

Test Methods and Tools in Model-Based Function Development

**Dr.-Ing. Dipl.-Math. Klaus Lamberg,
Dipl.-Math. Michael Beine
dSPACE GmbH**

ASIM , Fachtagung Simulations- und Testmethoden für Software in Fahrzeugsystemen,
1./2. März 2005, Berlin, Germany.



Test Methods and Tools in Model-Based Function Development

Abstract:

There can be no doubt that electronics, and most particularly the software they contain, are the key to innovative and marketable functionality in modern vehicles. However, a high level of reliability, safety, and quality in vehicle electronics is vital. There are currently two very important trends in automotive software development:

- Controllers and functions are developed with the aid of simulation models. The C code needed for each specific target system is generated automatically on the basis of the models. Automatic production code generation has become a firm part of production development.
- As software grows in complexity, and development cycles speed up, the software that is produced must be tested as early and as exhaustively as possible. Testing is now seen as a key element in quality assurance.

Model-based development methods are shifting tests from code level to model level. In the earlier development phases especially, there is still great potential for complementing largely experimental development procedures with methods of model-based, automated testing. Typical test activities in model-based function development are model-in-the-loop, software-in-the-loop, and processor-in-the-loop simulation, with back-to-back methods based on them; black-box approaches such as the classification tree method; white-box methods with coverage measurement at model and code level; and formal verification.

Combining model-based development with automated testing provides a powerful and efficient collection of tools and methods, from test project management to systematic test case generation, right through to automated test evaluation and final test report generation. This paper describes the processes involved and the methods that underlie them in the overall context of testing.

Function Development with Automatic Production Code Generation

Software development for electronic control units (ECUs) is increasingly being performed with model-based development tools such as MATLAB®/ Simulink®/Stateflow® [(1)]. Once the specification has been produced with the aid of these tools, coding work can begin. Functions described in graphical form must be converted into C code and implemented on the ECU. Automatic production code generators are increasingly being used for this task. Today's code generation methods are now so mature compared with the first generations of code generators that they are well able to bridge the gap between model-based function design and production software development. Production code generators like TargetLink [(2)] generate highly efficient, reliable, production-quality C code from a block diagram. They not only reduce development time, they also enhance the quality of the software and avoid human programming errors.

Currently, automatic production code generators are primarily being used for new functional software components, in other words, control functions, in electronic control units (ECUs). The hardware-related software and the software infrastructure are handcoded. As the bulk of ECU software consists of function code, using automatic production code generators has rapidly grown in importance. TargetLink, for example, is now breaking through into widespread use in rollouts for major development projects.

As the focus of development activities shifts from code level to model level, followed by automatic production code generation, testing at model level

becomes essential. There is currently a clear trend towards applying the methods and approaches of classic software testing to model-based development.

Basic Aspects of Testing

Testing is a quality assurance and quality enhancement activity performed with the aim of finding errors. Testing is easy to learn and can be applied to even very complex systems. Although testing can be fairly expensive, the cost-benefit ratio is usually very positive.

Testing is performed for validation or verification, depending on the test objective. Validation relates to the requirements established for the system under development. The requirements can be functional, i.e., relating to the actual purpose of the system, or nonfunctional, i.e., relating to performance, real time, memory consumption, etc.

Verification means testing a test object against its specification. The specification describes the technical requirements that the object has to meet; these generally relate to the interfaces and to how the object must behave at the interfaces. The test object can be on any system level (unit, module,

component, subsystem), depending on the level to which the specification applies and at which it is tested. Verification itself does not prove that the test object complies with the original system requirements, as the specification itself may be faulty.

There is a wide variety of methods available for testing, which can be classified according to the scheme shown in Figure 1.

There is a basic difference between static and dynamic testing. In dynamic testing, the test object is executed, while in static testing, it is not. Formal verification is a further method class. The following discussion focuses primarily on dynamic testing and formal verification, as these are the approaches mainly used in model-based development.

Functional Testing

In functional testing, i.e., tests on the test object's functional behavior, the test object is usually stimulated by specified signal behaviors (test data), and its response behavior is captured and evaluated with reference to the desired behavior (expected values). Information on the internal structure of the test object is neither required nor used.

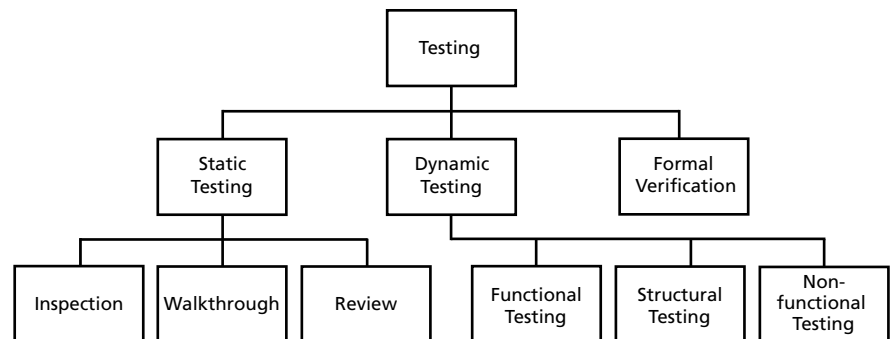


Figure 1 Classification of test methods

Functional testing is therefore also called black-box testing. Equivalence class tests (classification tree method) and back-to-back tests are examples of the use of functional testing.

The Classification Tree Method

The classification tree method is a black-box method in which the input variables of a test object, for example, a TargetLink subsystem that requires testing, are classified according to specific, relevant aspects. The aim of variable partitioning is to select individual classes in such a way that they behave uniformly with regard to detecting potential errors [(3)]. In other words, the test object behaves either correctly or incorrectly for all the values in one class ("uniformity hypothesis"). The stepwise partitioning of the input data is shown graphically in the form of a tree. Finally, the resultant classes are combined with one another as required, and test sequences are compiled from them.

Figure 2 shows a classification tree. The name of the test object (VDC) is the root of the tree, and the input signals define the classifications under the root node. Below that are the equivalence classes. It is now possible to define test sequences based on this partitioning of the input space.

The test scenarios themselves are formed by systematically combining the individual classes: Each row in the combination table describes one step in a test sequence.

DaimlerChrysler AG developed the classification tree method as long ago as the early 1990s. The method was subsequently adapted to the requirements of the model-based development process by integration into MTest [(2)].

Model-in-the-Loop, Software-in-the-Loop, Processor-in-the-Loop

Systematic use of the simulation facilities available in the development environment used enables developers to perform fast, simple checks on the results obtained and on the modifications and adjustments that have been made, during the development process.

Different simulation modes involve simulating the function model, the generated code on the development PC, and the generated code on the evaluation board. This means that testing can be performed stepwise (Figure 3).

- Simulating the function model as an executable specification is generally known as model-in-the-loop (MIL) simulation.

- To simulate the generated code on the development PC in what is known as software-in-the-loop (SIL) simulation, the generated code is translated by a host compiler.

- Simulating the generated code on an evaluation board, which will typically contain the processor used in the ECU, is known as processor-in-the-loop (PIL) simulation. The generated code is translated by the appropriate target compiler.

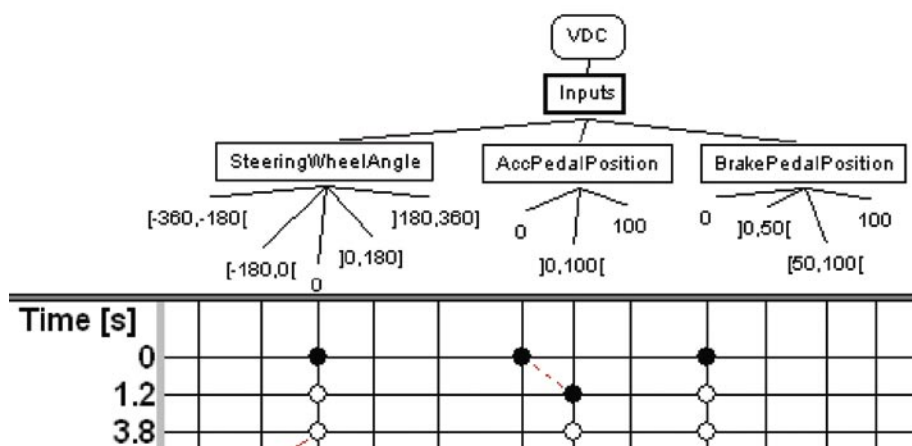


Figure 2 Classification tree for a test object called "VDC"

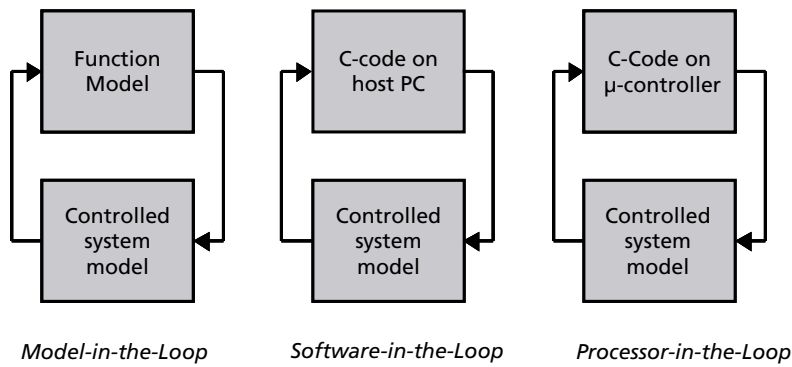


Figure 3: Principles of model-, software-, and processor-in-the-loop simulation

Back-to-Back Tests

Back-to-back tests compare the results of different simulation modes with one another. For example, the same test data is used to stimulate both the Simulink model and the program code. The output behavior of the program code in relation to the test data can then be directly compared with that of the executable specification. As can be clearly seen, the simulation results of the different modes match each other closely; visually, at any rate, they are hardly distinguishable. Figure 4 The right-hand side of Figure 4, however, shows the difference signal between the TargetLink MIL and the TargetLink SIL simulation that was calculated during the test run. It

shows a deviation in the range up to -3 for output signal uPI. The deviation increases in amount along the time axis. Deviations like this can be caused by fixed-point quantization effects, as is the case here.

There are various criteria available for evaluating this difference in terms of passing/failing the test. One frequent method is to determine whether the difference signal is within a defined

Structural Testing

Unlike functional testing, structural testing involves information on the internal structure and properties of the system under test and uses that

information to develop test cases. Structural testing is therefore also called white-box testing. For example, when the internal structure of a system is known, it is possible to define targeted test cases that run through specific structural elements in the test object, such as statements and decisions. This allows the stimulation and testing of targeted parts of the test object.

Model and Code Coverage

In the other direction, it is possible to determine the frequency with which specific system parts are run through during test execution. Performing measurements such as this at model level is called model coverage, and at program code level, it is called code coverage. While Simulink [(1)] provides various coverage metrics at model level (decision coverage, condition coverage, modified condition decision coverage, look-up table coverage), TargetLink [(2)] supports the measurement of statement coverage (C1) and of decision coverage (C2) at code level during the simulation or test run. This means, for example, that parts of the model or program that are never executed, despite a large number of tests, can be identified. Further test cases can then

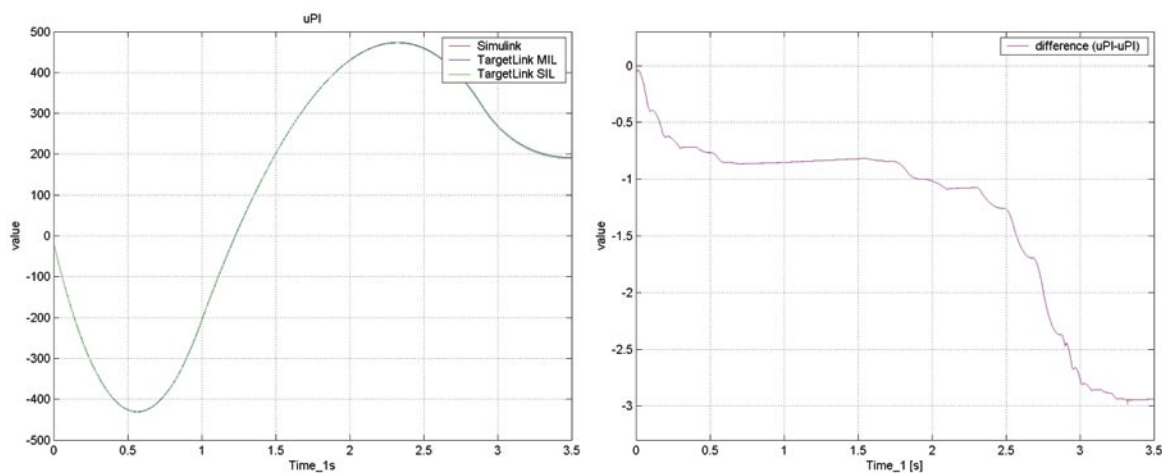


Figure 4: Comparison of different simulation modes

be defined specifically for these parts. In individual cases, specific parts of the model or code may even be removed, if they are unreachable and therefore superfluous (dead code).

Formal Verification

While validation and verification as described above generate specific stimulation situations to test specific system properties, formal verification is used to check the system’s properties as defined. For example, a system can be proven correct, in other words, it meets requirements, in a mathematical sense. What distinguishes this method from dynamic testing is its completeness and consistency.

There are two basic approaches to formal verification: model checking and theorem proving. These differ, among other things, in the extent to which automation is possible, the prior knowledge that is required, and the type of verification model used. EmbeddedValidator is a tool [(5)] that uses model checking for the formal, automated verification of models in embedded systems in Simulink, Stateflow, and TargetLink. EmbeddedValidator tests a model against a formal description of requirements, such as “Error condition A is never reached”, or against more complex requirements that include causality and that can be described, for example, by temporal logic,

in practice, formal verification is only rarely used in the development of automotive software. Its main application lies in developing safety-critical software.

Methods and Tools

Table 1 contains an overview of the tools used for testing in function development and the methods they use. MTest [(7)] is a platform for systematic and automated testing in function development with MATLAB/Simulink/Stateflow and TargetLink. It has an integrated classification tree method that can be used for test development. A further feature is that test vectors can easily be imported from other sources and applied to the model under test. MTest supports testing throughout Simulink simulation, and also model-in-the-loop, software-in-the-loop, and processor-in-the-loop simulation.

These approaches are more black box in nature.

They can be extended in the direction of white-box testing by connecting Simulink model coverage and TargetLink code coverage functionality to MTest. Automatic test vector generators can also be integrated on this basis.

Integrating TargetLink with EmbeddedValidator and combining them with MTest in this way adds to the range of available tools by providing support for formal verification by model checking.

Tool	Methods Used
MTest / CTE	Equivalence class test, threshold analysis, requirements-based testing, black-box testing
Model- / software- / processor-in-the-loop	Black-box-testing, back-to-back-tests
Simulink® model coverage	Structural tests (white-box testing))
TargetLink code coverage	Structural tests (white-box testing)
Embedded Validator	Formal verification

Table 1: Methods supported by the tools

Test Management

Even single functions require a large number of tests. Test runs can take anything from a few minutes for single tests up to many hours for whole test series, for example, overnight or on weekends. With such a large number of tests, plus test data and test results, an efficient test management system is vital [(6)]. The purpose of test management is to provide uniform and structured access to all the numerous tests and to their associated data, models, documents, and results.

Figure 5 shows a test project in MTest. All the elements in the project are displayed and organized hierarchically. The most important elements in the project tree are explained in Table 2 along with their meanings.

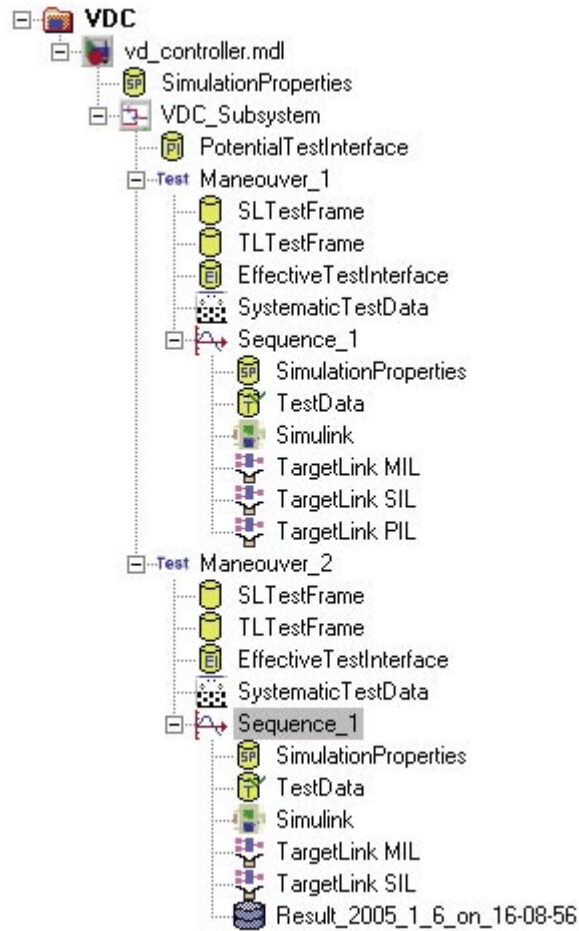


Figure 5: Test project in MTest

Symbol	Description / Function
VDC	Test project VDC
vd_controller.mdl	Represents the Simulink files with the test object
VDC_Subsystem	Simulink or TargetLink subsystem as a test object
-Test Maneuver 1	Test for the VDC_Subsystem test object"
SLTestFrame TLTestFrame	Automatically generated test frame in Simulink or TargetLink
SystematicTestData	Classification tree file (CTE)
Sequence_1	Test sequence described by the classification tree method
Simulink TargetLink MIL TargetLink SIL TargetLink PIL	Different execution modes for the Sequence_1 test sequence
Result_2005_1_6_on_16-08-56	Result object

Table 2: Elements of the MTest project tree

Finally, the test results can be used to generate test reports in various formats and with varying levels of detail, for permanent documentation of test results - for example, as a basis for release decisions.

Figure 6 shows the results display in MTest on the left and a corresponding test report on the right.

The screenshot shows the MTest interface. On the left is a tree view of test results under the execution name 'Maneuver_2'. The tree includes folders for SLTestFrame, TLTestFrame, EffectiveTestInterface, SystematicTestData, CumulativeModelCoverage, CumulativeCodeCoverageSIL, and Sequence_1. Under Sequence_1, there are folders for SimulationProperties, Simulink, and TargetLink MIL. Each folder contains sub-items like UsedTestData, OutputData, EvalResults, and ModelCoverage.

On the right is a table with two columns: 'Name' and 'Value'. The table lists the following items:

Name	Value
SLTestFrame	
TLTestFrame	
EffectiveTestInterf	
SystematicTestDa	
CumulativeModelC	
CumulativeCodeC	
Sequence_1	

Figure 6: Result display and automatically generated report in MTest

The screenshot shows a web browser window titled 'AutomationDesk Report - Microsoft Internet Explorer provided by dSPACE'. The address bar shows the URL: 'stemManeuver_2R@Result_2005_1_6_on_17-15-12V@Result_2005_1_6_on_17-15-12(1)HTML/index.html'. The browser displays a report for 'Sequence_1'.

The report includes a tree view on the left and a main content area with the following sections:

Description :

SimulationProperties

Simulation parameter	Value
SolverType	Fixed-step
Solver	ode1 (Euler)
StartTime	0.0
StopTime	3.5
FixedStep	0.001
SolverMode	SingleTasking
OutputOption	RefineOutputTimes
Refine	1

TargetLink

Simulation parameter	Value
SolverType	Fixed-step
Solver	ode1 (Euler)
StartTime	0.0
StopTime	3.5
FixedStep	0.001

Conclusion

The testing of software and ECUs plays a key role in the development of automotive electronics. Automated testing has now established its position in the later development phases, especially for testing ECUs by means of HIL simulation. In the earlier development phases, however, there is still great potential for supplementing largely experimental development procedures with methods of model-based, automated testing. This combination of model-based development and automated testing provides a powerful and efficient collection of tools and methods, from test project management to systematic test case generation, right through to automated test evaluation and final

test report generation.

Testing always relates to the requirements imposed on the object under test. Thus, testing can be successful only if the requirements are known and formulated, and if test specifications and development procedures take them into account. Otherwise, successful testing is purely a matter of luck. This is underlined by the fact that up to 40% of ECU errors that are found are caused by inadequate and incomplete or unclear specifications [(8)]. For this reason, testing will come to be seen less and less as the last, tedious step in the development chain, but instead as an integral component of the entire development process.

Literature

- [(1)] *The MathWorks: Simulink®/Stateflow® product information, 2004.*
- [(2)] *dSPACE GmbH: product information on TargetLink, MTest, AutomationDesk, 2004.*
- [(3)] *Grochtmann, M.; Grimm, K.: Classification Trees for Partition Testing. Software Testing, Verification and Reliability, 3, 63-82, 1993.*
- [(4)] *Wiesbrock, H.-W.; Lim, M.: Automatische Signalanalyse – Bessere Qualität durch automatisierte Testauswertung. Elektronik Automotive 2/2004.*
- [(5)] *OSC Embedded Systems AG: Embedded Validator Produktinformation, 2004.*
- [(6)] *Lamberg, K., Richert, J.; Rasche, R.: A New Environment for Integrated Development and Management of ECU Tests. SAE 2003-01-1024, Detroit, USA, 2003.*
- [(7)] *Lamberg, K.; Beine, M.; Conrad, M.; Eschmann, M.; Fey, I.; Otterbach, R.; Model-based testing of embedded automotive software using MTest. SAE-Paper No. 2004-01-1593, Detroit, USA, 2004.*
- [(8)] *Hanselmann, H.: Vom Modell zum Serieneocode. Electronic Automotive III/2003*



Headquarters in Germany

dSPACE GmbH
Technologiepark 25
33100 Paderborn
Tel.: +49 52 51 1638-0
Fax: +49 52 51 66529
info@dspace.de
www.dspace.de

France

dSPACE SARL
Parc Burospace
Bâtiment 17
Route de la plaine de Gisy
91573 Bièvres Cedex
Tel.: +33 1 6935 5060
Fax: +33 1 6935 5061
info@dspace.fr
www.dspace.fr

USA and Canada

dSPACE Inc.
28700 Cabot Drive · Suite 1100
Novi · MI 48377
Tel.: +1 248 567 1300
Fax: +1 248 567 0130
info@dspaceinc.com
www.dspaceinc.com

United Kingdom

dSPACE Ltd.
2nd Floor Westminster House
Spitfire Close · Ermine Business Park
Huntingdon
Cambridgeshire PE29 6XY
Tel.: +44 1480 410700
Fax: +44 1480 410701
info@dspace.ltd.uk
www.dspace.ltd.uk