Derivation of Executable Test Models from Embedded System Models Using Model Driven Architecture Artefacts - Automotive Domain -

Justyna Zander-Nowicka[1], Ina Schieferdecker[1,2], Tibor Farkas[1]

[1]Fraunhofer FOKUS, MOTION
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany

[2]Technical University Berlin,
Faculty IV, Straße des 17. Juni 135
10623 Berlin, Germany

{zander-nowicka, schieferdecker, farkas}@fokus.fraunhofer.de

Abstract: The approach towards system engineering compliant to Model-Driven Architecture (MDA) implies an increased need for research on the automation of the model-based test generation. This applies especially to embedded real-time system development where safety critical requirements must be met by a system. The following paper presents a methodology to derive basic Simulink test models from Simulink system models so as to execute them in the same framework as the system model. The meta-models for Simulink and Test are explained and will enable automatic transformation in the future. The testing concepts of UML 2.0 Testing Profile (U2TP) have been partially adopted, however also enriched so as to deal with continuous functions, real-time constraints and to mirror Simulink-specific features.

1 Introduction

Embedded systems development needs modelling, simulation and analysis of dynamic behaviour. One of the tools enabling these functionalities is Simulink [MatML] - being often used in the automotive industry. It supports linear and nonlinear systems, modelled in continuous time, sampled time or a hybrid of the two. Simulation means the execution of the system model. It enables the prediction of the behaviour of the system from a set of parameters and initial conditions. Testing, on the other side is a process of identifying the completeness and quality of developed software. It demands the definitions of system under test (SUT), test objectives, environmental constraints and test criteria. In the early phases of the V-Model [VML97] testing means also execution of the test model. This leads to a conclusion that simulation provided by Simulink, may be involved in testing, however test definition and test artefacts (i.e. timer actions, clocks, actions assigning the verdict from the outside) must be additionally provided. The tool offers a set of model verification blocks which enable to state if the system is build right, according to the requirements. Test harness can be modelled only to some extent, thus the motivation of the following work is to extend Simulink with additional libraries, features and facilities to let the generated test models be executed. The aim is to provide automotive industry with an integrated environment for modelling and executing system as well as for modelling and executing tests in one common framework. Another motivation is the Model-Driven Architecture (MDA) and its artefacts which can be adopted for testing. As MDA gained much momentum in industry, the focus is to use

these concepts so as to show that retrieving executable test instances from system model can be supported via either manual or even automatic test model generation.
The paper is divided into five sections. After the introduction, Section 2 is devoted to the Simulink, Simulink Test, Environment Test Directive and Test Directive meta-models which should be provided behind the tools. Section 3 provides an example of retrieving the executable tests, which is possible by applying some restrictions during modelling. In Section 4 work related to test generation from system models for embedded systems in automotive domain is considered. In Section 5, the results are depicted and conclusions are drawn. Finally, future work challenges are outlined.

2 Meta-models Behind the System and Test Designs

MDA prescribes a set of model artefacts to be used along system development, how those models may be prepared and their relationships [MDA03]. It is an approach to system development that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform. Meta Object Facility (MOF) [MOF04] together with MDA has shown that meta-model based language definitions impose new ways of defining language semantics as a separation of syntax and semantic concept space, of integrating languages via a common meta-model base and of generating and deriving models from other models via model transformers [SD04].
This paper presents an approach to test embedded systems along the MDA-based paths. Simulink and Test Simulink meta-models are both defined as MOF models. Test Directive and Environment Test Directive meta-models are created in the same manner so as to enhance automatic test models generation in the future work. Those additional meta-models serve as sources of information about test requirements and safety-critical requirements, respectively. Details concerning the meta-models are explained in the next Section. Transformation rules defined for the example in this paper are applied only manually. They define relations between source and target meta-classes of given meta-models [ZDS+05]. The whole approach is shown in Figure 1.

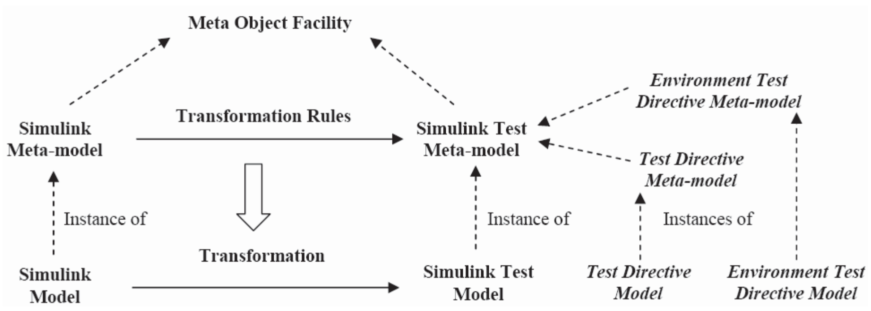

Figure 1: Retrieval of Simulink Test Models from Simulink Models Using the MDA Approach

Due to lack of explicit tool supporting MOF notation, the UML notation has been deliberately used to visualize selected meta-models.

2.1 Development of the Simulink Meta-Model

A proposal for Simulink meta-model (SL meta-model) has been developed by researchers from Vanderbilt University [NK04]. Although some insights of the core have been adopted, the following approach is slightly changed, extended and reflects more elements of the hierarchical, actor-oriented [Lee04] models. The meta-model defines the Simulink concept space with additional support for the graphical presentation format. It does not directly reflect the structure of Simulink models, but rather gives the semantics overview. It is defined in different packages with concept structures for Model, System, Block, Annotation, Line and Port, Parameter, Presentation, etc.
In this paper, only excerpts from the meta-model are presented because the main focus is put on the integration of system and test designs. The principal building blocks of system specifications in Simulink are models. Every model contains a single system. A system may also contain blocks that are used for modelling the behaviour, lines for blocks' connection or annotations used to add user's information. Some of these blocks may be typed by the subsystem. A subsystem can be built in the same way as a block. Every subsystem contains a single system [NK04], which enables hierarchical representation of complex embedded systems.

2.2 Development of the Simulink Test Meta-Model

The Simulink Test meta-model (SLTest meta-model) is developed from scratch, however the experience gained from UML 2.0 Testing Profile [U2TP04] has been used. It defines the testing concept space with special attention put on the Simulink-specific features. It does not directly reflect the structure of Simulink Test models, but rather gives the semantics overview. It is provided in different packages with concept structures for Test Architecture – describing test architecture, Test Behaviour – defining specific aspects of the test behaviour, Clock for Test – dealing with time constraints, test case timer actions and clocks, finally Simulation Test Data – for supplying continuous test data. It re-uses also some concepts provided within the SL meta-model (i.e. Model meta-class or blocks of type Model Verification). The root meta-class is the test model inheriting from and applying to the model meta-class. The test architecture concepts are related to the organization and realization of a set of related test cases. These include test context meta-class, which consist of one or more related test cases. The verdict of a test case is assigned by an arbiter. There is a default arbitration algorithm ordering the verdicts according to the hierarchy of their types. Similarly, test behaviour concepts describe the behaviour of the test cases that are defined within a test context. Associated with test cases are test objectives, which describe the capabilities the test case is supposed to validate. Test cases consist of behaviour, which includes validation actions. Validation actions update the verdict of a test case. They are created as user-defined blocks with Matlab functions behind them yet. Log actions, which write information about the tests are represented by an attribute (logSignalData) of a connection line called signal. Behavioural concepts also include the verdicts that are used to define the test case outcomes.
Simulink Test meta-model depends on the SL meta-model, while two additional meta-models, described in the next subsection, depend on the SLTest meta-model.

2.3 Additional Meta-models Overview

The additional Test Directive (TestDirective) and Environment Test Directive (EnvTestDirective) meta-models govern information gained from the test requirements and safety-critical requirements, respectively. TestDirective models are used only to refine the transformations from system model to test model according to the requirements or adding some implicit statements enhancing the mapping rules. Whereas EnvTestDirective models provide data on situations, when critical variables exceed/don't reach their defined values, some dangerous data ranges are achieved, unexpected situations or additional interference actions from the environment occur. These additional data must be treated as test data and enrich/refine the tests appropriately.
The proposed EnvTestDirective meta-model is developed using safety-critical requirements analysis. It gives the semantics overview and is defined in different packages with concept structures for EnvTestDirModel, Environment Noise, Critical Data, etc. It re-uses, similarly to the other test meta-models given in this paper, concepts provided within the SL meta-model (i.e. Model meta-class), but also SLTest meta-model (i.e. package containing Simulation Data).

3 Transformation on the Base of an Example

In this section, the example of transformations from a simple system model to test models representing executable test cases is depicted. Additionally, a model providing interferences from the environment is introduced.
The test objectives are connected with the definition of the functionality belonging to the validation actions. These are: checking of the quality of signals (continuous and discrete), looking for timeouts and matching of the test requirements with test model elements.
Our example model is restricted to two levels hierarchy. More complicated designs (including closed loops models) might cause additional problems or demand constraints for the transformation rules. It is also assumed that an *environment test directive model* is previously delivered as a Simulink design.
Some of the general transformation rules on the meta-models level used for the example are given below and explained after that:

```
pre: SystemModel::Element(n)→ TestModel::Element(n)
1.SystemModel::Subsystem→ TestModel::TestComponent
  || SystemModel::Subsystem→ TestModel::SUT
2.SystemModel::Number(Subsystem)→ TestModel::Number(TestModel)
3.SystemModel::Number(Subsystem)→ TestModel::Number(TestContext)
       if (!SUT) {
4.SystemModel::Block.Type(Inport)→TestModel::Block.Type(UserDefinedFun
  ctions.StartPathTestCaseTimer)
  && TestModel::PathTestCaseTimer.Value==[t(expectedExecution)+t(delay)]
5.SystemModel::Block.Type(Outport)→ TestModel::Block.Type(UserDefinedFun
  ctions.StopPathTestCaseTimer)
6.SystemModel::Block.Type(Outport)→ TestModel::Block.Type(UserDefinedFun
  ctions.ValidationAction)
7.SystemModel::Block.Type(Outport)→ TestModel::Block.Type(Sinks.Scope)
8.SystemModel::EnvTDModel→TestModel::Block.Type(UserDefinedFunctions.
  ValidationAction)}
```

3.1 Transformations from System Model to Test Models

Figure 2 presents a simple Simulink system *model*[1], which is to be converted into a set of *test models*. Continuous signal no. 1 coming from the sensor is flowing through a *line* from the *block* of type *Inport* to the *block* typed by *Demux*. Then the signal is split going to the *block* typed by *Integrator* on the one side and to the *block* typed by *Gain* on the other side. After that the Integrator's activity starts and the resulting signal becomes to be continuous signal no. 2 and is lead using the *line* connection to the *block* typed by a *subsystem*. The functions called inside the subsystem give continuous signal no. 3, which is introduced into the *block* typed by *Outport* – a link to the potential actuator. A similar procedure applies to the other paths in the model.

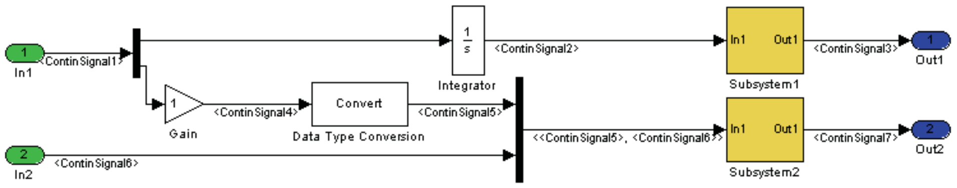

Figure 2: Simulink System *Model*

Let us consider a testing scenario of the given *model*. Firstly the structure of the system *model* is adopted to the *test model* `(rule pre)`. Afterwards, test architecture artefacts are added. In Figure 3 Simulink *test model* derived from the system is shown. It deals with the situation when Subsystem1 is the *SUT* `(rule 1)`. In another scenario the Subsystem2 could be treated as *SUT*, letting all the other *subsystems* on this level be a *test component*. Each *subsystem* which is not empty is the potential candidate to be the *SUT*. That is why the *test contexts* define all the possible combinations of *SUT* and *test components* with the rotating *SUT* role assignment. *Test contexts* have no explicit notation and each separate *test context* is expressed as a separate *test model* `(rule 3)`. Hence, there are two instances of such *test models* for the considered example `(rule 2)`. The next transformation step is to add the other test artefacts expressing behaviour to the *test model*. Two types of *test cases* are distinguished: *block test case* and *path test case*. Additionally, *path test case timer actions*, *clocks* – giving the simulation time and *validation actions* are attached.
Time measurement is used to assure proper termination of *test cases* [DS04] as well as appropriate duration of *block* execution. In order to assure a proper termination of a *path test case*, a *path test case timer* is started at the beginning of a *path test case*. Its duration is chosen to be slightly longer ($t_{PTcT} = t_{expected\ execution} + t_{delay}$) then the expected execution time of the *test case* `(rule 4, 5)`. At the end of each possible test sequence, the considered *timer* is stopped. A *verdict* value expressing the incorrect behaviour is generated when the *path test case timer* expires. For each path describing model behaviour in the diagram of Figure 3 a timer is started at the beginning and stopped at the end of it. Simulink doesn't support such *path timer actions*, thus additional user-defined blocks delivering those functionalities must be still created. Further on, *validation action* is applied. *Validation actions* should update the *verdict* of a *test case*.

[1] Words written using cursive represent the *meta-classes* from the given meta-models, respectively.

They are created as user-defined *blocks*. There are two kinds of *validation actions – path validation action and block validation action. Path validation action* applies to the *path test case*. It checks for timing out of the timer and updates the *verdict* according to the other *verdicts* established on the path `(rule 6, 7, 8)`.

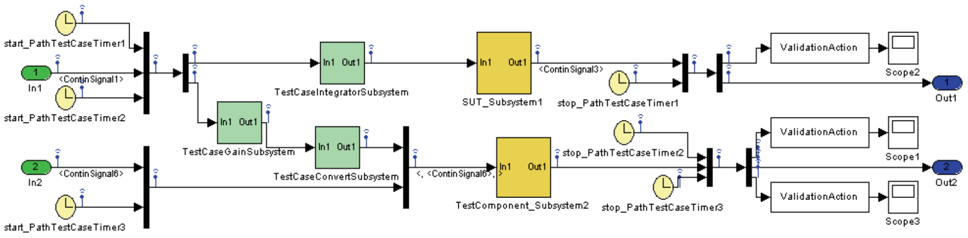

Figure 3: Derived *Simulink Test Model* for Subsystem1 being the *SUT*

Following our transformation, all the *blocks* present at the *model* or *subsystem* level in the considered system model example are handled. For such a particular block, a new subsystem in the *test model* is created so as to establish a so called *block test case*.
In every reactive system, the response time of an event should be restricted by a timer [DS04]. For this purpose, a *clock* is assigned to each *block* to measure its execution time. If the response is received within its expected duration, the *verdict* is set to the value expressing correct behaviour. Otherwise an alternative behaviour, i.e. *default* behaviour, is activated. Figure 4 presents insights of a *block test case* for Integrator subsystem. Herein a *clock* measuring the simulation time is attached to the *block* of type *Integrator*.

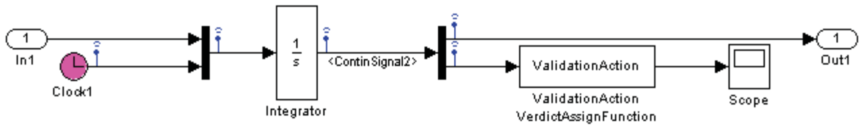

Figure 4: Derived *Block Test Case* for Integrator Encapsulated in a *Subsystem*

Block validation action is provided at the end of each *block test case*. Additionally to the *block test case*, a *reference model for block test case* is build. It includes almost all the elements as the former *test case* and is encapsulated in a *subsystem*, however in a separate *model* (see Figure 5).

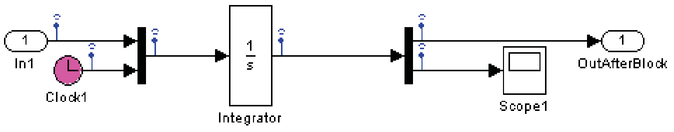

Figure 5: Derived *Subsystem* Put in *Reference Model for Block Test Case*

Block validation action should compare dynamically those two *subsystems'* outputs *(block test case subsystem* and *subsystem*, which is put in the *reference model for block test case)* after their simulation. *Reference model for block test case* is executed without any interferences, while *block test case* is executed having applied *EnvTestDirective model*. The results of the considered *block validation action* can be additionally observed by the application of the *Scope* block.

Reference model for block test case (Figure 5) for the considered *block test case* (Figure 4) has similar structure as the former one. However during simulation no interference data from the environment, no delays and no additional data from the requirements are provided. This enables to get both the pure and the real behaviour of the considered test case, so as to compare them and conclude about the verdict.
Simulation input data are needed for the functions continuum dynamic comparison. Data are derived from equivalence classes of their types and additional values given in the specification. This idea bases on CTM/ES method [CFS04] discussed in the next sections and is not concerned for the example yet.
The *block* typed by *validation action* must be intelligent enough to assure that some slight discrepancies between the pure and the real results are permissible on the one side, however safety critical requirements mustn't be ignored on the other side. Requirements are delivered by application of *EnvTestDirective model*, which enables to distinguish between soft and hard real-time requirements.
Additionally to the validation actions, *verdict* types and their hierarchy should be discussed. One of the proposals is to use fit/misfit for the arbitration of test purposes coming from the system requirements, pass/fail for the functional aspects, in-time/out-of-time for checking the duration and continuous-in-time/discrete-in-time for the time continuous aspects. Moreover, the interactions between the *verdict* types must be also considered in the future.

3.2 Additional Environmental Test Directive Models Derived from the Specification

Let us imagine an automatic control unit being activated in a given system. The safety-critical requirement specifies the following: *Always when any action from the environment (implied by a user or caused by other external action (i.e. external behaviour or changed value of some variable)) occurs, the control unit should be switched off/deactivated*.
The situation is modelled in Figure 6. On the meta-model level all the elements are put in the *EnvTestDirective model* meta-class, which inherits directly from SL *model* meta-class. The external action is represented by a *block* of type *Inport* (for SL meta-model), but also by *critical action* (for EnvTestDirective meta-model). *Inport* is connected via a *line* carrying the signal value with another *block* of type *Relational Operator*. *Block* of type *Constant* delivers the 0.0 value additionally to the action so as to let the operator compare the signal value with 0.0. If the external action value is bigger then 0.0, a *block* of type *trigger* is activated shutting off the control unit (a *subsystem*). Comparison of the two values and the result are encapsulated by *if* meta-class. The resulting action becomes to be the *body*. Finally, the deactivation of the control unit is a *critical action.*

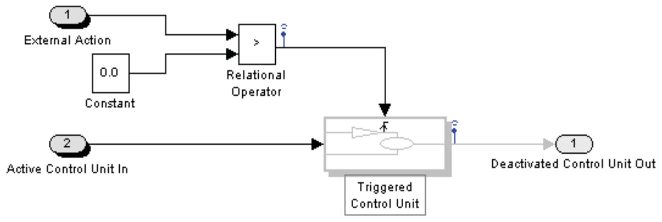

Figure 6: *Environment Test Directive Model*

This example shows that application of an *EnvTestDirective model* on an existing SL *model, system* or *subsystem* implies the *validation action* to be sensitive on the changes coming from the system outside.
As already mentioned, the transformation rules for the example in this paper are applied only manually. Due to missing technology, the rules implementation is not applicable for the large scale projects yet. However, further investigations on the subject promise that the presented solution may raise a lot of attention in the middle future.

4 Related Work

Several efforts have been undertaken to establish an approach to model-based test generation for embedded systems. Research and industrial work is being continuously developed. Algorithms have been defined to derive tests from formal system specification given in various notations.
One of the approaches is Time Partitioning Testing (TPT) [Leh00]. It is an approach for testing the dynamic functional behaviour of embedded systems. It supports the selection and documentation of test data on the semantic basis of so-called testlets and the syntactic techniques Direct Definition, Time Partitioning and Data Partitioning which are used to build testlets. Testlets facilitate an exact description of test data and guarantee the automation of test execution and test evaluation. The concepts of TPT correspond mainly to Environment Test Directive models specified using the system and test requirements in the approach presented in this paper.
Data Partitioning is strictly related to the next tool, called MTest [CFS04] and its Classification Tree Method for Embedded Systems (CTM/ES). This method contributes to testing of embedded systems very much. Using an interface description, which can be based on the specification and/or an executable model of the software, test scenarios can be derived systematically and described in a graphical way so as to provide the user with visual information about test coverage [Con04]. In our work, the data derivation from the equivalence classes of their types is provided with Simulink Test models, whereas catching of the interferences from the environment is given by EnvTestDirective models. Additionally, intermediate data flowing along the path are also treated as interfaces between different components (*subsystems*).
A similar approach to CTM/ES test data retrieval is realised by Reactis tool [Reactis]. The tests may be run on the models themselves to study and revise model behaviour. They may be applied to source-code implementations of models to ensure conformance with model behaviour. Also back-to-back tests are supported.
Matlab Automated Testing Tool (MATT) [Hen00] enables to create custom test data for model simulations or executables. Here basic functional tests on model level may be created and executed. Those issues are covered by EnvTestDirective models given in this paper.
Such tools, like EmbeddedValidator [BBS04] are considered for the discussed approach, as they validate and verify the model. The model checking technology is automatic and complete in a mathematical sense, meaning that it can detect every logical design flaw and error in the model being validated. However, it is assumed that model checkers are used before test generation.

MLIB/MTRACE and ControlDesk [dSpac99] tools make it possible to monitor, display and change the variables while the simulation is running. This is a good technological hint, how to apply the EnvTest Directive models onto SL test models.
Moreover, Simulink Test meta-model designed for embedded systems has been inspired by two factors: existence of UML 2.0 Testing Profile [U2TP04] for object-oriented software development as well as by MDA-driven testing [ZDS$^+$05]. They are both related to this research.

5 Outlook and Future Work

The aim of this paper is to demonstrate that it is worth to apply MDA artefacts for Simulink models and their testing. For this purpose Simulink, Simulink Test, Environment Test Directive and Test Directive meta-models are introduced and presented in brief. An example of retrieving the executable tests from Simulink models is provided. Some simple transformation rules are depicted. Corresponding meta-classes are assigned to the elements of the models. Further on, work related to the subject regarding automotive domain is given. Finally, conclusions are taken and further work challenges are outlined.
All the meta-models need a lot of improvement. Their structure is still in the development phase in the ongoing research projects. The tools adapters must be built to be able to interpret the Simulink models and store them in the repository at the one side, as well as to retrieve the generated Simulink Test models from the repository and expose them correctly in Simulink at the other side. Derivation of executable test models is possible by applying some modelling restrictions. They should be collected in the future. Further transformation rules have to be formalised and implemented. Some tools [IKV], [EMF] already enable transformations between models on the meta-model level. One of the environments being often used to demonstrate the feasibility of transformations is Eclipse [Eclipse] with its Eclipse Modelling Framework (EMF) plug-in. The meta-models can be defined by Rational Rose tool in UML. The EMF generator can create a corresponding set of Java implementation classes from such a Rose model. The mapping rules can be realised via EMF Java API. The transformations can generate objects within a Simulink Test meta-model, which could enable the execution of the tests directly in Simulink. The transformation rules could be also formalized using Query/View/Transformation (QVT) [QVT04] language. However the tools, like Borland Together [BT06] or Tefkat [SL04], both enabling QVT, are still very limited and incomplete to perform such formalism.
Continuing research will focus also on formal models of computation [LN04] found behind the presented models to let define formally-proven transformation rules for some complicated cases. Moreover, tool-independent ideas for meta-modelling of hybrid real-time systems will be considered. This would give a generic view on the development and testing in this area, taking into account real-time, continuous signals. Any other tools like SCADE [BDK05] used for embedded software development could be enriched with such generic test automation ideas. Additionally, requirements-driven automatic test information retrieval in the context of EnvTestDirective models must be investigated (i.e. by help of temporal logic). Also a facility to manage the obtained test results should be provided in the finishing phase of the work.

Last, but not least the autonomic communication paradigms could contribute to the self-organisation of the system model enabling the automatic generation of the executable test models. Simulink model elements could interpret themselves and re-build them so as to achieve the expected goals. Furthermore, test-driven improvements of the system under test may be established. For example, when the variable value is reaching a critical range, the system would organise itself to avoid the dangerous value by usage of some intelligent rules, that have to be defined.

References

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

[BDK05] Bouali A., Dion B., Konishi K., Using Formal Verification in Real-time Embedded Software Development, JSAE 2005

[BT06] Borland Together 2006 - http://www.borland.com/downloads/download_together.html

[CFS04] Conrad M., Fey I., Sadeghipour S., Systematic Model-Based Testing of Embedded Control Software: The MB3T Approach, Edinburgh, May 25th, 2004, ICSE 2004

[Con04] Conrad M., A Systematic Approach to Testing Automotive Control Software, Convergence Transportation Electronics Association 2004

[dSpac99] MLIB, http://www.dspace.de/ftp/patches/NFMG/SfC32/NewFeaturesAndMigration.pdf

[DS04] Dai Z.R., Schieferdecker I.: Time Concepts for UML 2.0 Based Testing. SIVOES 2004, RTAS 2004, Toronto, Canada, May 2004

[Eclipse] Eclipse Platform: http://www.eclipse.org/platform/

[EMF] Eclipse Modelling Framework: http://www.eclipse.org/emf/

[Hen00] Henry J., Automation: The Key to Improving Testing in the Maintenance of Real-time Systems, Submission of a Position Paper for WESS 2000, University of Montana

[IKV] IKV++ Technologies, medini MM tool, http://www.ikv.de/

[Lee04] Lee E. A., Actor-Oriented Design: A focus on domain-specific languages for embedded systems, MEMOCODE'2004, San Diego, California, 2004

[Leh00] Lehmann E., Time Partition Testing: A Method for Testing Dynamic Functional Behaviour, TEST2000, London, UK, 2000

[LN04] Lee E. A., Neuendorffer S; Concurrent Models of Computation for Embedded Software, TM UCB/ERL M04/26, University of California, Berkeley, CA 94720, July 22, 2004

[MatML] Mathworks, Matlab/Simulink/Stateflow, http://www.mathworks.com/products/matlab/

[MDA03] OMG: Model-Driven Architecture (MDA), http://www.omg.org/docs/omg/03-06-01.pdf

[MOF04] OMG: MOF, v1.4, http://www.omg.org/technology/documents/formal/mof.htm

[NK04] Neema S., Karsai G., Embedded Control Systems Language for Distributed Processing (ECSL-DP), Vanderbilt University, ISIS-04-505, 2003-2004

[QVT04] OMG: MOF Query/Views/Transformations, 2nd Revised Submission, ad/04-01-06

[Reactis] Reactis Tester Tool, http://www.reactive-systems.com/tester.msp

[SD04] Schieferdecker I., Din G.: A meta-model for TTCN-3. 1st International Workshop on Integration of Testing Methodologies, ITM 2004, Toledo, Spain, Oct. 2004.

[SL04] Steel J., Lawley M., Model-Based Test Driven Development of the Tefkat Model-Transformation Engine, ISSRE 2004, pp. 151-160

[U2TP04] OMG: UML 2.0 Testing Profile. Final Adopted Specification, ptc/04-04-02, 2004

[VML97] IABG: Das V-Modell-Entwicklungsstandard für IT-Systeme des Bundes, Vorgehensmodell, Kurzbeschreibung, 1997, http://www.v-modell.iabg.de/

[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, Canada, Montreal, TestCom’05