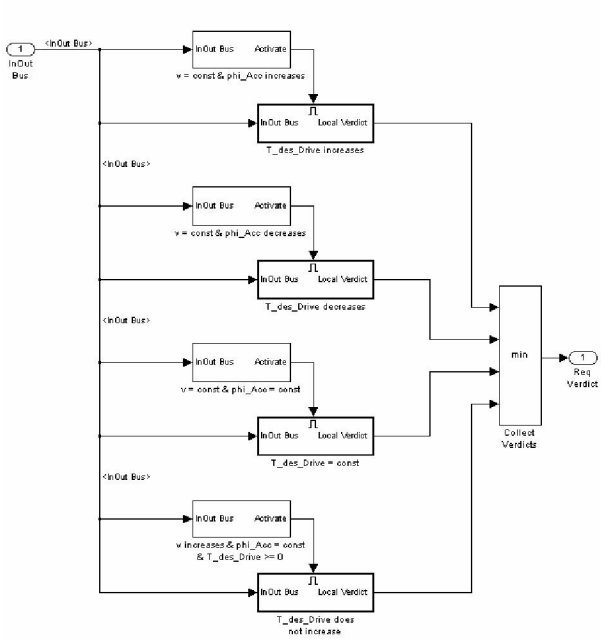


4.1 Functionality behind the Automotive Validation Functions

In the following simplified excerpts of *AVF* blocks are introduced. *AVFs* analysis is performed starting with SRPI-02.2 requirement and going up through SRPI-02.1 and SRPI-01.1(3) where complexity is proportional to the number of the requirement. This proceeding order will illustrate how the test design looks like.

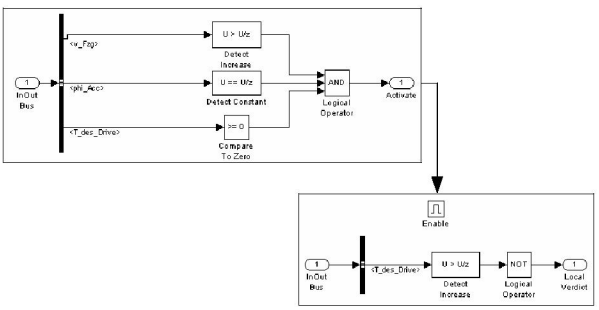
Concerning the requirement SRPI-02.2 the *AVFs* are designed and shown in Figure 4. Apart from the actual feature check, we deal here with preconditions that must be assured so as to activate the proper assertion.

Figure 4. Evaluation of Requirement SRPI-02.2



Let us analyse the second part of the SRPI-02.2 requirement in details. The following feature must be checked: If one quantity increases under certain conditions, the other one decreases (Figure 5). Hence, firstly the preconditions are assured. If the desired driving torque is non-negative, an increase in velocity is detected and the acceleration pedal position is constant then the feature can be evaluated. When the expected decrease is detected, pass verdict is delivered, otherwise fail appears.

Figure 5. *AVF* for the Selected Part of Requirement SRPI-02.2



The next requirement of our concern is to check if the dependency of two quantities proceeds linearly with an assumption that the line is crossing certain points. In our example, the requirement SRPI-02.1 demands a linear dependence of brake pedal position to brake torque and crossing of (0,0) and (100, T_max_Brake) points. This feature belongs to the category *detect a predefined function flow*. Thus, the following approach decides about the verdict.

The slope of a linear function is equal to:

$$slope = \frac{p_A}{T_A}, \quad equivalently \quad slope = \frac{p_B}{T_B},$$

where:
p_N - pedal position at time N
T_N - Torque_des_Drive at time N

Hence,

$$\frac{T_B}{T_A} = \frac{p_B}{p_A}, \quad thus \quad \frac{T_{max}}{T_A} = \frac{100}{p_A},$$

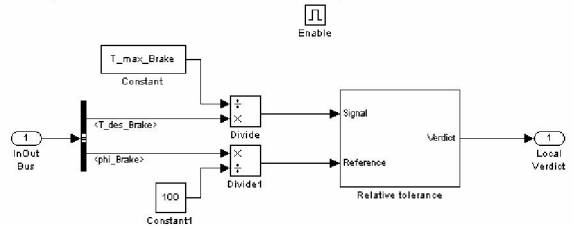
where:
T_{max} - T_max_Brake

Additionally, tolerance value of the signal linearity deviation should be considered. In our case we assume 5 % of T_{max}/T_A in the positive and negative direction as tolerance range.

$$-0.05 \times \frac{T_{\max}}{T_A} \geq \left(\frac{T_{\max}}{T_A} - \frac{100}{p_A} \right) \leq 0.05 \times \frac{T_{\max}}{T_A}$$

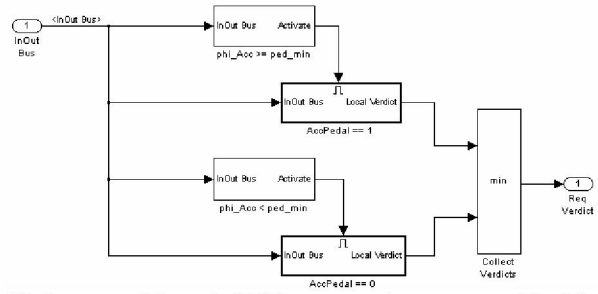
All the mentioned equations are designed using Simulink/Stateflow in the assertion block shown in Figure 6. This requirement must be fulfilled during the entire test execution, thus there are no preconditions.

Figure 6. Assertion for Requirement SRPI-02.1



Finally, SRPI-01.2 and SRPI-01.4 requirements are tested. We verify the functionality of PedalInterpretation subsystem where brake and drive pedals flags are calculated. We deal here with value-discrete signals at the *SUT* output. The evaluation of the pedal positions recognizes a pedal as pressed only if it is activated above a certain threshold value, called ped_min. This statement forms the precondition blocks contents of the *AVF*. Assertion blocks assume that the flag is activated if the pedal position is equal or higher than ped_min, whereas it is deactivated for pedal position lower than ped_min. The situation is depicted in Figure 7.

Figure 7. Evaluation of Requirement SRPI-01.2



All the considered *AVFs* concepts are provided in a Simulink library created for testing purpose. Simulink libraries are collections of blocks that can be used in models. Blocks copied from a library remain linked to their originals such that changes in the originals automatically propagate to the copies in a model. Libraries ensure that the models automatically include the most recent versions of previously defined blocks and give fast access to the commonly used functions. The testing blockset contains *AVFs* and other elements needed to generate Simulink/Stateflow test models. At the moment, tests are generated manually. Automation of this process is aimed, however this is beyond the scope of this paper.
The presented *AVFs* are created for component tests, however they may be re-used without any changes also for integration or system tests as some issues must be evaluated on comprehensive levels. This is an advantage of our solution.

5. CONCLUSIONS

This work presents the splitting of the validation process for hybrid real-time systems. Different types of *AVFs* are applied for test result assessment. The nature of the evaluated behavior establishes the way of test methods division. *Continuous* and *discrete* aspects are explicitly provided so as to express hybrid nature of the *SUT*. In particular, we show examples of checking the dependencies between two quantities and value-discrete signals analysis. In this paper, we demonstrated our research with Matlab/Simulink/Stateflow models. The approach itself, however, is independent of the modeling language. The essence is the notion of “model-in-the-loop” simulation.

We consider the character of requirements and SUT in- and output signals to be the deciding factor for the way of test means definition. Work related to the evaluation and arbitration mechanism is shortly mentioned. We present different types of *AVFs* and *verdicts*. The feasibility of the approach has been depicted on the base of the PedalInterpretation test models in Simulink. Functionality and algorithms hidden behind the concepts have been elaborated.

In the future work the definition of *AVFs* must be completed. We will analyze further commonly known signals and their features to provide more systematic *AVFs* patterns.
In our approach we focus on *AVFs* without analyzing test data generation, that is why our further aim is to combine the validation functions with intelligent data selection and thus to provide abstract test scenarios. We want to support a general fault model to handle detected failures by using default or self-adaptation mechanisms. Report generation for black-box tests is partially available, where duration, frequency, location or reasons of particular failures are explicitly named.
Finally, the quality of the requirements (also in our example) is often not high enough to let directly apply the appropriate *AVFs*. Thus, we consider the specification driven tests for further investigation.
Last but not least, an advantage would be the automatic generation of test models from system model and/or requirements and their execution in a common framework. Thus, the transformation rules between system and test development lines have to be considered.

REFERENCES

1. Bienmüller T., Brockmeyer U., Sandmann G., Automatic Validation of Simulink/ Stateflow Models, Formal Verification of Safety-Critical Requirements, 2004, Stuttgart
2. Born M., Schieferdecker I., Kath O., Hirai C., Combining System Development and System Test in a Model-centric Approach, RISE 2004, Luxembourg.
3. Broy M., Jonsson B., Katoen J.-P., Leucker M., Pretschner A. (Eds.), Model-Based Testing of Reactive Systems, Advanced Lectures, Springer 2005, ISBN 3-540-26278-4
4. Cleaveland R., Hansel D., Sims S., Smolka S., Reactis Validator, Embedded Software Design Automation, Reactive Systems, Inc.
5. Conrad M., A Systematic Approach to Testing Automotive Control Software, Detroit (US), Oct. 2004, SAE Technical Paper Series 2004-21-0039
6. Conrad M., Modell-basierter Test eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien. Dissertation, Deutscher Universitätsverlag, Wiesbaden (D), 2004
7. Dai Z. R., Model-Driven Testing with UML 2.0, Proceedings of Second European Workshop on Model Driven Architecture (MDA) EWMDA, September 2004, Canterbury, England
8. Dai Z. R., Grabowski J., Neukirchen H., Pals H., From Design to Test - Applied to a Roaming Algorithm for Bluetooth Devices. Next Generation Testing for Next Generation Networks, St Anne's College, Oxford, UK, TestCom 2004
9. ETSI ES 201 873-1 V2.2.1: The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, 2003
10. Glossary of terms used in Software Testing, December, 2004, International Software Testing Qualification Board
11. Lehmann E.: Time Partition Testing, Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen, Promotion, Technischen Universität Berlin, November 2003
12. Matlab, version 7, R14 , Mathworks, <http://www.mathworks.com/products/matlab/>
13. Neukirchen, H. W., Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests, Dissertation, Goettingen 2004
14. Safety Checker Blockset, TNI-Software, <http://www.vnvware.com/?ss=scb&type=overview>
15. Simulink/Stateflow, Mathworks, <http://www.mathworks.com/products/simulink/>
16. OMG: Model-Driven Architecture (MDA), <http://www.omg.org/docs/omg/03-06-01.pdf>
17. OMG: Meta-Object Facility (MOF), version 1.4, <http://www.omg.org/technology/documents/formal/mof.htm>
18. OMG: UML 2.0 Testing Profile. Final Adopted Specification, ptc/04-04-02, 2004
19. Schieferdecker I., Bringmann E., Grossmann J., Continuous TTCN-3: Testing of Embedded Control Systems, SEAS 2006, Shanghai, China, ISBN:1-59593-402-2
20. Using Model Verification Blocks, <http://www.mathworks.com/access/helpdesk/help/toolbox/slvnv/ug/f9093dfi4.html>
21. Utting M., Model-Based Testing, The University of Waikato, New Zealand, Verified Software: Theories, Tools, Experiments, VSTTE 2005
22. Zander J., Dai Z. R., Schieferdecker I., Din G., From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing, Canada, Montreal, TestCom 2005
23. Zander-Nowicka J., Schieferdecker I., Farkas T., Derivation of Executable Test Models From Embedded System Models using Model Driven Architecture Artefacts - Automotive Domain -, Dagstuhl, Germany, MBEES 2006