Testing the Continuous Behavior of Embedded Systems

Eckard Bringmann
DaimlerChrysler
Alt Moabit 96a
10559 Berlin, Germany
eckard.bringmann@daimlerchrysler.com

Abstract
The test of embedded systems is well-supported by a number of different test methods and tools if the system under test displays a discrete behavior. However, for systems with continuous behavior these approaches are not appropriate. Practical experiences with the test of embedded systems in the automotive industry have revealed that testing the continuous behavior is different from testing discrete systems. In this paper, we discuss the reasons for the specialities of continuous behavior testing and the consequences for practicing testers. In addition, we introduce a new test approach that enables the systematic definition of executable test cases for testing the continuous behavior of automotive embedded systems. This method is already in use in several production-vehicle development projects at DaimlerChrysler and is based on a graphical notation for test cases which is easy to understand but also powerful enough to express complex, fully automated tests as well as reactive tests of continuous behavior.

Keywords
continuous behavior testing, reactive test, test automation, real-time test, test specification language, systematic test case design

1 Motivation

Automotive software is embedded in control systems (e.g. central locking system), feedback control systems (e.g. engine control), or information systems (e.g. instrument cluster). In general, such systems have a fairly complex functional behavior with large interfaces and usually deal with continuously changing signals as input and output quantities. This kind of behavior is called the continuous behavior of a system\(^\dagger\).

\(^\dagger\) Some authors use the term dynamic behavior instead of continuous behavior.
The test of systems with continuous behavior 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. Even worse, there is no systematic procedure to help testers reveal redundancies and missing test-relevant aspects within their test cases. In other words, the selection of test data occurs ad hoc and is based on some use cases and typical extreme scenarios, but often does not cover all functional requirements of the system under test (SUT).

2 Requirements for Testing the Continuous Behavior

There are several requirements that are special for testing the continuous behavior of embedded systems and, therefore, require the use of special testing approaches. These requirements are discussed below.

2.1 Test automation

Embedded systems with continuous behavior usually interact with a real-world environment which is constantly controlled by 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. As a consequence, an iterative process is needed with a considerable number of intermediate pre-releases of the integrated system. Thoroughly testing these pre-releases is crucial to ensure that faults in the design or implementation can be revealed as early as possible. This means that the same tests have to be repeated again and again over the development cycle. To ensure efficient test processes fully automated tests are needed. For that reason, test automation is one of the most important requirements in practice.

In addition, testing the continuous behavior often requires test scenarios with a precise sequence of time-sensitive actions. The only way to execute such scenarios is automation.
2.2 Consistency between test platforms

Embedded controllers are integrated in embedded devices with proprietary hardware (which is called ECU). In order to test the controller functionality as early as possible, tests must be performed before the software is integrated into the final ECU. In general, the following test platforms are distinguished:

- **Model-in-the-Loop (MiL):** The first test platform only exists if the system developed is model-driven. This means that the embedded software development is performed by means of the graphical simulation languages of tools such as Matlab/Simulink or Statemate instead of traditional programming languages. Testing an embedded system on MiL level means that the software and its environment are simulated (interpreted) in the modeling framework without any physical hardware components. This allows tests at early stages of the development cycle.

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

- **Processor-in-the-Loop (PiL):** Testing on PiL level is similar to SiL tests, but the embedded software runs on a target board with the target processor. These tests can reveal faults that are caused by the target compiler or by the processor architecture.

- **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. 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.

Efficient development of embedded systems with continuous behavior requires tests on the various test platforms described above. However, the system under test and, with that, the relevant test cases are always the same independent of the test platform. Thus, a test procedure for embedded systems with continuous behavior should support the 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 the easy comparison of test results between the different levels. Although this requirement
sounds trivial it is the one weak point in today's testing practice for continuous behavior testing because test procedures and test languages are usually specialized for one particular test platform and are very difficult to transform to another platform.

2.3 Systematic test case design

For many reasons, embedded systems with continuous behavior usually consist of a complex functionality. First of all, the systems interact with physical components that have a complex behavior which depends on many quantities. Controlling such components also causes complex controller functionality. Secondly, electrical, mechanical, and hydraulic components can fail and the embedded system has to detect such failures and compensate them, if possible. The implementation of handling all of these exceptions makes a big part of the software.

Testing such complex systems requires a thoroughly considered selection of test cases to ensure that all test-relevant aspects are covered and that redundancies are avoided. A test procedure must support this thorough selection process by providing means that allow keeping an overview over the set of selected test cases, even if this set contains hundreds or thousands of test cases.

2.4 Readability

Testing embedded systems with continuous behavior is a collaborative work between testers, system engineers, and programmers. All of these experts have different perspectives on the system and provide important information for the identification of test-relevant aspects: system engineers know about 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 that have to be considered in the tests.

As a consequence test cases must be readable and easily understandable by all of these experts and not just by a few testing specialists.

2.5 Reactive testing

When testing the continuous behavior of embedded systems test cases often naturally depend on the system behavior. That is, the course of a test case depends on what the system under test is doing while testing it. In this sense the system under test and the test driver run in a 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. 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.

2.6 Real-time issues and continuous signals

As mentioned in section 2.2 testing on 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 measure.

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

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

3 The “Time Partition Testing” approach

In section 2 a number of requirements for testing the continuous behavior of embedded systems have been discussed. Although there are test approaches that address some of these requirements there is no approach so far that fulfills all of them. The Time Partition Testing (abbreviated as TPT) is a new approach with the clear objective to support all of the requirements above.

TPT has been specifically designed to support continuous behavior tests. The objective of the Time Partition Testing is (1) to support the systematic selection of test cases, (2) facilitate a precise, formal, but easy to read representation of test cases for continuous behavior testing, and (3) thereby provide an infrastructure for automated test execution and automated test assessments even for real-time.

As a general design principle, TPT test cases are independent of the underlying 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 test platforms (i.e., back-to-back testing).

The TPT test method is strongly associated with a corresponding test language. The TPT test method has the objective to support the systematic selection of relevant test cases. A language is, therefore, necessary to describe these selected cases in a reasonable way. However, the test language, in turn, affects the way test cases are modeled, compared, and selected, and that therefore the language affects the test method itself. Due to this strong relation 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 increasing the test efficiency. Redundancies and missing test relevant scenarios can only be identified by seeing the set of test cases as a whole. As a consequence, 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. First of all TPT provides a language for modeling executable test cases for continuous behavior which is explained in section 3.1. In addition, TPT supports the systematic test case selection as an extension of the test language, as described in section 3.2.

### 3.1 Modeling single Test Cases

The TPT test language is best explained by means of a simple example of an exterior headlight controller (EHLC). An outline of a possible specification looks as follows:

There is a switch with three states: **ON**, **OFF**, and **AUTO**. The headlights are on when the switch is set to **ON**, and off when the switch is set to **OFF**. If the switch is in the **AUTO** mode, the headlights are switched on if the ambient light around the car falls below 200lux and are turned off if the ambient light exceeds 1000lux (hysteresis curve). If the switch is in the **AUTO** mode when the system starts, the lights will be switched on/off if ambient light is below/above 800lux. The ambient light is detected by a light sensor which has a range from 0lux to 50,000lux. Generally, the lights when turned on should remain on for at least 4 seconds, and when turned off, should remain off for at
least 2 seconds to avoid flickering lights (this may cause a delay in adjusting the switch and the reaction at the lights). The system interface is shown in Figure 1.

Figure 1: Example (Exterior headlight controller EHLC)

The first test case lasts for 17 seconds and starts with the switch in the OFF position. After a period of 2 seconds the switch is turned to the ON position and held 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 2.

Figure 2: A simple test case

This simple graphical notation has the same meaning as the textual description above, but is easier to handle for complex test scenarios and provides a more formal way to describe the procedure. Nonetheless, for test automation more formal semantics are needed. We assign simple equations to the states and temporal predicates to the transitions (see Figure 3). Usually these formulas are hidden behind the graphics.

Figure 3: Test case with formal statements

Each state semantically describes a stream-processing component, even if all streams are constant signals in this simple example. 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. The semantics of such a state machine is a hybrid system.

With these formulas the example 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 in this test case (if the system behaves correctly), a definition is necessary in order to have a unique, reproducible test case.

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 4. The concrete definition describes a signal that starts at 500.0lux and constantly increases with gradient 10lux/s. For a more realistic scenario a noise component has been added. The syntax and semantics of the concrete formulas in this example are not explained in detail here.

\[ \text{sensor}(t) := 500 + 10 \times t + 4 \times \text{noise}(t/10) \]

**Figure 4: The complete test case**

Since all inputs of the SUT are now completely determined by the test case, it can be executed. The result of this test execution covers both the input signals and the output signal of the SUT. The corresponding curves can be seen in Figure 7.

**Figure 5: Test data after execution**

In the example the test case and the SUT behave as expected. Since the switch is never in the AUTO position in this scenario, the sensor signal does not affect the be-
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. For practical usage there are more sophisticated techniques available such as transition branches, hierarchical state machines, junctions, actions at transitions, and others, which can not be explained in detail at this point of time, due to the limited amount of space of this paper.

Formal definitions of states can be given by means of equations or systems of equations, or by any element that describes a stream-processing component. Thus, states can be described by embedded state machines, which allows the modeling of hierarchical state machines for example.

3.2 Modeling Test Case Sets

In section 3.1 the language for modeling a single test case has been described briefly. Now we consider another test case: Again we want to turn the light ON, but this time only for 3 seconds. By specification the headlights remain on for at least 4 seconds, which is checked by this test case. This test case is similar to the first one, as shown in Figure 6.

After test execution, the curves are similar to the first test case. The system behaves as expected (see Figure 7; sensor is equal to the curve in Figure 5). The headlights remain on for 1 second after the switch has been turned to OFF.
Both test cases defined so far are almost identical. Only the condition of the third transition is different. Remember that the reason for defining the second test case was the requirement that headlights must always remain on for at least for 4s. Thus, the difference regarding the third transition is related to this requirement because we must consider the aspect of how long the headlights have been turned on when the switch is turned off again with two alternative variants: shorter and longer than 4 seconds.

As a consequence of this consideration, the functional requirements tested by a set of test cases manifest themselves only in the differences between the test cases. If something is different between two test cases, obviously, there must be an unique functional requirement with at least two different possible outcomes, which are being tested by the test case. If such a requirement cannot be found, the test cases are inevitably redundant.

To cut a long story short, 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, with TPT, all test cases are based on a single model that can be considered as a “super model of all test cases”. This super model is called a testlet.

To illustrate this concept we use the two test cases introduced above. The integrated testlet is shown in Figure 8. The general structure of both test cases is identical. The third transition has two alternative formal definitions which check one of the functional requirements causing the difference between the two cases. Elements in the model that have more than one formal definition are called variation points.

The testlet model itself is not executable because the semantics at the variation points are ambiguous. However, to derive a concrete test case from this model it is sufficient to choose one of the two variants for every variation point. According to this, test case 1 chooses the variant "longer than 4s" while test case 2 chooses the variant "shorter than 4s".
Now we introduce a third test case that treats the same situation as in test case 2 but it starts with the switch in the AUTO position and the ambient light as daylight to ensure that the headlights are turned off initially. The testlet must be extended in order to integrate this test case, as shown in Figure 9.

The extended testlet model has three variation points: the “initial switch” state, the transition “when lights have been turned on”, and the “ambient light” state. Each of the variation points has two alternative variants. The maximum number of possible combinations – and thereby the maximum number of possible test cases – is $2^3=8$. The three example test cases are a subset of these 8 possible test cases. Table 1 summarizes the combination of the chosen variants for all three cases.
Table 1: Variations of test cases

<table>
<thead>
<tr>
<th>Test case</th>
<th>Initial switch</th>
<th>when lights have been turned on</th>
<th>ambient light</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 1</td>
<td>off</td>
<td>longer than 4s</td>
<td>with noise</td>
</tr>
<tr>
<td>Test case 2</td>
<td>off</td>
<td>shorter than 4s</td>
<td>with noise</td>
</tr>
<tr>
<td>Test case 3</td>
<td>auto</td>
<td>shorter than 4s</td>
<td>daylight</td>
</tr>
</tbody>
</table>

With the selection of one variant for every variation point a test case is precisely defined and can be executed automatically. 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 thousands or millions of possible combinations. To keep such large models comprehensible, TPT utilizes the idea of the classification tree method [2,4]. While this method is important in the practical testing of huge test problems, it will only be briefly explained in this paper.

In general, the classification tree method has the objective to simplify the test case selection on an abstract level. 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 the examples is depicted in Figure 10 below.

The classification tree and the corresponding combination table can be automatically generated for each testlet. The tree representation is therefore an alternative view of the test problem that focuses on the combinatorics whereas the state machines focus on the general test procedure 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.

4 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 11 and explained in the following.

**Figure 11: TPT test process**

**Test case design:** During the test case design, test cases are selected and modeled by means of the language already described in section 3. The basis of this test case design is the functional system requirements.

**Compiling:** Test cases are compiled into compact 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 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.

**Test execution:** During test execution the virtual machine (TPT-VM) executes the byte code of the test cases. During execution the TPT VM communicates continually with the SUT via so-called 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, 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.
**Test assessment:** The recorded test data are initially just raw signal data without any evaluation of whether the behavior of the SUT was as expected. These data are then automatically assessed by means of the compiled assessment scripts. Since test assessments are performed off-line, real-time constraints are irrelevant for this step. Currently, TPT uses Python as the script language so that an existing Python interpreter can be used as the runtime engine. However, TPT does not rely on the actual scripting language or on the interpreter.

**Report generation:** With the results of the test assessment, a report is generated that depicts the result of the test case in a more human-readable way. For that purpose, the report contains the test result (with one of the values *success*, *failed*, and *unknown*), curves of relevant signals, data tables with computed and assessed results as well as customizable comments that illustrate the evaluated behavior. Test reports can be generated in HTML and PDF formats.

Summarizing, it can be stated that 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 and others are not covered by TPT. However, integration with external tools, such as TestDirector for test management, is currently under development at DaimlerChrysler.

## 5 Practical Experience

TPT has been developed at DaimlerChrysler Research and established in cooperation with production-vehicle development projects. Therefore, TPT contains a lot of solutions for simplifying the daily test practice in automotive development projects. Today, TPT is already the major testing approach in some interior and power train projects. It is even used by some suppliers to specify, exchange and obtain agreed-upon test cases which are used by DaimlerChrysler as part of our final acceptance test.

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 “super state machine” 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.
6 References


