Model-based Testing of Automotive Systems

Eckard Bringmann, Andreas Krämer
PikeTec GmbH, Germany
Eckard.Bringmann@PikeTec.com, Andreas.Kraemer@PikeTec.com

Abstract

In recent years the development of automotive embedded devices has changed from an electrical and mechanical engineering discipline to a combination of software and electrical/mechanical engineering. The effects of this change on development processes, methods, and tools as well as on required engineering skills were very significant and are still ongoing today.

At present there is a new trend in the automotive industry towards model-based development. Software components are no longer handwritten in C or Assembler code but modeled with MATLAB/Simulink™, Statemate, or similar tools. However, quality assurance of model-based developments, especially testing, is still poorly supported. Many development projects require creation of expensive proprietary testing solutions.

In this paper we discuss the characteristics of automotive model-based development processes, the consequences for test development and the need to reconsider testing procedures in practice. Furthermore, we introduce the test tool "TPT" which masters the complexity of model-based testing in the automotive domain. To illustrate this statement we present a small automotive case study.

TPT is based on graphical test models that are not only easy to understand but also powerful enough to express very complex, fully automated closed loop tests in real-time. TPT has been initially developed by Daimler Software Technology Research. It is already in use in many production-vehicle development projects at car manufacturers and suppliers.

1. Motivation

Within only a few years the share of software controlled innovations in the automotive industry has increased from 20 percent to 80 percent, and is still growing. Forecasts claim that software will determine more than 90% of the functionality of automotive systems within the next decade. Consequently the impact of software on the customer and hence on market shares and competition will be enormous. This establishes software as a key technology in the automotive domain.

During recent years the growth of software in the automotive industry led to a development process founded on model based technologies which have many advantages for automotive developments. Firstly model-based technologies such as MATLAB/Simulink, Statemate, MatrixX, or LabView are domain specific and support powerful mechanisms for handling and processing continuous signals and events which are the most important data types in the automotive domain. These model-based technologies allow the development of high-level models that can be used for simulation in very early stages of the development process. This in turn is important since automotive development is an interdisciplinary business with software, electrical, and mechanical engineering aspects inextricably entwined. Graphical models and simulation of such models allows engineers to find a common functional understanding early in the design phase. So, model-based development improves communication within development teams, with customers, or between car manufacturers and suppliers. It reduces time to market through component re-use and reduces costs by validating systems and software designs up front prior to implementation. Consequently models are often treated as part of the requirements specification, since the models illustrate the required functionality in an executable manner. Model-based development provides a development process from requirements to code, ensuring that the implemented systems are complete and behave as expected. Model-based development allows segregation of concerns; technical aspects such as fixed-point scaling (i.e. the transformation of floating-point algorithms to fixed-point computations), calibration data management, and the representation of signals in memory are separated from the core algorithms thereby keeping the models as lean as possible.

2. Model-based testing in practice

Along with the growing functionality and the introduction of model-based development processes, the demands on quality assurance have also increased. In terms of testing, model-based development enables system engineers to test the system in a virtual environment when they are inexpensive to fix, i.e. before the code is actually implemented or integrated on the final hardware (which is called ECU or electronic control unit). However, in practice there are just a few testing procedures that address the automotive domain-specific requirements of model-based testing sufficiently. The applied test methods and tools are often proprietary, ineffective and require significant effort and money. About 15 years ago testing embedded devices comprised primarily of four well-understood areas: (1) electromagnetic compatibility (EMC tests), (2) electrical tests (e.g. short-circuit, creeping current, stress peaks), (3) environmental tests (i.e. testing under extreme climate conditions), and (4) field tests (on proving ground or the road). There was no need for dedicated functional testing methods because the functional complexity was comparatively low. With the increasing popularity of model-based development the engineering discipline of automotive model-based testing has been neglected for a long time. With the promise of model-based development to prove concepts at early development stages by means of executable models it was assumed that testing those models and the derived code is less important and therefore would not require new techniques.

3. Requirements for automotive MBT

The term ‘model-based testing’ is widely used today with slightly different meanings. In the automotive industry it is normally used to describe all testing activities in the context of model-based development projects.

Automotive model-based development has specific characteristics that require the use of dedicated model-based testing approaches. The characteristics of, and the resulting requirements for model-based testing are discussed below.

3.1 Test automation

Automotive systems usually interact with a real-world environment which is under continual control of the embedded system. Thus the whole system is not only a piece of software, but a complex construction that consists of software, hardware, electrical, mechanical, and/or hydraulic parts. The development of such a system requires the co-design of software and hardware components. Consequently an iterative process is needed with a considerable number of interim releases of the integrated system. Thoroughly testing these interim releases is crucial to ensure that requirements inconsistencies and design or implementation faults can be uncovered as early as possible. This means that the same tests have to be repeated again and again over the development cycle. Test automation is therefore essential as the manual test workload over many iterations would be at best expensive and likely not practical, leading to less than thorough quality assurance standards.

Test automation also simplifies the coordination between car manufacturers and suppliers during development. Every ECU sample delivered by the supplier during development must fulfill at least a predefined acceptance test before being integrated in a new car setup. Finally, testing automotive systems often requires test scenarios with a very precise sequence of time-sensitive actions, especially for power-train and chassis systems (in the range of μ-sec). The only way to execute such scenarios is automation.

3.2 Portability between integration levels

One major benefit of model-based development is the possibility to “do things as early as possible”. In terms of testing this means to test the functionality in the model before the software is implemented and integrated into the final ECU. Between the initial modeling and the integration into the ECU there are intermediate integration levels described below. Since the functionality of the system should remain constant and independent of the integration level, relevant test cases should also be constant throughout the integration and implementation.

In order to maximize reuse, a test procedure for model-based developments should support the portability or reuse of test cases between the various platforms. On the one hand this reduces the effort of test case design tremendously and, on the other hand, allows for easy comparison of test results between the different integration levels which may be tested on different systems. In addition sharing test cases between different levels also means that test cases can be expressed in a common notation. Test cases can be corrected or extended in a central test model without the need to adjust a lot of different test implementations for the different integration levels.

Although this requirement sounds trivial from a theoretical point of view it is a weak point in today's testing practice for model-based development because test procedures and test languages are usually specialized for one particular test platform and are very difficult to share with other test platforms.

In general, the following integration levels are distinguished:

Model-in-the-Loop (MiL): The first integration level is based on the model of the system itself. Testing an embedded system on MiL level means that the model and its environment are simulated (interpreted) in the modeling framework without any physical hardware components. This allows testing at early stages of the development cycle. In most automotive model-based development projects there are different kinds of models. Functional models (aka physical models) are rather abstract and do not consider all aspects such as robustness and performance. In the course of the development physical models are transformed into implementation models. These models are designed to meet the requirements from a software engineering point of view. Aspects such as encapsulation, abstraction, robustness, performance, fixed-point scaling, and reuse are treated in implementation models. Implementation models are often used together with a code generator to automatically derive production code from the system models.

Both physical and implementation models can be tested. Physical models are tested on a system level (system test). When testing implementation models for complex systems it makes sense to distinguish between module tests (testing subsystems of the model that handle particular functional areas) and system tests.

Software-in-the-Loop (SiL): Testing an embedded system on SiL level means that the embedded software is tested within a simulated environment model but without any hardware (i.e. no mechanical or hydraulic components, no sensors, actuators). Usually the embedded software and the simulated environment model run on the same machine. Since the environment is virtual, a real-time environment is not necessary. Usually SiL tests are performed on Windows- or Linux-based desktop machines.

Processor-in-the-Loop (PiL): Embedded controllers are integrated in embedded devices with proprietary hardware (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. 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. Additionally acceptance tests of components delivered by the supplier are executed on the HiL level because the component itself is the integrated ECU. HiL testing requires real-time behavior of the environment model to ensure that the communication with the ECU is the same as in real application.

Test rig: Testing in a test rig means that the embedded software runs on the ECU. The environment consists of physical components (electrical, mechanical, or hydraulic). Furthermore, a test rig often utilizes special equipment for measurements and other analysis tools.

Car: The last integration level is obviously the car itself. The final ECU runs in the real car which can either be a sample or a car from the production line.

3.3 Systematic test case design

For many reasons automotive model-based developments usually consist of complex functionality. The systems interact with physical components that have a complex behavior which depends on many variables. Controlling such components also introduces complex controller functionality. Secondly the electrical, mechanical, and hydraulic components can fail and the embedded system has to detect such failures and compensate for them in a very robust way, as far as technically possible. The implementation of code to handle all such failure modes is a significant portion of the system software.

Testing such complex systems requires an astute selection of test cases to ensure that all test-relevant aspects are covered while redundancies are avoided. A test procedure must support this thorough selection process by providing a means of keeping an overview of the selected test cases, even when there are hundreds or thousands of test cases.

3.4 Readability

Automotive model-based testing is a collaborative work between testers, system engineers, and programmers. All of these experts have different perspectives on the system, different skills and experiences, and provide important information for the identification of suitable test cases: system engineers know about the possible pitfalls of the application domain, programmers know about the complexity of algorithms and their risk of faults, and testers know about coverage and combination issues, boundary tests, robustness tests and others methods that have to be considered in the test implementation.

The principle of model-based development is to use models as a "common language" comprehensible to all engineers involved in the development. Hence model-based testing should be readable and comprehensible by all of these experts too and not just by a few testing specialists.

3.5 Reactive testing / Closed loop testing

When testing model-based developments, test cases are often 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.

A simple example of a reactive test case for an engine control is the following scenario: The test case should simulate a sensor failure at the moment when the revs per minute of the engine exceed a certain critical value. In this case there is no fixed point in time at which the system is known to exceed the critical rpm value because this event is related to many parameters through a complex relationship. To model the test case in a natural way we must be able to express the dependency from the system.

Existing testing approaches support such reactive testing by means of scripting languages. Although these languages are powerful enough for reactive tests they are rather low-level concepts so that it is often difficult to easily understand such test cases.

3.6 Real-time issues and continuous signals

As mentioned previously, testing on the HiL level or on test rigs requires the environment and the test cases to run in real-time. Conventional test approaches rarely satisfy this requirement. Instead they run test scripts on a non-real-time machine and interact with the system under test without a precise timing environment.

However, real-time behavior is an important requirement to guarantee reproducible test cases. Without real-time there may be jitters in the communication – especially between continuously changing signals. Even slight deviations in the timing behavior at the interfaces can have a huge impact on the system behavior, with consequences for traceability and reproducibility.

In addition, there are test cases where the timing constraints are crucial for the systems functionality, for example in engine controls. In such systems many tests are impossible to perform without the real-time certainty of the test behavior.

3.7 Testing with continuous signals

Automotive control systems have a fairly complex functional behavior with complex interfaces and deal with continuously changing signals as input and output entities. Testing systems with continuous signals is poorly supported by conventional test methods. Existing methods are data-driven and have no means of expressing continuous signals and continuous timing issues. As a consequence of this methodical gap, the test of automotive systems in practice focuses on simple data tables to describe input signals or on script languages, such as Visual Basic, Python or Perl, to automate tests. Nevertheless, signals are still very difficult to handle in these languages.

4. TPT for automotive model-based testing

All of the requirements mentioned above must be considered when testing model-based developments in the automotive domain. As mentioned previously, most of the testing technologies lack support for the desired features. This leads to development of many proprietary testing solutions that focus on the testing problems of individual model-based development projects but do not address the overarching challenge.

Our new approach – which is called Time Partition Testing (abbreviated as TPT) – has been specifically designed to bridge this gap for model-based automotive testing. The objective of TPT is

1. to support a test modeling technique that allows the systematic selection of test cases,
2. to facilitate a precise, formal, portable, but simple representation of test cases for model-based automotive developments, and thereby
3. to provide an infrastructure for automated test execution and automated test assessments even for real-time environments. This is important for hardware-in-the-loop tests, for example.

As a general design principle, TPT test cases are independent of the underlying software architecture and technology of the SUT and the test platform. This enables test cases to be easily reused on different test platforms, such as MiL, SiL or HiL environments. This is not only a big step towards efficient testing, but also supports the direct comparison of test results for a test case that has been executed on multiple integration levels (i.e., back-to-back testing).

The TPT test method is strongly associated with a corresponding test language. An objective of the TPT test method is to support the systematic selection of relevant test cases. A language is therefore necessary to describe these selected cases in a comprehensible manner. The test language in turn affects the way test cases are modeled, compared, and selected. Therefore the language also affects the test method. Due to this strong relationship between method and language we will discuss both aspects in this section together.

The systematic, well-directed and well-considered selection of test cases is crucial for improving the test efficiency. Redundancies and missing test relevant scenarios can only be identified by viewing the set of test cases as a whole. Consequently a test method and a test language must always answer two questions:

1. How can a single test case be described using the test language?
2. How does the test method support the selection of the whole set of test cases for a SUT? In other words, how does it contribute to the avoidance of redundancies between test cases and to the identification of missing test cases?

Amazingly existing test approaches that support automated tests usually avoid the second question and only provide sophisticated languages which allow the definition of complex, fully automated test scenarios.

TPT tries to go one step further. TPT provides a language for modeling executable tests for automotive systems together with a systematic test case selection procedure embedded into the test modeling concept.

5. Test process using TPT

The unique feature of the Time Partition Testing is the way test cases for continuous behavior are modeled and systematically selected during the test case design activity. Nonetheless, TPT goes far beyond this activity. The overall testing process of TPT is defined as presented in Figure 1 and explained in the following.

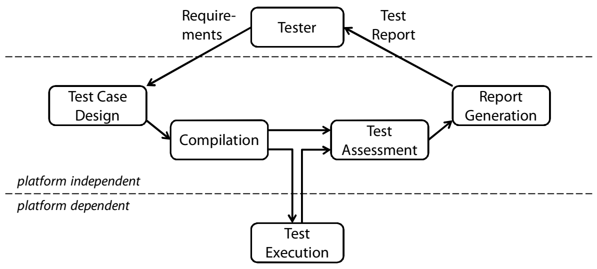

Figure 1: TPT test process

5.1 Test case design

During test case design, test cases are selected and modeled by means of the graphical test modeling language described in the next section. The basis of this test case design is the functional system requirements. So tests modeled with TPT are black-box tests.

5.2 Compiling

Test cases are compiled into highly compacted byte code representations that can be executed by a dedicated virtual machine, called the TPT-VM. The byte code has been specifically designed for TPT and contains exactly the set of operations, data types, and structures that are required to automate TPT tests. This concept ensures that test cases as well as the TPT-VM have a very small footprint. This is important in test environments with limited memory and CPU resources, such as PiL and HiL.

Additionally all assessment properties that describe the expected results of the SUT are compiled into integrated assessment scripts. For each test case there is one byte code representation and one assessment script.

5.3 Test execution

During test execution the virtual machine (TPT-VM) executes the byte code of the test cases. The TPT VM communicates continually with the SUT via platform adapters. The platform adapter is also responsible for recording all signals during the test run. Due to the clear separation between test modeling and test execution, tests can run on different platforms such as MiL, SiL, PiL, and HiL environments. HiL environments (which usually run in real-time) can be automated with TPT tests because the TPT-VM is able to run in real-time too. The clear and abstract semantic model of TPT test cases allows the test execution on almost every test environment provided that a corresponding platform adapter exists.

5.4 Test assessment

The recorded test data is initially just raw signal data without any evaluation of whether the behavior of the SUT was as expected or not. This data is then automatically assessed by means of the compiled assessment scripts. Since test assessments are performed off-line, real-time constraints are irrelevant. Currently TPT uses Python as the script language so that an existing Python interpreter can be used as the runtime engine. A powerful library has been provided to simplify signal handling, signal observation and signal manipulation. However, TPT does not rely on the actual scripting language or on the interpreter.

5.5 Report generation

A report is generated from the results of the test assessment. It depicts the result of the test case in a human-readable format. For this purpose the report contains the test result (with one of the verdicts passed/success, failed, or unknown), curves of relevant signals, data tables as well as customizable comments that illustrate the evaluated results.

TPT supports all major test activities and automates as many of these steps as possible. Other activities such as test management, test coverage measurement, data logging, diagnosis handling and others are not covered by TPT. However, integration with many of-the-shelf management tools such as HP Quality Center, Telelogic Doors do exist or are currently under development.

6. Case study explaining TPT

The TPT test language is best explained by means of a case study which is a simplified version of an exterior headlight controller (EHLC) of an existing production-vehicle ECU.

An outline of the specification can be stated as follows:

[#1] There is a switch with three states: ON, OFF, and AUTO.
[#2] When switch == ON the headlights shall be turned on.
[#3] When switch == OFF the headlights shall be turned off.
[#4] When switch == AUTO the headlights shall be turned on or off depending on the ambient light around the car (acquired by a light sensor with range 0% … 100%).
[#5] In AUTO mode, to avoid flickering headlights turning headlights on and off shall be controlled by means of a hysteresis curve and a debounce time (see #7 and #8).
[#6] If switch == AUTO when the system starts, the headlights shall be turned on/off if ambient light is below/above 70%.
[#7] In AUTO mode, the headlights shall be turned on if the ambient light around the car falls below 60% for at least 2 seconds.
[#8] In AUTO mode, the headlights shall be turned off if the ambient light around the car exceeds 70% for at least 3 seconds.

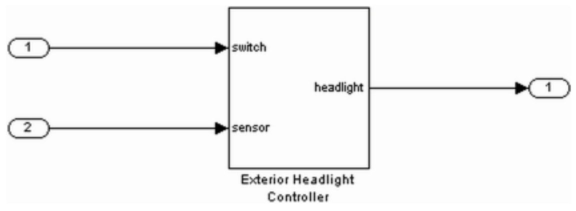

Figure 2: Top-level model of EHLC

The system is developed using model-based development by means of a MATLAB/Simulink model. The top-level subsystem of this model is shown in Figure 2.

To test the EHLC system we start with a simple test case: The test case lasts for 17 seconds and starts with the switch in the OFF position. After 2 seconds the switch is turned to the ON position and remains there for 10 seconds before being turned back to the OFF position. TPT uses a graphical state machine notation to model such a scenario as shown in Figure 3.

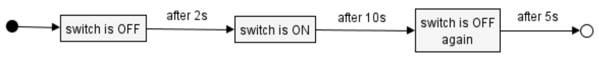

Figure 3: First test case OFF-ON-OFF

This simple graphical notation has the same meaning as the textual description above, but is easier to interpret for complex test scenarios and provides a more formal way to describe the procedure. More formal semantics are needed for test automation. TPT assigns simple equations to the states and temporal predicates to the transitions (see Figure 4). Usually these formulas are hidden behind the graphics.

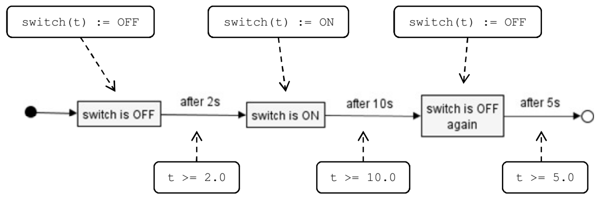

Figure 4: First test case with formal statements

Each state semantically describes how input signals of the EHLC system should be stimulated. Signals can be constants or continuously changing over time. By means of transitions and their temporal conditions the behavior of the state machine switches from one state to the next as soon as the condition is fulfilled. In the simple test case the conditions depend on time alone.

With these formulas the first test case is almost complete. The only missing information is how to define the sensor input. Even if the signal should not affect the behavior of the SUT, a definition is necessary in order to verify if the sensor value indeed does not affect the behavior. It should be noted that the test case is unique and reproducible only with well-defined specifications for all system inputs.

The sensor curve is independent from the switch state, thus it is modeled using a parallel automaton with just one state, as depicted in Figure 5. The specific definition describes a signal that is constant at 80%. For a more realistic scenario a noise component has been added. The syntax and semantics of the formulas in this example are not explained in detail here.

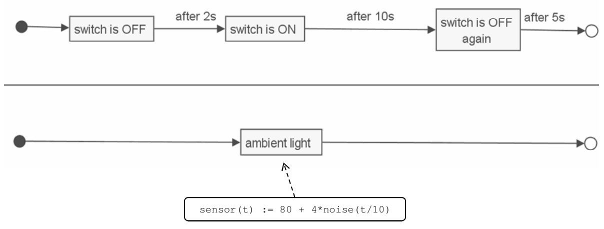

Figure 5: The complete test case

Since all inputs of the SUT are now completely defined by the test case, it can be executed deterministically. Note that the test case itself has no direct dependencies to the EHLC model, to the software implementation, or to the ECU. The test case can therefore be executed on arbitrary integration platforms if the platform is supported by TPT. Many integrations of TPT already exist on MiL, SiL, PiL, and HiL level.

Note that in general there are test cases that rely on a particular platform because they require access to signals that are not available on all levels.

During the test execution all interface signals are recorded, i.e. the input signals and the output signal of the SUT. The corresponding curves can be seen in Figure 6.

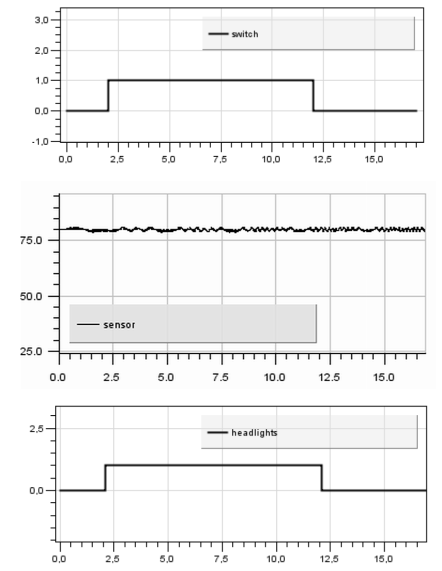

Figure 6: Test data recorded during execution

In the example the test case and the SUT behave as expected. Since the switch is never in the AUTO position, the sensor signal does not affect the behavior of the headlights at all. Headlights are turned on and off synchronously with the switch state.

Although this example of a test case is simple, it demonstrates the basic idea of the test language as a combination of graphical and formal techniques to describe readable, executable test cases that support continuous signals independently of execution platform. For practical usage there are more sophisticated techniques available such as transition branches, hierarchical state machines, junctions, actions at transitions, parameter and calibration data handling and others which are not in scope of this paper.

After modeling and executing the first test case we consider another test case: This time we want to turn the light to AUTO instead of ON for just 1 second whereas all other definitions remain unchanged. By specification the headlights should remain off due to requirement #8.

After test execution, the curves are similar to the first test case, except for the headlight signal which remains constantly at 0.

Both test cases defined so far are almost identical. Only the switch state and the transition condition in the second test case are different. Remember that the reason for defining the second test case was the requirement #8. Thus, the difference between the two test cases is related to this requirement because we must consider the aspect of how headlights behave in AUTO mode if the ambient light is bright for 1 second.

As a consequence of this consideration, differences in test cases exist to test different test-relevant functional aspects of the SUT. In order to test the functional requirements, it is therefore necessary to emphasize the differences in test cases. If test cases are modeled independently from each other, then comparisons between them are rather difficult. For that reason, all TPT test cases modeled are integrated into a single test model that allows sharing of common parts between test cases and definition of individual parts if desired.

To illustrate this concept we use the two test cases introduced above. The integrated test model is shown in Figure 7. The general structure of both test cases is identical. The second state 'turn switch' and the transition 'after …' have two alternative formal definitions causing the difference between the two cases. Elements in the model that have more than one formal definition are called variation points.

The test model itself is not executable because the semantics at the variation points are ambiguous. However, to derive an instance of a test case from this model it is sufficient to choose one of the two variants for every variation point. In the example the selection leads to the two test cases we already introduced above.

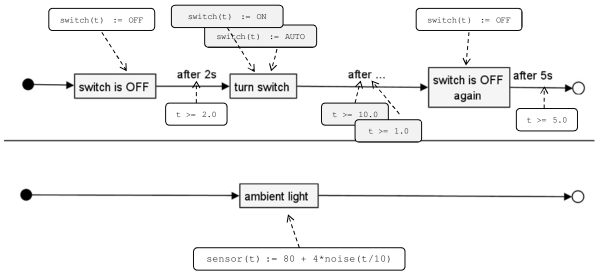

Figure 7: The integrated test model

For more realistic test problems usually there are more than 10 or 20 variation points with many variants. Thus the combinatorial complexity increases tremendously to millions of possible combinations. To keep such large models comprehensible, TPT utilizes the idea of the classification tree method [1, 4].

In the context of TPT the classifications in a classification tree represent the variation points, whereas the classes represent the corresponding variants. The combination table contains the definition of all selected test cases. Each line in the table specifies the selected variants by means of placing a mark on the line where it crosses the selected variant. The tree that corresponds to our examples is depicted in Figure 8 below.

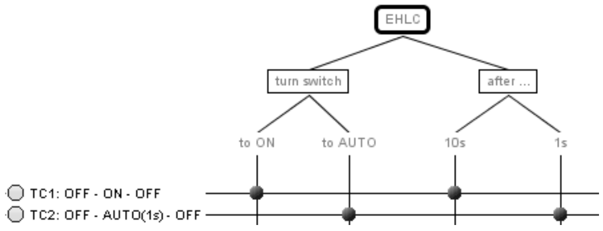

Figure 8: Classification tree as an alternative view

The classification tree and the corresponding combination table can be automatically generated for each test model. The tree representation is therefore just an alternative view of the test model that focuses on the combinatorics whereas the state machine view focuses on the timing aspect and on the variation points. Both views can be used in parallel.

The classification tree method has some sophisticated techniques to express logical constraints between classes (variants) as well as to express combinatorial rules describing the desired coverage in the space of all possible combinations. The classification tree tool CTE XL automates the generation of test cases based on these constraints and rules [4]. For complex test problems this automation is a crucial factor in reduction of effort.

The complete case study example of the EHLC controller is too complex to describe here. For a thorough test the case study defined 72 test cases with various ambient light signals and switch scenarios. With these 72 test cases it can be verified if the system meets all details of the hysteresis and timing requirements.

7. Practical experience

TPT has been initially developed at Daimler Software Technology Research and established in cooperation with many production-vehicle development projects. It is already in use in many production-vehicle development projects at car manufacturers and suppliers. As an example, all current interior production-vehicle projects at Daimler (which are all model-based incidentally) use TPT as the central testing technology for their projects. TPT is used by car manufacturers and suppliers to specify, exchange and agree on test models as the basis for acceptance tests.

TPT test cases can run on different platforms without modification. This fact has been proven in many projects where hundreds of test cases designed for MiL tests could be executed in SiL and HiL environment without any modification. This increases not only the test efficiency but also the maintenance of tests since test cases will not become outdated.

The TPT method is scalable. Even for large testing problems the idea to concentrate all test sequences into a single test model has been proven to be a good way to keep even large test sets comprehensible and to support testers in finding weaknesses within the tests.

8. Summary and outlook

Model-based development caused a radical change in automotive system development which is still in progress today. The need to test as early as possible; to test on multiple integration levels; under real-time constraints; with functional complexity; provide interdisciplinary exchangeability; facilitate continuous signal handling and many others make high demands on testing procedures, techniques, methods and tools.

TPT is one test approach that facilitates the design, execution, assessment, and report generation of test cases for automotive model-based systems. TPT test cases are portable, i.e. reusable on different test platforms such as MiL, SiL, or HiL. TPT supports reactive tests and can be executed in real-time. The graphical language for test case design is easy to understand but precise enough for test automation.

However, there is still a long way to a fully integrated and feature rich testing technique for model-based testing in the automotive domain. Particularly the interaction with test management, version and configuration management, and the issue of product families is still to be tackled.

Furthermore, model-based testing on integration levels for automotive systems is a completely unknown quantity today and has a high potential for improvement. Central questions such as the relationship between component architecture and integration tests, utilization of architectural models as the basis for integration tests in this area are yet to be answered.

9. References

[1] M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. *Software Testing, Verification & Reliability*, 3(2):63-82, 1993.

[2] E. Bringmann and A. Krämer. Systematic testing of the continuous behavior of automotive systems, ICSE2006, in *Proceedings of the 2006 international workshop on Software engineering for automotive systems*, Shanghai, May 2006.

[3] E. Lehmann. *Time Partition Testing – Systematischer Test des kontinuierlichen Verhaltens eingebetteter Systeme*, Ph.D. Thesis, Technical University of Berlin, 2003.

[4] E. Lehmann and J. Wegener. Test Case Design by Means of the CTE XL. In *8th European International Conference on Software Testing, Analysis, and Review (EuroSTAR)*, Copenhagen, 2000.

[5] B. Lu, X. Wu, H. Figueroa, and A. Monti. A low cost realtime hardware-in-the-loop testing approach of power electronics controls. *IEEE Transactions on Industrial Electronics*, 54(2):919-931, April 2007.

[6] O. Maler, Z. Manna, and A. Pnueli. From Timed to Hybrid Systems. In *RealTime: Theory in Practice*. LNCS, pages 447-484. Springer Verlag, 1992.

[7] H. Sthamer, J. Wegener, and A. Baresel. Using Evolutionary Testing to improve Efficiency and Quality in Software Testing. In *Proc. of the 2nd Asia-Pacific Conference on Software Testing Analysis & Review*. Melbourne, 2002.

[8] X. Wu, S. Lentijo, and A. Monti. A novel interface for power-hardware-in-the-loop simulation. In *IEEE Workshop on Computers in Power Electronics*, 2004.

[9] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 1996.