Systematic Testing of the Continuous Behavior of Automotive Systems

Eckard Bringmann
DaimlerChrysler
Alt-Moabit 96a
10559 Berlin, Germany
+49 30 39982 242
eckard.bringmann@daimlerchrysler.com

Andreas Krämer
DaimlerChrysler
Alt-Moabit 96a
10559 Berlin, Germany
+49 30 39982 336
andreas.kraemer@daimlerchrysler.com

ABSTRACT
In this paper, we introduce a new test method that enables the systematic definition of executable test cases for testing the continuous behavior of automotive embedded systems. This method is based on a graphical notation for test cases that is not only easy to understand but also powerful enough to express very complex, fully automated tests as well as reactive tests. This new approach is already in use in several production-vehicle development projects at DaimlerChrysler and at some suppliers.

Categories and Subject Descriptors
D2.5 [**Software Engineering**]: Testing and Debugging – *testing tools*

General Terms
Verification

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

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[1].

Testing 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 testing 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).

Our new approach – which is called *Time Partition Testing* (abbreviated as 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 simple 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 environments – which is important for hardware-in-the-loop tests, for example.

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).

2. SEMANTICS
Semantically, the TPT test language is based on a clear compositional model of so-called *stream-processing components*. This theoretical model is explained briefly in the following section to emphasize the computational power of TPT. However, practising testers do not need to become acquainted with this theory. As we will show in section 4, although TPT is built on this theory, it is a method for practising testers, not for theory enthusiasts.

2.1 Stream-processing Components
The time model behind TPT is dense real-time. We use the set of real numbers $\mathbb{R}$ as a representation of real-time. For modeling signals and hybrid components, we adopt the theory of stream-processing functions proposed by [1] and [6]. This concept has been extended by the expressiveness of hybrid systems [5]. In the following paragraphs we give a short discourse about the

[1] Some authors use the term *dynamic behavior* instead of *continuous behavior*.

Conference'04, Month 1–2, 2004, City, State, Country.

fundamental idea of this theory. For a detailed introduction we refer to [3].

Every set M containing a special value $\varepsilon \in M$ is called a *type*. A total function $s: \mathbb{R} \to M$ is called a *stream over M*. Partial streams are modeled by means of the element ε, that is $s(t)=\varepsilon$ means that s is undefined at time t. Let C be a finite set of so-called *channels* (or *variables)* which are used to assign streams to. An *assignment c* for a set of channels C assigns a stream $c_\alpha: \mathbb{R} \to M_\alpha$ to every channel $\alpha \in C$. $\vec{C}$ denotes the set of all possible assignments of C.

For two assignments $c, c' \in \vec{C}$ and a time interval I we say c and c' are *equal in I*, denoted as $c =_I c'$, if the streams c_α and c_α' are equal in I for all channels $\alpha \in C$. As abbreviations we use $c =_{<t} c'$ (short hand for $c =_{(-\infty, t)} c'$) and $c =_{\leqslant t} c'$(short hand for $c =_{(-\infty, t]} c'$).

With these terms we can introduce a stream-processing component as follows: Let I and O be two disjointed sets of channels. A *stream-processing component* is a relation $F: \vec{I} \leftrightarrow \vec{O}$ that fulfills the following constraint, which is called *time causality*: There is a $\delta > 0$ so that for all $t \geq 0$ and for all assignments $i, i' \in \vec{I}$ and $o, o' \in \vec{O}$ with $(i, o) \in F$, $(i', o') \in F$ and $o =_{<0} o'$ the following property holds:

$$i =_{\leqslant t} i' \wedge o =_{\leqslant t-\delta} o' \Rightarrow o =_{\leqslant t} o'$$

The test language introduced in section 3 allows the description of test cases that represent stream-processing components semantically. The outputs O of a test case are inputs of the SUT and vice versa. Therefore it is important that the output at some t can be computed on-line. This characteristic is expressed exactly by time causality. Time causality guarantees that F determines the

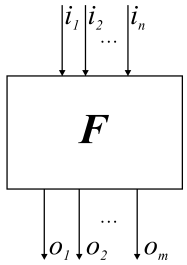

Figure 1: Stream-processing component

behavior at the outputs O up to time t unambiguously, if the behavior at the inputs I is known up to t and the behavior at the outputs is known up to $t-\delta$. By definition, we assume that a component always "starts" at $t=0$. For that reason, the constraint above is only required for $t \geq 0$.

Although this component term is based on dense time, it provides a concept of runtime computability without referring to discrete computational steps.

Every stream-processing component F determines O for all non-negative points in time in dependence of I, as the following theorem underlines:

For any input assignment i and two output assignments o and o' with $(i, o) \in F$ and $(i, o') \in F$, both outputs are equal ($o = o'$) if they are equal in negative time ($o =_{<0} o'$). The proof is given in [3].

Note that this component definition includes partial components, such as $F = \emptyset$, which are irrelevant in practice. We therefore say a component F is *total*, if and only if, for every pair *(i,o)*, there is at least one o' with $(i, o') \in F$ and $o =_{<0} o'$. If a component is not total, it is called *partial*. From the theorem above, it follows that o' is unique if F is total. In other words, for a given input assignment i and a given output assignment o in negative time, a total component uniquely determines the outputs o for all points in time. Every test case modeled by means of the TPT test language describes a total stream-processing component semantically (see section 3).

2.2 Operators

Components can be composed into more complex components by means of some powerful operators which are introduced in detail in [3].

The *parallel operator* allows two components F_1 and F_2 to run in parallel, denoted as $F_1 || F_2$ (see Figure 2). Both components may reciprocally use the outputs of the other component as inputs. The *feedback* operator μF allows recursive functions to be defined by sticking some input and output channels of F together. *Equations* are components of the form *c*(t)=*expression.* This component has just one output channel c which is explicitly defined for every $t \geq 0$ based on the time dependent expression on the right-hand side of the equation. The *sequential* operator $F_1 \overset{P}{\sim>} F_2$ combines two components F_1 and F_2 in such a way that F_1 determines the behavior of the integrated component until a certain point in time, which is specified by a temporal predicate P. F_2 determines the behavior from that time on.

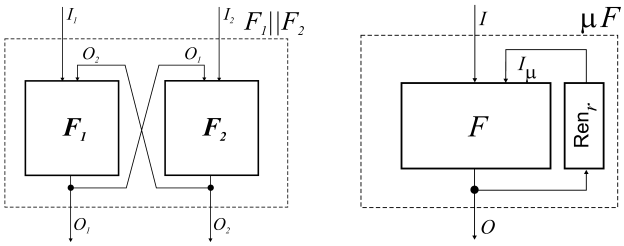

Figure 2: Parallel and feedback operator

The most complex operator is a hybrid system. A hybrid system is based on a finite state machine with a stream-processing component assigned to each state and a temporal predicate assigned to each transition. In this way the hybrid system is a generalization of the *sequential* operator and allows modeling of complex stream-processing components with the expressive power of state machines. The formal definition of hybrid systems as stream-processing components is given in [3].

All operators preserve the totality of components [3].

2.3 Practical Gain of this Theory

The major objective of this theory in context of the TPT test language introduced in section 3 is to provide a compact, problem-oriented, and abstract model for test cases that interact with the SUT in dense real-time. An essential aspect is the modularity and compositionality of stream-processing components supported by the different operators.

As mentioned above, each TPT test case semantically describes a total stream-processing component running in dense time. However, in order to execute test cases automatically, we must discretize these continuous semantics with some sample rate $\delta > 0$. The sampled executable semantics differs from the intended continuous semantics in terms of timing and value imprecision. The sampled semantics converges towards the continuous one if δ tends to $+0$.

Discretization is a common issue for continuous system design and has usually been already dealt with during interface design. Sampling is, therefore, not a problem in practical application of TPT; it can simply be reused from system design.

3. TEST METHOD AND LANGUAGE

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 3.

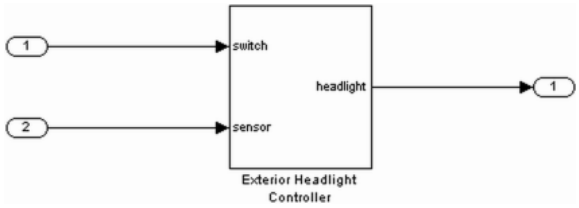

Figure 3: Example (Exterior headlight controller EHLC)

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 4.

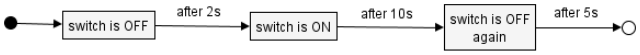

Figure 4: A simple test case

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. As described in section 2.2, we assign simple equations to the states and temporal predicates to the transitions (see Figure 5). Usually these formulas are hidden behind the graphics.

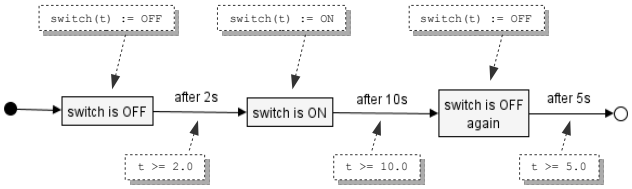

Figure 5: Test case with formal statements

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, as mentioned in section 2.2.

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 6. 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.

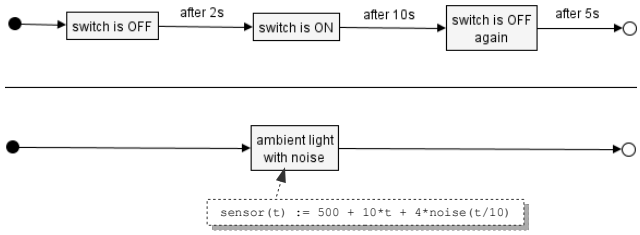

Figure 6: The complete test case

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.

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 behavior of the headlights at all. Headlights are turned on and off synchronously with the

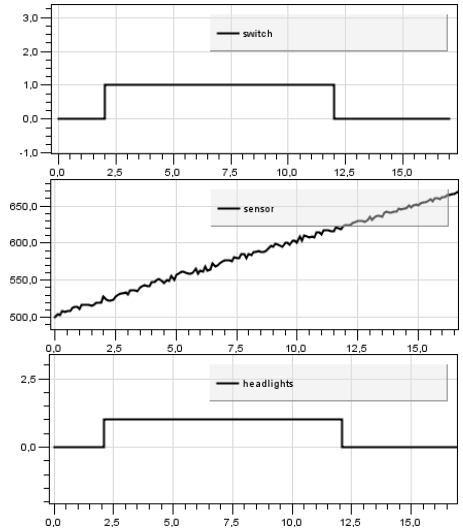

Figure 7: Test data after execution

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 8.

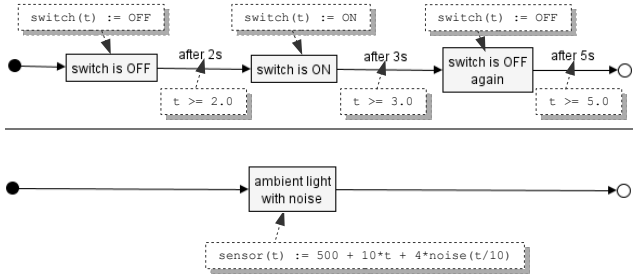

Figure 8: The complete test case

After test execution, the curves are similar to the first test case. The system behaves as expected (see Figure 9; `sensor` is equal to the curve in Figure 7). The headlights remain on for 1 second after the switch has been turned to `OFF`.

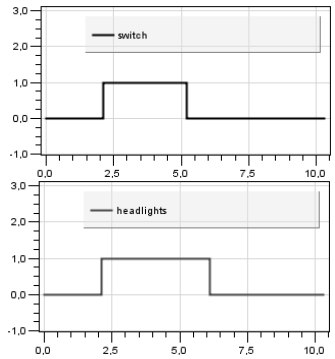

Figure 9: Test data after execution

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 an 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 10. 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".

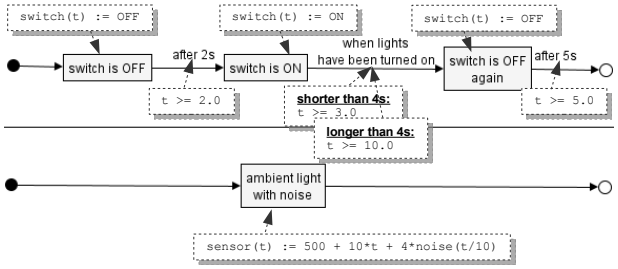

Figure 10: The integrated testlet

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 11.

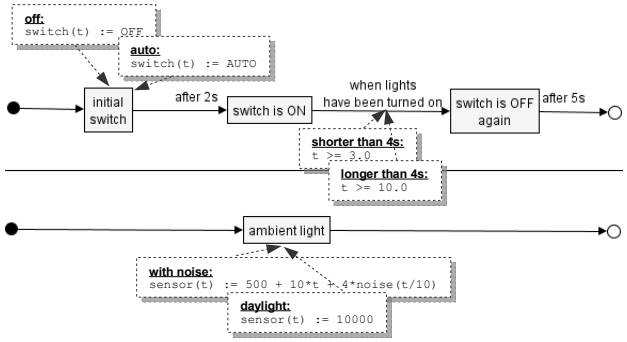

Figure 11: Extended testlet

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 $2 \cdot 2 \cdot 2 = 8$. 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

Test case	Initial switch	when lights have been turned on	ambient light
Test case 1	off	longer than 4s	with noise
Test case 2	off	shorter than 4s	with noise
Test case 3	auto	shorter than 4s	daylight

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 12 below.

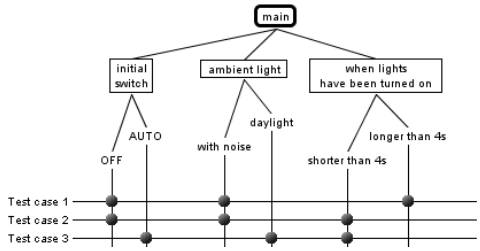

Figure 12: Classification tree as an alternative view

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.

3.3 Assessment Properties

Up to this point the test cases formulate how the tests should be performed. However, the expected results are still missing. TPT

supports so-called *assessment properties* that can be used to precisely express the expected results. Assessment properties can be automatically analyzed to guarantee an efficient test process. Assessment properties can be computed on-line (that is in parallel to the test execution) and off-line (that is after the test execution has been finished). While the on-line assessment uses the same techniques as the test case modeling, the off-line analysis offers a set of more complex features and functions. This includes, for example, operators for signal comparison with external references signals, boundary checks, signal filters, analysis of state sequences, and timing constraints.

In practice, it is not sufficient to compute whether a test case has revealed an error in the SUT or not. In almost all cases, more details are necessary to understand the behavior of the SUT. As a consequence, TPT assessment properties can compute both, single values (such as OKAY/FAILED or the duration of some system state) and signals (such as the difference between two signals or the gradient of a signal). These properties are derived from the signal data that have been logged during the test execution or from other assessment properties.

Technically speaking, the computational engine for TPT assessments is based on Python that has been extended by specific syntactical features and a specialized assessment library to support the test assessment. The advantage of using a scripting language is that it guarantees high flexibility: accessing reference data from external files, communication with other tools, and development of domain specific libraries are possible without any problems.

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

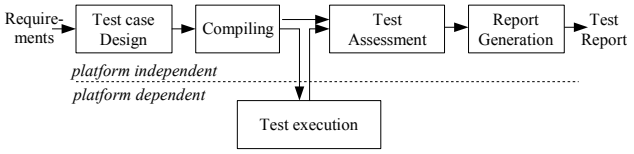

Figure 13: TPT test process

goes far beyond this activity. The overall testing process of TPT is defined as presented in Figure 13 and explained in the following.

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 very 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, 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.

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

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. SUMMARY AND OUTLOOK

Time Partition Testing is a test method that facilitates the design, execution, assessment, and report generation of test cases for automotive systems that show continuous behavior. At the same time, it provides a systematic approach for test case selection that helps to reveal redundancies and missing, but relevant aspects in test sets.

TPT test cases are independent of the architecture and technology of the system under test due to an abstract semantic model that the test language of TPT is based on. This enables test cases to be easily reused 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. Formal details are hidden behind graphical models. Thus, TPT test models are comprehensible for experts from different domains, such as testers, programmers, system designers, and assessors from authorities.

Current research activities at DaimlerChrysler focus on advanced integration with the classification tree method [2] for semi-automated test case selection by means of the combinatorial generation rules of the CTE XL [4]. In addition, integration with advanced testing technologies, especially the evolutionary testing [7] is under development. The objective of this integration is to automate tests which are difficult to specify using explicitly defined input signals and are characterized by logical constraints. As a result of this integration, new test cases have been found by means of evolutionary testing algorithms.

7. REFERENCES

[1] Manfred Broy. Refinement of Time. *In Transformation-Based Reactive System Development*, Volume 1231 of LNCS, pages 44-63. Springer-Verlag, 1997.

[2] Matthias Grochtmann and Klaus Grimm. Classification Trees for Partition Testing. *Software Testing, Verification & Reliablity*, 3(2):63-82, 1993.

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

[4] Eckard Lehmann (Bringmann) and Joachim 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] Oded Maler, Zhar Manna, and Amir Pnueli. From Timed to Hybrid Systems. In *RealTime: Theory in Practice.* LNCS, pages 447-484. Springer Verlag, 1992.

[6] Olaf Müller and Peter Scholz. Functional Specification of Real-Time and Hybrid Systems. In *Proc. Hybrid and Real-Time Systems*, LNCS. Springer Verlag, 1997.

[7] Harmen Sthamer, Joachim Wegener, and Andre 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.