

Time Partition Testing: A Method for Testing Dynamic Functional Behaviour

Eckard Lehmann

DaimlerChrysler AG, Research and Technology

Abstract

This paper introduces the *Time Partition Testing* approach, a method for systematically testing the dynamic functional behaviour of embedded systems. *Time Partition Testing* supports the selection and documentation of test data on the semantic basis of so-called *testlets* and the syntactic techniques *Direct Definition*, *Time Partitioning* and *Data Partitioning* which are used to build testlets. Testlets facilitate an exact description of test data and guarantee the automation of test execution and test evaluation.

1 Introduction

Test procedures determine how to select test data, namely those input data that will be used for testing a system. Test procedures where the selection of test data is determined on the basis of the requirements specification are called functional tests, black-box tests, or behavioural tests[1,9]. By using functional test procedures test data are selected in order to systematically check the functionality specified in the requirements specification.

As a rule, test data selection for functional testing cannot be automated because requirements specifications usually exist as verbal and semi-structured documents. Despite this weakness of automation, functional testing has been continuously enhanced over the past 30 years and has been applied intensively in the field of commercial software development.

This success grounds on the fact that compared to other test procedures functional testing is the only way to detect incompleteness of the implementation with respect to functional requirements. Furthermore, a systematic specification analysis helps to concentrate on examining critical areas and more or less to ignore non-critical areas. This keeps the over-all number of test data comparatively low and the expenditure for test execution, documentation, and evaluation remains reasonable. Functional testing is very intuitive. "*When most people first start testing, especially if they do so without formal training in the subject, they rediscover ... functional testing.*"[1] Its popularity is also based on the fact that manually defined test data are better understandable than automatically generated data and this facilitates the test evaluation. As a whole, functional testing is very effective and practicable.

If one believes forecasts concerning the development of electronic control units, the number of embedded software-based systems will increase enormously. The outstanding characteristic of embedded systems is the fact that they interact closely interlinked with a real world environment. This environment usually is a highly complex one and cannot be characterised by means of a simple abstract model. Therefore embedded systems need to *observe* their environment and *react* correspondingly to its behaviour. The system works in an interactional cycle with its environment¹ (see. Fig. 1).

¹ For simplification here the term *environment* includes the entire surroundings of the system. System and environment interact within a closed loop.

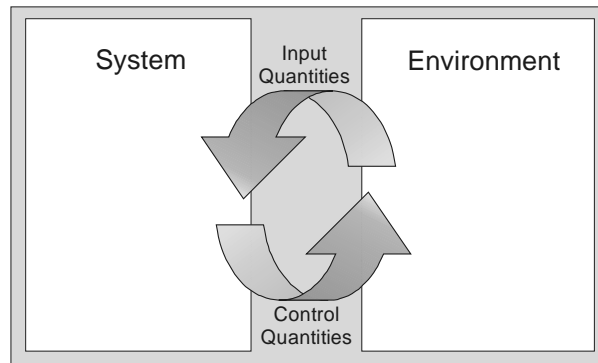


Fig. 1: Interacting system and environment

Example 1: A heating system controls the temperature of a room. The desired temperature can be set manually. The system (controller) needs to react to the actual room temperature and to the temperature setting (input quantities) in order to set a corresponding heating level (control quantity) for the heater. In this case, the room and the person who chooses a temperature constitute the environment.

Example 2: A motor control system controls the engine parameters (control quantities: throttle potentiometer setting, advance angle, etc) depending on the desired engine torque (input quantity) which itself depends on the position of the accelerator. In this example the engine, vehicle, and the driver are the environment.

It is hardly possible to explain the functionality of a system without taking into account its environment. In example 1 the system's function – from the user's point of view – is to regulate the temperature to meet the desired level. The system itself merely defines a heating level. So, the system's direct output often is quite uninteresting. Only the effect within the environment makes it possible to explain a system's functions.

In general, behaviour within a real world environment is subject to temporal constraints (e.g. due to inertia in mechanics) therefore functionalities are usually also subject to timing constraints. This means, for embedded systems each test data needs to consider a temporal sequence in order to check the functionality.

Traditional functional testing methods concentrate on test data with single inputs and therefore cannot be applied effectively to the system class mentioned above. Consequently, functional tests of such systems are currently carried out more or less pragmatically in practice. The selection of test data usually happens ad hoc and is based on a few typical use cases of the system instead of extreme-scenarios and cases with high error probability.

On this account the method for *Time Partition Testing* has been developed within the DaimlerChrysler research department. The method facilitates systematic functional testing of dynamic system behaviour. The objective of *Time Partition Testing* is to (1) support the selection of test data, (2) facilitate a precise, formal, and simple representation of test data, and therewith (3) create a basis for an automated test execution.

The underlying semantic concept on which *Time Partition Testing* is based, is the concept of so-called *testlets*. This concept will be introduced in Section 2. Subsequently there will be an account of examples in Section 3 which give a first idea of the method and the possibilities of *Time Partition Testing*. The three central techniques of *Time Partition Testing*, namely *Direct Definition*, *Time Partitioning* and *Data Partitioning* will be introduced in Section 4. Finally automation possibilities in regard to test execution (Section 5) and test evaluation (Section 6) will be discussed.

2 Testlets

First, it should become clear how test data is modelled in *Time Partition Testing* and how test data, system, and environment interact. The closed loop in which system and environment interact inevitably has to be manipulated for test execution in order to be able to

use test data. This manipulation is achieved through substituting parts of the environment by a test driver. The role of test drivers is to deliver test data during test execution. A test driver may supply input quantities of the system as well as environment quantities. In example 1 a test driver might determine the desired temperature (system input quantity). In example 2 a test driver could determine the position of the accelerator (environment quantity) which is not a direct input quantity for the motor control.

In order to keep the modelling possibilities of test data most flexible and easy-to-use, test data also need the ability to “react” to the behaviour of system and environment. If, in example 1, it is to be tested what happens when an emergency button is pushed the moment the temperature exceeds 60°C, the room temperature is a quantity to which the test data has to “react”. The course of temperature determines *when* the emergency button has to be pushed.

From the test’s perspective a test driver can *supply* system input and environment quantities in the same way as well as it is able to *react* to system output and environment quantities equally. Therefore, it is not necessary for the test driver to distinguish system and environment quantities. The relation may be represented as shown in the diagram below (Fig. 2).

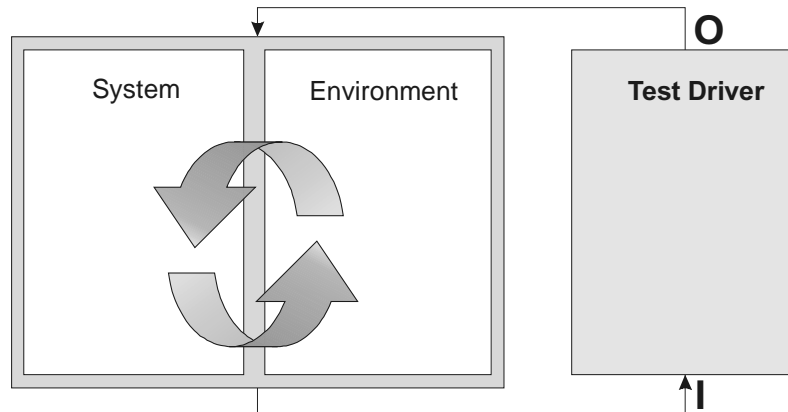


Fig. 2: Test driver interacting with system and environment

Common features of all test drivers are their clearly defined input and output interfaces to environment and system respectively. As a rule, these interfaces consist of a number of *channels* which transfer signal courses (value courses over time). In example 1 the room temperature could be seen as an input channel of the test driver, the alterable desired temperature and the emergency switch position as output channels of the test driver.

This is exactly the point where *Time Partition Testing* begins. Each test data – semantically – describes signal courses (over time) for all output channels depending on the signal courses of input channels. In order to obtain a more precise description of test data, a couple of formal terms will be introduced first. Input and output channels are labelled K_I and K_O respectively. Each is a subset of a universal label set K for labelling channels. D_k identifies the range of value for each channel $k \in K$ that can be assigned to k , i.e. at any time one of the D_k values is associated to channel k , hence the signal course is a partial² function $Time \xrightarrow{p} D_k$. Assumed temporal model is a dense time with the minimum value 0_s , i.e. $Time = \mathbb{R}^+ \cup \{0\}$, because this model is more natural and easier to apply for the formulation of signal courses than a discrete temporal model would be. The set of all possible signal courses for one channel k is labelled $\tilde{k} = (Time \xrightarrow{p} D_k)$.

The test execution for each test data has to terminate after a definite span of time or may run forever. Therefore, the time of termination is $t_{stop} : Time^\infty$ with $t_{stop} = \infty$ if a test should not

² The partiality of signal courses is granted, in order to express “unknown values”. This may be used for example for channels which are transferred in data packages via a bus.

terminate at all. However, similar to courses of output channels the value of t_{stop} might depend on courses of input channels.

With these formal terms a test data can be characterised as a function $td: \tilde{i}_1 \times \dots \times \tilde{i}_n \rightarrow Time^\infty \times \tilde{o}_1 \times \dots \times \tilde{o}_m$ with $K_I = \{i_1, \dots, i_n\}$ and $K_O = \{o_1, \dots, o_m\}$. It describes the time of termination and a signal course for each output channel in dependence of signal courses of input channels.

This construction of conditional signal courses with a conditional time of termination is a central concept of *Time Partition Testing*. Therefore it will be generalised as follows: A non-empty set of functions $T \subseteq (\tilde{i}_1 \times \dots \times \tilde{i}_n \rightarrow Time^\infty \times \tilde{o}_1 \times \dots \times \tilde{o}_m)$ to the sets of input and output channels K_I and K_O is called a *testlet*, an element from T is called *testlet element*. In this sense, the set of test data for testing a system is a specific testlet. In order to ensure the acceptance of *Time Partition Testing* in practice, the construction of testlets has been defined in abstract and simplified manner. The formal semantics of testlets is merely a side issue in practical use of the method. Initial experiments have shown that the concepts of *Time Partition Testing*, similar to programming languages, can be easily learned by doing, whereas the clear-cut semantics of testlets is only required to clarify controversial issues and for the automation of test execution.

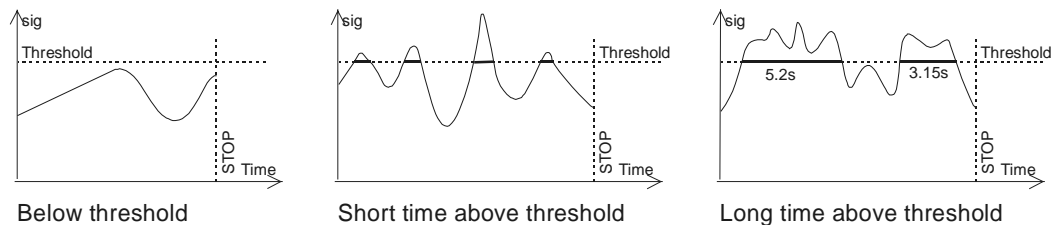
For systematic reasons each testlet element $t \in T$ will be human-readable labelled in order to improve clarity and comprehensibility of test data. Still, this label is irrelevant to the semantics of test data, therefore it will not be formalised.

3 Examples of Application

In this section, simple and intuitive examples shall show the principle of *Time Partition Testing*. They will demonstrate how testlets are built syntactically and explain the corresponding testlet semantics. *Time Partition Testing* includes three different techniques for the construction of testlets: *Direct Definition*, *Time Partitioning*³ and *Data Partitioning*. There will be one example for each technique. The subsequent section (Section 4) will provide a more detailed description of the three techniques.

Example 3 (Direct Definition): A system's task is to check whether a continuous input signal sig exceeds a threshold value. If sig exceeds the threshold for at least 3 seconds, a warning flag should be set by the system.

In this simple example test data can be listed. This is possible for small systems or simple functionalities as long as the set of relevant data is clear. By using *Direct Definition* for each test data the concrete signal course for sig and the termination point t_{stop} are drawn graphically. The situation under test is therefore clear as shown in the figure below.

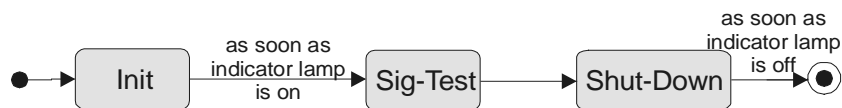


The three test data build a testlet with no input channel ($K_I = \emptyset$) and one output channel ($K_O = \{sig\}$). Each test data (testlet element) has its signal course for sig and t_{stop} fixed because there is no input signal course on which it might depend.

³ The term *Time Partition Testing* originates in the central and typical technique of *Time Partitioning* for partitioning the temporal course of test data. Still, the *Time Partition Testing* method includes all techniques and semantic concepts which are described in this paper. This differentiation is mentioned here in order to avoid misunderstandings.

Example 4 (Time Partitioning): The system in example 3 shall now be equipped with a main switch (*power*) in order to turn it on or off. By switching it on the system will be initialised. After initialisation has been completed (after ca. 5-10s) an indicator light *indicator* goes on. Then the signal *sig*, as described in example 3, will be checked for threshold overflows. After switching off the system, the indicator light will not go out until the system is completely inactive.

For the test this means that each test data describes two output channels: one for the signal *sig* and one for the switch *power* ($K_O = \{sig, power\}$). The signal course *sig* is not relevant to the test until the *indicator* goes on (i.e. until the system is ready to work). Therefore, each test data should be able to react to the indicator light in order to be able to differentiate between initialisation, work phase, and shut-down phase. Therefore, *indicator* is the only input channel ($K_I = \{indicator\}$). By using *Time Partitioning* each test data will be partitioned into three subsequent phases which is expressed by means of state transition diagrams[5,8] as follows.

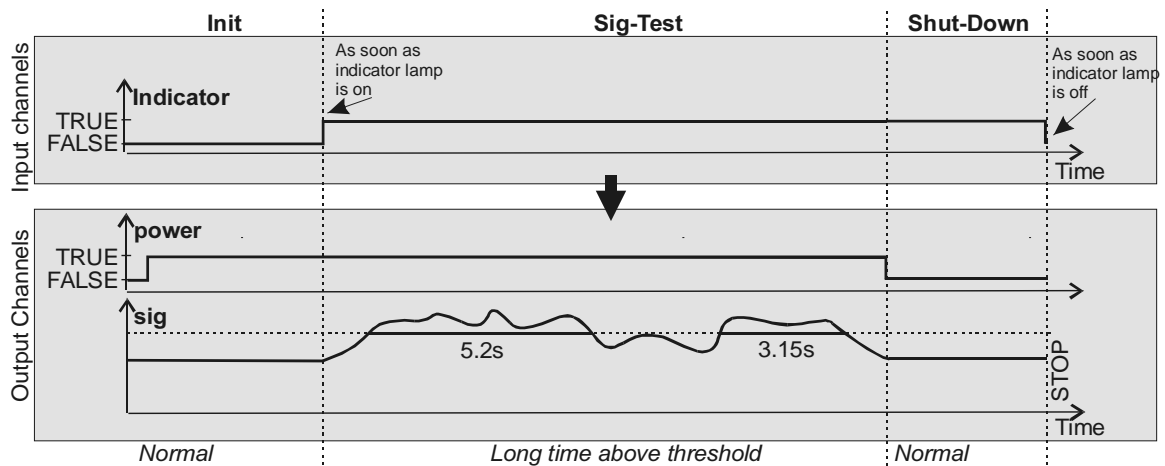


The state transition diagram shows that the temporal course of each test data is partitioned into three phases *Init*, *Sig-Test* and *Shut-Down*. Still, at this stage the description for the testlet is not sufficiently precise. Neither is there a precise description of (1) *how* signal courses of the channels are defined within each phase, nor (2) *when* phase changes occur.

(1). For every state in the state transition diagram a particular testlet (called *state testlet*) is given which describes all relevant variants of courses for this state. If, for example, the initialisation for the test is secondary, a state testlet for *Init* with only one element could be selected, describing a "normal" signal course *sig* and the "turn-on behaviour" of *power*. Furthermore the testlet from example 3 could be used as the state testlet for *Sig-Test* (though it needs to be extended by the additional output channel *power* which constantly equals "On" for all variants because the system must be turned on during *Sig-Test*). In the same way *Shut-Down* could be described, for example, with a testlet containing two elements (the courses of which are not discussed here).

For deriving a test data for the over-all course from this information a testlet element has to be selected from each state testlet. In the example scenario described above there are $1 \cdot 3 \cdot 2 = 6$ possible combinations to do so because the state testlets have one (for *Init*), three (for *Sig-Test*), and two (for *Shut-Down*) elements. A combined over-all testlet element can automatically be presented in a diagram as follows.⁴

⁴ Because for a testlet the output signal courses depend on the input signal courses a "prototypical" input course needs to be given for the graphical presentation. This enables that output signals can be derived from there and be displayed.

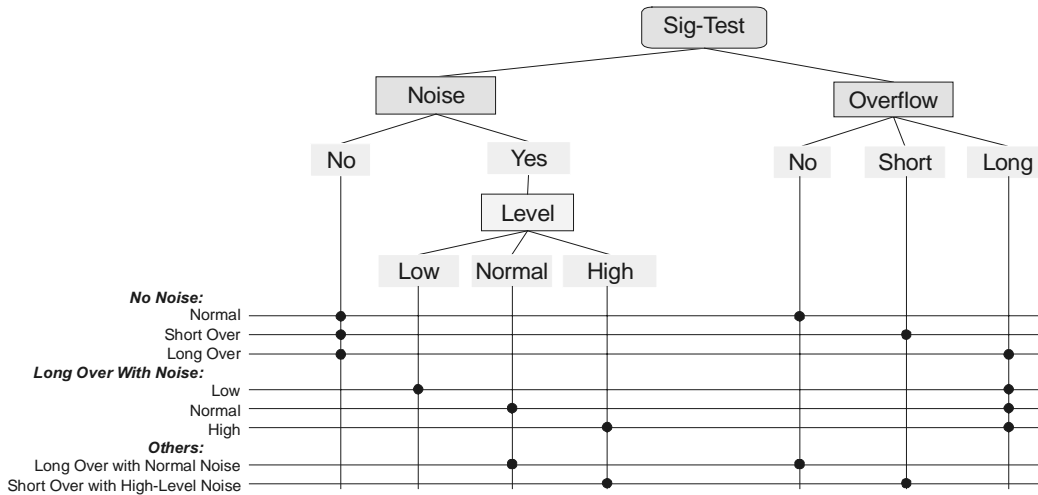


(2). The signal course within each phase is defined with state testlets, but not the points in time when phases change. Therefore each transition is backed with a formal condition which, for reasons of clarity, is not graphically mapped in the state chart. Each transition can fire for two reasons. Either the formal condition occurs or the testlet element of the previous state terminates (i.e. time of termination t_{stop} has been reached). The formal condition (`indicator == TRUE`), for example, is attached to the transition *Init*→*Sig-Test*, i.e. the state change occurs when *indicator* is set. The condition for transition *Sig-Test*→*Shut-Down* is `FALSE`, i.e. the condition is never met. The transition fires only when the state testlet element of *Sig-Test* terminates (reaches t_{stop}). Hence, the transition can be made dependent on “how long” *Sig-Test* needs for a concrete test data.

Consequently, the over-all testlet is completely and precisely describable with the definition of state testlets, formal conditions for transitions, and the selected combination of state testlet elements to over-all testlet elements.

Example 5 (Data Partitioning): The system described in example 4 shall now use a sensor for monitoring the signal *sig*. This sensor occasionally makes a high frequency noise. The system is meant to identify the noise and compensate it so that it has no influence on the monitoring of threshold overflows. Because for the test the signal *sig* is only relevant in the state *Sig-Test* anyway, we will consider in the following only the testlet for this state. Now two different aspects need to be taken into consideration in respect to *sig*: the duration of the threshold overflow and the noise of the sensor signal. Both aspects are relevant to be considered in different combinations. Obviously, in this case the problem is not about partitioning the temporal sequence of a test data but about the complexity of interface data regarding various aspects. Here, *Data Partitioning* which is based on the classification tree method [2,3] will be applied. The classification tree method is especially suited for dealing with the combinations of different aspects. For each aspect single *classes* will be determined. These classes describe potential test-relevant variants of the aspect that partition the input data space disjointedly and completely. The classes may be hierarchically refined.

By using *Data Partitioning* the test problem is described using a classification tree and a corresponding combination table as shown in the figure below.



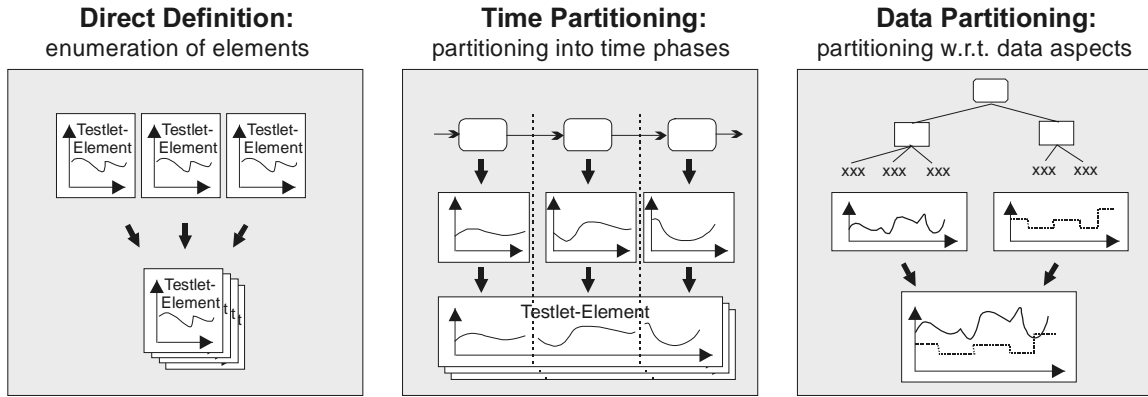
The aspect *Noise* is divided into different classes *No*, *Low*, *Normal* and *High* (where the latter three will be grouped in the joint class *Yes*). The aspect *Overflow* includes the classes *No*, *Short* and *Long*. With the classification tree method the classes can be easily combined. Each line of the table below the tree corresponds to a test data and all lines describe the sought over-all testlet. The marks in the lines identify the classes which apply to the respective test data. Let the first test data for example be sig without noise within normal range (no overflow).

For semantically precise test data of the testlets, all classes will be described through single-element testlets (called *class testlets*) which generally determine the signal course for just *one* channel. By combining several classes the definitions of *several* channels will be combined. In the example above each class testlet of classes *No*, *Low*, *Normal* and *High* describes a channel sig_{noise} for the noise of the sensor signal; the class testlets of *No*, *Short* and *Long* describe a channel sig_{raw} in order to determine the overflow. In each possible combination of noise and overflow the actual sensor signal is defined for the over-all testlet as $sig = sig_{raw} + sig_{noise}$. Therefore the definition will be globally assigned to the root class *Sig-Test* of the tree.

Subsequently, the over-all testlet for the phase *Sig-Test* describes three output channels $K_O = \{sig_{raw}, sig_{noise}, sig\}$, which are independent from any input channel (i.e. $K_I = \emptyset$). With the definition of testlets for single classes and the combination in the classification table the sought over-all testlet is completely and precisely describable.

4 Techniques in Detail

The objective of testing dynamic functional behaviour with the *Time Partition Testing* method is to construct a testlet with which the system can be thoroughly tested. As the examples above have demonstrated, *Time Partition Testing* provides three techniques for constructing a testlet: the *Direct Definition*, the *Time Partitioning* and the *Data Partitioning*. The two latter mechanisms are recursive procedures. They divide the over-all test problem into smaller sub-problems and put the testlets which have been found as the solutions to those sub-problems together into the sought over-all testlet. Depending on the specific test problem these techniques are applied individually or recursively combined. The following overview schematically shows the techniques.



The following sections describe each of these techniques in more detail.

4.1 Direct Definition

The *Direct Definition* is primarily used with “trivial problems”. The testlets usually contains less than five elements and all of them test the same situation with varying concrete data (e.g. small variance of frequency or amplitude in sinusoidal oscillations). The definition may be presented graphically or analytically. A graphic definition, as in example 3, is descriptive but it is only possible if $K_I = \emptyset$, i.e. if output courses and time of termination are independent from input channels.

If an output course or a time of termination depends on one or more input channels, it is possible to use analytical terms (e.g. $out(t) = in(t) + 17$ or $out(t) = in(\frac{t}{2}) + in_2(t)$). However, the value of an output channel at time t must not depend on the value of an input channel at a later time $t' > t$ in order to guarantee computability of the testlet (consequently $out(t) = in(t + 1s)$ is impossible). Formally this means that for each testlet element $e: \tilde{i}_1 \times \dots \times \tilde{i}_n \rightarrow Time^\infty \times \tilde{o}_1 \times \dots \times \tilde{o}_m$ and for each time $t \in Time$ also the mapping $e^t: \tilde{i}_1^t \times \dots \times \tilde{i}_n^t \rightarrow \tilde{o}_1^t \times \dots \times \tilde{o}_m^t$ reduced to the time interval $[0s, t]$ is a well-defined function where \tilde{k}^t describes the reduced mapping $\tilde{k}|_{[0s, t]}$ for a channel k .

The description language for analytical definitions guarantees this semantic limitation. It will not be discussed further in this paper.

4.2 Time Partitioning

In *Time Partitioning* the temporal course of a test data will be broken down to sub-phases. In order to achieve this a state transition diagram will be used. Branches in the state transition diagram characterise alternative potential sequence paths for the test data. In example 4 the state transition diagram contains just one path. This implies that all test data comply with this path. In general, in *Time Partitioning* first those paths which are relevant to the test have to be selected from the set of all possible paths. A path is a sequence of transitions from the initial state to the terminal state, and a path must not contain two transitions with the same source state, i.e. the sequence of the states passed for one path is deterministic. For each selected path the test elements of those states located on the path will be combined into over-all testlet elements as described in example 4 above.

In case a test data is supposed to have a non-deterministic branch, simple paths through the state transition diagram are not sufficient because their run through the single states is always deterministic. Therefore there are so-called *multi-transitions* which describe non-deterministic branches. A multi-transition is a set of elementary transitions with a common source state and (generally) different target states. Each of the elementary transitions is an alternative to the branching. A path can only contain a complete multi-transition, i.e. with all its elementary transitions. The condition for an elementary transition that occurs first defines the following state. In state transition diagrams multi-transitions are made easily identifiable

by an exit of the source state common to all transitions which belong to the multi-transitions (see Fig. 3).

By introducing and graphically highlighting multi-transitions the number of potential paths is kept relatively low. In Fig. 3 there are only two possible paths; however, without multi-transitions the illogical path "after 3s"/"without loop: ..." would be theoretically possible.

Multi-transitions also allow the modelling of loops (see Fig. 3). As long as the condition for termination "until flag is set" is not fulfilled (i.e. as long as the flag is not set) the cyclic transition switches after 3 seconds. Consequently the loop state *LoopOp* will be repeated every 3 seconds until the flag is set. *Time Partitioning* using multi-transitions is able to express complex temporal facts.

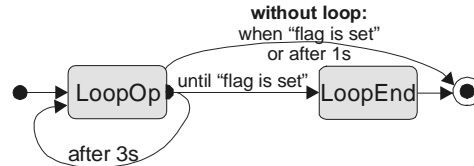


Fig. 3: Representation of multi-transitions

With *Time Partitioning* it is also possible to model events. Within a state continuous channels with signal courses are considered, whereas events can be fired at the exact time of state changes. In this case besides a formal condition an action can also be assigned to each transition. Events will be semantically modelled as channels whose signal courses can only be defined pointwise at state transition points. So, if at the exact time of a state change the event e shall be fired transferring the value γ , it is sufficient to assign the action $e := \gamma$ to the corresponding transition. The channel e is then defined exactly at transition point t_s i.e. $dom(e) = \{t_s\}$.

With this technique the *Time Partition Testing* supports the definition of event-driven test data, e.g. for protocol tests. It is for example possible to convert test data from TTCN[6,7] to test data represented using *Time Partition Testing*.

4.3 Data Partitioning

In *Data Partitioning*, as shown in example 5, several aspects for one and the same time interval are taken into consideration. Generally these aspects refer to different channels. If several aspects describe a channel in combination local channels will be introduced which then will be used jointly for the global definition of the actual channel. In example 5 the channel sig has been defined on the basis of sig_{raw} and sig_{noise} . Such channel may be marked *local* in order to limit their scope to the current classification tree.

Each line in the combination table below the classification tree determines a test data which is defined by merging the testlet elements belonging to the marked classes. If a channel is simultaneously an output channel of one testlet element and an input channel of another testlet element, the dependency needs to be acyclic and has to be unambiguously dissolvable. The time of termination of the over-all testlet element is defined as the smallest time of termination of the defined testlet elements.

In example 5 each class is trivial, i.e. the class testlets are always singletons. If the classes are more complex – e.g. when a class need to be partitioned into phases according to *Time Partitioning* – classes may contain more than one element. If this is the case, the set of elements will be automatically dissolved by refining the class according to the class testlet elements, i.e. the class will be partitioned so that there will for each testlet element a sub-class with the same name. Consequently, the class testlets of the sub-classes are singletons and can be selected with help of the combination table as demonstrated in example 5.

5 Automated Test Execution

The formal semantics of testlets allow the test to be executed automatically. For automation it is necessary to generate a test driver that automatically performs the execution

of testlet elements (test data). The architecture of the test driver and the communication between test driver, system, and environment depend on the system under test. This means, the generation of test drivers depends on the platform whereas the definition of testlets using *Time Partition Testing* is platform-independent.

Testlets are based on a dense temporal model. Necessarily, for execution there has to be a time discretisation. Resulting errors in discretisation generally have no practical impact because usually the executability of tests was regarded already during modelling of test data. Theoretically, problems may arise for example when a condition of transition occurs only for a very short interval between two discrete times and therefore can not be recognized during execution. The higher the sampling frequency the more errors in discretisation move into the background.

6 Test Evaluation

Generally, a fully automated test evaluation is not possible because a complete definition of the desired behaviour is costly. Though, as a rule, it makes sense to install watchdogs which signal violation of specific requirements. When a watchdog reports a violation a failure has been found. Still, it is possible that a failure might have occurred though no violation can be found.

Watchdogs may be represented semantically by special output channels. Then for example a value overflow can be checked by a watchdog $w_1(t) := (x(t) > 127)$. If w_1 is *true* at any time, a value overflow occurs at this time. As another example, if in *Time Partitioning* a specific transition is supposed to signal an error (e.g. a timeout condition) an action $w_2 = \text{true}$ may be attached to this transition. If the channel w_2 is *true* at any time, the timeout has occurred. This is a way to characterise watchdogs.

Another very powerful method for test execution is the *inverse matching* which supports regression testing. Here, signal courses of input channels are compared to reference courses which have been recorded in previous test executions. First, the processed states of the reference case will be compared to those of the current test. If they are identical, the signal courses are partitioned into single phases corresponding to the processed states. Subsequently, these sub-courses will be compared. So, both courses (reference and current test) are “synchronised” according to the states of the test data. This is a way to compensate temporal differences. The signal courses within a single time sequence will be compared with view to similarity criteria (e.g. acceptable maximum tolerance).

In correspondence with the similarity criteria the quality of a measured signal course can be evaluated relative to the reference course. By this means, especially in later software development phases where regression testing plays a major role, the test becomes enormously efficient. The field of inverse matching is still subject to substantial research within our group.

7 Conclusion

Time Partition Testing focuses on the definition of test data which describe temporal courses of channels. Testlets provide the underlying semantic model of these courses. There are three basic syntactic techniques for the definition of testlets: the *Direct Definition*, the *Time Partitioning*, and the *Data Partitioning*, the latter being recursive procedures.

Time Partition Testing supports the modelling of continuous, partial signals which can also be used for the modelling of events. The option to define signals in dependence of input data facilitates a very natural modelling of test data and provides a formal framework for the test evaluation.

Due to its variety of techniques and the automation capabilities, *Time Partition Testing* requires a tool support which is currently under development. This tool supports modelling of test data as well as the test execution and evaluation.

8 References

- [1] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons Inc., New York, 1995.
- [2] Klaus Grimm. *Systematisches Testen von Software – Eine neue Methode und eine effektive Teststrategie*. GMD-Bericht Nr. 251. R. Oldenbourg Verlag, München/Wien, 1995.
- [3] M. Grochtmann. *Test Case Design Using Classification Trees*. In *Proceedings of STAR'94*, pages 93-117, Washington, DC, May 1994.
- [4] G.J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, 1978.
- [5] A. Gill. *Introduction To The Theory Of Finite-State Machines*. McGraw-Hill, 1962.
- [6] ETSI TC MTS. *Guide for the use of the second edition of TTCN (Revised Version)*. European Guide 202 103, 1998.
- [7] T. Walter, J. Grabowski. *Real-time TTCN for Testing Real-time and Multimedia Systems*. In *Testing of Communicating Systems*, volume 10, Chapman & Hall, 1997.
- [8] D. Harel. *Statecharts: A visual formalism for complex systems*. In *Science of Computer Programming*, pages 231-274, 8, 1987.
- [9] W. Perry. *Effective Methods for Software Testing*. John Wiley & Sons, Inc. New York, 1995.