TestML – A Test Exchange Language for Model-based Testing of Embedded Software

Jürgen Großmann[1], Mirko Conrad[1], Ines Fey[1], Alexander Krupp[2], Klaus Lamberg[3], Christian Wewetzer[3]

[1]DaimlerChrysler AG, Alt Moabit 96a, D-10559 Berlin
[2]C-LAB, Fürstenallee 11, D-33102 Paderborn
[3]dSPACE GmbH Technologiepark 25, D-33100 Paderborn

{juergen.grossmann | mirko.conrad | ines.fey}@daimlerchrysler.com ; krupp@c-lab.de ; {klamberg | cwewetzer}@dspace.de

Abstract. Test processes in the automotive industry are tool-intensive and affected by technologically heterogeneous test infrastructures. In the industrial practice a product has to pass several test levels such as Model-in-the-Loop- (MIL), Software-in-the-Loop- (SIL) and Hardware-in-the-Loop- (HIL) tests. Normally, different test systems are applied for this purpose and almost each test system has its individual requirements on the test description and often requests a proprietary test description language. As a result, whole test specifications are created as an assembly from a variety of different description languages. Efforts to integrate these heterogeneous specifications, to address test exchange in a general manner and to standardize and harmonize the existing language environment are still at the beginning and not tailored towards the requirements of the automotive domain. To keep the whole development and test process efficient and manageable, the definition of an integrated and seamless approach is required. TestML – the test exchange language presented in this contribution – is defined to overcome the technological obstacles that almost automatically accompany the application of heterogeneous test tools and test infrastructures. TestML supports the exchange of different test notations in a heterogeneous tool environment. By defining an XML-based interchange format spanning different tools, TestML offers a common basis. Individual, tool-specific notations can be mapped here through the realization of appropriate import/export adapters. In this paper, we introduce the XML schema of TestML and demonstrate the efficiency of the interchange format by giving examples from the model-based development of electronic control units.

1. Model-based testing in the automotive industry

Model-based testing of embedded vehicle systems has gained increasing significance in recent years. The application of model-based specifications in development and the establishment of powerful code generators have led the development process to be noticeably more effective, automated, and reach a higher level of abstraction. Due to the availability of executable model specifications, tests and analytical methods can al-

ready be begun at an early point in time and be integrated into development subsequently. The positive effects are obvious. Early error detection and early starting the necessary bug fixing becomes possible.

To keep the whole model-based development and test process efficient and manageable, the definition of an integrated and seamless approach is required. For this purpose, the BMBF project IMMOS (Integrated Methodology for Model-based ECU Development) was carried out by DaimlerChrysler AG, dSPACE GmbH, IT Power Consultants, Fraunhofer FIRST, FZI Karlsruhe and the University of Paderborn. **TestML** – the test exchange language presented in this contribution – is a substantial project result. Defined to overcome the technological obstacles that almost automatically accompany the application of heterogeneous test tools and test infrastructures in model-based development **TestML** supports the interchange and reuse of tests and a seamless test development.

2. TestML basics

TestML is an XML-denoted language that is independent from tools and that was developed for the interchange of test descriptions. It was tailored specifically to meet the demands of model-based testing of embedded software in the automotive sector. The language covers the different test stages from the module to integration and system tests as well as test levels from MIL to HIL. Besides describing strictly functional tests, different comparative test approaches such as regression testing and back-to-back testing are supported. Our aim is to realize an interchange format for the large spectrum of test description languages established in the automotive industry. This chapter describes the purpose of **TestML** and expounds on the background and the demands of the language design. Chapter 3 addresses the set-up and structure of the language. Chapter 4 illustrates the description of different kind of test behavior with **TestML**. Chapter 5 summarizes the paper.

2.1 An unified format for test exchange

Our basic assumption is that semantic similarities as well as overlaps exist between the different test description languages (see [2] for an overview). On the one hand, these similarities represent the indispensable precondition for the idea of interchanging test scenarios and test data across tool and language barriers. On the other hand, they offer the necessary foundation for the definition of a generic exchange format.

TestML was conceived as a language for exchanging test descriptions in the context of model-based testing of embedded automotive software. Language requirements thus arose from the domain-specific demands on the one hand and the general needs of an interchange language on the other. The portrayed general conditions of testing control units constitute a series of specific demands that have to be made on a technology-independent test description language spanning different tools in the automotive domain:

- support of the specification of discrete and continuous (analogue) stimuli

– a concept of time to describe time-dependent events and sequences
– the ability to specify reactive test cases
– the description and management of measurement data as inputs as well as reference data for comparative tests
– expressions for tests evaluation regarding the analysis of discrete and continuous signals.

The Basis of **TESTML** is a self-contained language definition that makes it possible to depict test descriptions of different levels of abstraction (such as test scenarios and test data) independent from the respective tool environment.

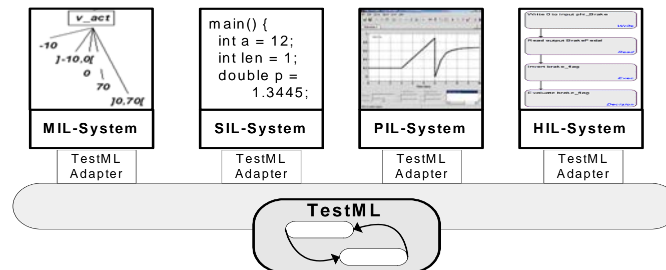

Fig. 1. Integration of different test descriptions by use of an intermediate language

The integrating effect of the language results from the potential to map the language constructs of existing test description languages to **TESTML** and vice versa by using appropriate adapters. **TESTML** itself acts as an intermediate notation that is interposed between the separate tool-dependent languages in the exchange of test data and test descriptions (Figure 1). The advantages are obvious. If multiple languages are to be supported, the complexity of integration increases only linearly for this solution. If integration is achieved through the realization of bilateral, point-to-point coupling without an intermediate format instead, the complexity increases exponentially.

2.2 Abstraction of specific test systems

Complex test systems are used for the testing of control units. Test systems usually consist of multiple logical components (signal generators, capture/replay tools, test evaluation components, environment models etc.) that have to be coordinated and collectively controlled in the course of test execution. Set-up and control of test systems differ contingent. In addition to the heterogeneous tool environment different test notations exist that can be used to describe the tests. For an evaluation and categorization of existing test notations for model-based software testing of embedded software systems see [2].

In its function as an exchange language, TestML is supposed to be able to operate on different test systems. Because individual test systems differ greatly in their concrete technical specifications, it must be possible to abstract from the concrete realization of the respective source and target system for the exchange of test descriptions. Thus, we defined one such abstraction termed **TESTML** test system. A **TESTML** test system consists of a combination of test components that we consider minimally neces-

sary regarding the exchange of test descriptions. The individual components of the test system subsequently given below are, except for the test interface, represented implicitly by TestML means of description. The TESTML test system itself is not an explicit part of the TestML language. Knowledge about set-up and structure of the system help to better understand the subsequent annotation of individual means of description in TESTML. Figure 2 shows a diagrammatic illustration of the TESTML test system including TESTML elements referring to the individual components.

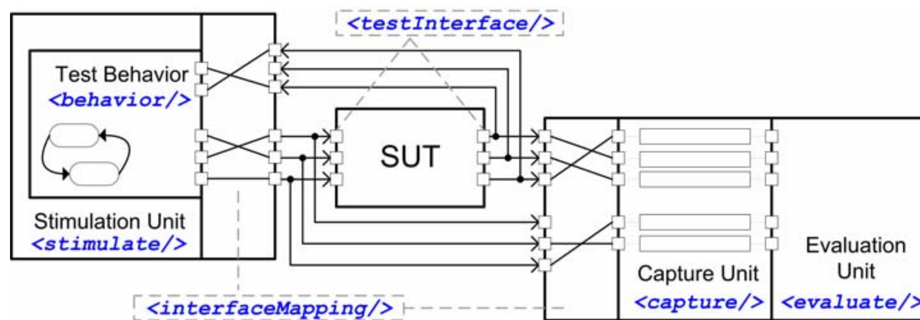

Fig. 2. Abstract test system for TestML

The abstract test system for TESTML consists of the following components:

– The system under test (SUT) represents the system that is to be tested. Mainly relevant for TESTML is its test interface. From the perspective of TESTML, the SUT itself disappears behind the test interface.
– The stimulation unit is responsible for the generation of test stimuli; actual test execution takes place here.
– The capture unit records the system reactions and/or the system reactions as well as the test stimuli.
– The evaluation unit is responsible for the evaluation of test cases. It accesses all data recorded by the capture unit and can be operated temporally independent from the stimulation unit.

2.3 XML representation

The TESTML language is represented in a XML-compliant notation. XML-compliant notations and formats were established as flexible instruments for data exchange as well as for different options of data storage, transformation, and retrieval. Many tools, also in the field of testing, already support data exports in the form of their own XML formats. The data necessary for an exchange is thus already available in XML and can easily be processed further. Additionally, a number of efficient concepts and languages exist for XML, particularly supporting the transformation and the access to XML data. The individual means of description of the TESTML language are represented by individual XML elements that are defined by an XML schema, as is common for XML notations. The complete XML schema for TESTML can be accessed in [12].

3. Structure and elements of a test description with TestML

This chapter describes the elements and the set-up of test descriptions with **TestML**. The most important basic elements of the language will be introduced individually and illustrated below. Each basic element represents an important concept and/or function from the field of testing and, as is common in XML, stands for a container that may contain further means of description and encapsulates them to the outside.

- The element **testML** forms the root element for each **TestML** description and represents a number of test cases. The element **testML** needs no further explanation from here on.
- The element **testInterface** serves to describe the interface to the SUT.
- The elements **testSequence** and **rtTestSequence** constitute test sequences that respectively describe an operating scenario that is to be tested by means of test inputs and outputs.
- The elements **stimulate**, **capture** and **evaluate** describe stimulation, recording, and test analysis within one test sequence.
- The element **behavior** is used for the specification of test behavior.

A number of other means of description exist besides the mentioned basic elements that represent either data types, operators and/or mathematical expressions, structure the language and/or form the inner structure of the above-mentioned elements. The following diagram shows the section of the **TestML** schema in which the basic elements are defined.

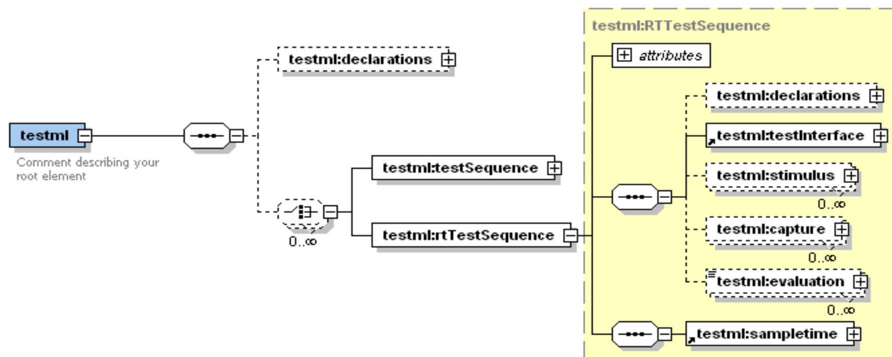

Fig. 3. The basic elements of a test description with TestML

3.1 The test interface

The test interface is described by the element **testInterface**. A large number of input and output channels represented by the element **port** in **TestML** are part of a test interface, ensuring typesafe communication between the SUT and the test behavior defined modularly in **TestML**. Values can be written to the SUT and/or read by the SUT through each defined channel. The type of communication can be specified in more detail by stating the direction of communication and the data types supported by the individual channel.

```
<testInterface ID="Testinterface_1">
  <port ID="P1_phi_Brake" type="double" direction="input"
   name="phi_Brake"/>
  <port ID="P2_phi_Gas" type="double" direction="input"
   name="phi_Gas"/>
  <port ID="P3_v_act" type="double" direction="input"
   name="v_act"/>
  <port ID="P4_BrakePedal" type="boolean"
   direction="output" name="BrakePedal"/>
</testInterface>
```

Listing 1. Specification of the test interface through the element testInterface

The listing depicts the description of a test interface with 4 channels in **TestML**. The direction of communication, the type and the name of the port are annotated in the form of XML attributes to the element **port**.

3.2 Test sequences

TestML represents test cases through the elements **testSequence** and **rtTestSequence**. To emphasize that test cases are usually complete application scenarios, they are identified as test sequences in **TestML**. While the element **rtTestSequence** can describe a real time test case, the element **testSequence** represents a non real time test case. Real time test cases and non real time test cases can be differentiated by the lack of temporal references within the element **testSequence**. The structural set-up of **TestML** test sequences otherwise remains the same.

A **TestML** test sequence usually consists of the elements **stimulate**, **capture** and **evaluate** as well as a modular behavioral description through elements of the type **behavior**. We will address this in more detail in chapter 3.5.

3.3 Stimulation, Recording and Evaluation

The specification of stimulation, recording, and evaluation in **TestML** is undertaken, with a strict conceptual separation, by the elements **stimulate**, **capture** and **evaluate**: Among other things, this separation is rooted in the separation of time- and resource-intensive evaluation operations necessary for real time tests that usually have to be carried out after the stimulation to ensure the necessary reaction times of the real time test system during stimulation.

The elements **stimulate**, **capture** and **evaluate** essentially have two functions. On the one hand, they serve as control commands for the abstract stimulation, capture and evaluation units defined in chapter 2.3. The control occurs implicitly, that is without the existence of explicit control commands. This means that for a concrete target system each element **stimulate**, **capture** and **evaluate** is interpreted as a number of platform-specific control commands. These are necessary to control the concrete test units provided by the concrete system.

On the other hand, the elements **stimulate**, **capture** and **evaluate** encapsulate the detailed expressions and statements of a **TestML** test description. The element **stimulate** contains the complete behavior specification to generate the required stimulus signals (see element **behavior** in chapter 3.5 for details). Moreover, the elements **stimulate** and **capture** contain mappings that map the input and output ports of a test interface (element **port**, see chapter 3.1) on the elements of a behavior signature (element **signature**, see chapter 3.5) or accordingly on capture variables. The mapping on a behavior signature is part of the stimulation description while mapping of capture variables is conducted in the element **capture**. Capture variables serve as references to access recording data and are later used within the element **evaluate** for test evaluation. Irrespective of it being used for stimulation or recording, mapping is specified through the element **interfaceMapping**. The following listing depicts mapping between channels of a test system and recording variables within an element **capture**.

```
<interfaceMapping>
  <map>
    <portRef IDREF="P1_phi_Brake"/>
    <signalRef IDREF="Sig1_phi_Brake"/>
  </map>
  <map>
    <portRef IDREF="P2_phi_Gas"/>
    <signalRef IDREF="Sig2_phi_Gas"/>
  </map>
</interfaceMapping>
```

Listing 2. Mapping of channels to capture variables

The element **evaluate** describes the test evaluation. Whereas the elements **stimulate** and **capture** are obligatorily started simultaneously at the beginning of test execution, the element **evaluate** can be started independent of the two other elements, even after test execution. In principle, test evaluation takes place based on the data recorded by the element **capture**. Test evaluation is carried out by specifically defined operators and commands, which are not discussed in this paper.

3.4 Data Types, Operators and Expressions

In most programming languages the basic data types are bool, integer, double, string. Naturally, they are supported by **TestML**. They may be represented according to their corresponding type as defined in the W3C XML schema, i.e. xs:integer for an integer. Also, test specific special data types are offered, such as time and signal. Times are given as a double value and a time unit. The possible units are day, hour, second, millisecond, and microsecond represented by their common abbreviations “d”, ”h”, “s”, “ms”, and “us”. The following listing shows the declaration of a double variable and, following the declaration, a write operation to set the variable to the value of 100. All declared variables may be referenced within **TestML** by their ID.

```
<double ID="Var1" name=" First examples Variable"/>

<write>
  <doubleRef IDREF="Var1"/>
  <value><double>
    <value>100</value>
  </double></value>
</write>
```

Listing 3. Instantiation of data types

The data type signal is special in that it may not only describe a singular value, but a time-dependent wave-form. Most of the times, the value data type of a signal is double. The wave-form is represented by means of simple signal expressions or **TestML** automatons, which are introduced in section 4. Listing 4 shows a simple signal expression specifying a ramp which rises from 0 to 100 within 10 seconds.

```
<signal>
  <time>
    <unit>s</unit>
    <double><value>10</value></double>
  </time>
  <ramp>
    <start><double><value>0</value></double><start>
    <end><double><value>100</value></double></end>
  </ramp>
</signal>
```

Listing 4. Instantiation of data types

For expression evaluation a set of simple operators is provided. The table below shows a non-exhaustive list of them. The four basic arithmetic operations are provided, as well as equality, comparison, and logical operators. The operators may be combined to form expressions as usual.

Operator	Meaning
<add/>	addition
<sub/>	subtraction
<mult/>	multiplication
<div/>	division
<abs/>	absolute value
<max/>	maximum value

Operator	Meaning
<and/>	logical and
<or/>	logical or
<not/>	logical not
<equ/>	equality
<grt/>	greater
<geq/>	greater or equal

Table 1. Non-exhaustive list of operators

An example of a conditional expression is shown below, which compares two signals with the identifiers “Data1_v_act” and “Data2_v_act”.

```
<cond>
  <grt>
    <signalRef IDREF="Data1 v act"/>
    <signalRef IDREF="Data2_c_act"/>
  </grt>
</cond>
```

Listing 5. Comparison Expression

3.5 Test behavior

The term test behavior subsumes processes that describe the stimulation of the test object at the moment of test execution. Focus of a behavioral description is ***how*** a test is carried out, that is which signals and messages have a stimulating effect on the test system at what point in time. Specification of test behavior is one of the main aspects of a test description and is an essential part of a test description language. In practice, a variety of different methods, notations, and tools exist for the description of test behavior. For a good synopsis for the automotive sector see [2].

This chapter describes the specification of test behavior with **TestML**. The first segment revisits basic demands to a test description language for the domain automotive. On the basis of criteria for the differentiation of diverse test behavior, we show the spectrum of behavioral descriptions that **TestML** has to cover. Subsequently, the concepts and basic descriptive means of **TestML** that can be used for a behavioral specification are introduced.

3.5.1 Support of different types of test behavior

To serve its function as an exchange language, **TestML** has to cover and integrate the widest possible spectrum of different behavioral descriptions. In this regard, we already defined some demands in chapter 2. Below we will specify a series of different criteria that can serve as the basis for the differentiation of varying classes of test behavior and fundamentally characterize the spectrum of **TestML** behavioral descriptions. The criteria are:

1. **Types of stimuli**
   The type of stimuli used for stimulation represents a basic criterion of differentiation in the stimulation of a test object. Four different types of stimulation signals are differentiated in literature ([5], [2]). Relevant at this point is the differentiation of *timed signals* and *timeless messages*.
2. **Determination**
   Test behavior can be specified as *reactive behavior* or as *determined behavior*. A test with reactive test behavior controls and changes the stimulation of the test object depending on how the test object reacts to the continuous stimulation. To do so, the output signals or output messages of the test object are interpreted, evaluated, and considered for the generation of new stimuli. In determined test behavior, the stimulation of the test object is established from the outset. The reaction of the test object is not considered for the generation of stimuli or is used only to abort the test at an appropriate time.

3. **Data synthesis**
   Another criterion for test behavior is the differentiation of *synthetically generated stimulation sequences* and recorded *measured data* that can later be replayed for test purposes. Synthetically generated stimulation sequences are usually described through programming techniques. Apart from the stimulation of a test object, recorded data is mainly used as a reference for test evaluation. Comparative approaches like regression testing or back-to-back testing explicitly require the existence of recorded reference data.

3.5.2 Description of test behavior with TestML automatons

As mentioned before the element **behavior** encapsulates complete behavior specifications by defining a hybrid automaton. Hybrid automatons emanate the theory of hybrid systems and are regarded as a mathematical model, which offers a reliable basis for modeling timed applications and systems with discrete and analog behavior (cf. [3]). The use of hybrid automatons to describe continuous test behavior in the context of embedded systems could be successfully shown in [9].

The structure and elements of the element **behavior** are depicted in Figure 4. They all together form a so called TestML automaton. The most important elements of a TestML automaton are the elements **signature**, **step** and **switch**.

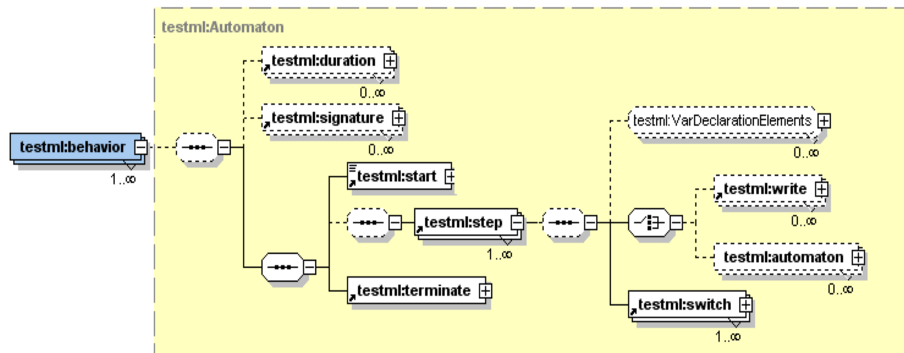

Fig. 4. The element behavior

The element **signature** provides an interface to the internally specified and encapsulated behavior. The element itself consists of a number of signal declarations (much like the ports in the test interface) that can individually be accessed in reading or writing depending on the specification.

The element **step** describes a defined state within a **TestML** automaton. Each **step** in turn is either defined by a **TestML** automaton or contains one or more commands that define test behavior directly – either through arithmetic equations, time-based signal primitives or alternatively for timeless messages. The element **switch** defines a transition that marks the passage between two **steps**. By use of the element **cond** every transition can be annotated with certain conditions, making a time- or value-dependent control of automatons possible. If the transition condition evaluates to true, the transition fires and the **step** referenced by the element **succ** will be processed

next. For each step one transition without condition is allowed. A transition without condition fires after all time-dependent instructions that are defined within the respective step have been terminated.

4. Exemplary use of TestML automatons

In the following, the description potential of **TestML** is shown and illustrated by means of short examples taken from practice. The samples selected each represent a specific type of test behavior mentioned in chapter 3.5. In the following examples, the **TestML** automatons are not depicted in their XML representation but as annotated UML statecharts[1].

4.1 Specification of timed, deterministic test stimuli

We will now examine a typical test sequence taken from a cruise control test. The focus of the test is on accelerator pedal interpretation, i.e. the unit which is responsible for interpreting the driver interaction via the brake and the gas pedal. For this example, the test interface was deliberately kept small. The port `v_act` describes the current vehicle speed in m/s, `phi_Acc` represents accelerator pedal travel and is given in % and `phi_Brake` represents brake pedal travel and also given in %.

In order to test the acceleration pedal interpretation the following timed test scenario is used:

1. Within the first second, the current vehicle speed is kept constantly at -10 m/s. and afterwards the value for `v_act` is set to -5 m/s for a second.
2. In the course of the test, the accelerator pedal travel is raised from 0% to 100% and then lowered linearly from 100% to 0%.
3. In the course of the test, the brake pedal travel is linearly lowered from 100% to 0% and then raised from 0% to 100%.

The following chart shows both the individual signal forms and the description used for the generation of signal forms with **TestML**.

[1] We deliberately avoid the XML representation, since this quickly becomes too large even for short examples. The use of UML statecharts makes a more compact visualization possible. The TestML element **step** is depicted as a state and the element **switch** is presented as a transition of the UML automaton. Statements which are used within the element **step** for the generation of signal primitives or messages are annotated in the form of pseudo code inside the states. For further information on XML representations, please refer to the enclosed schema..

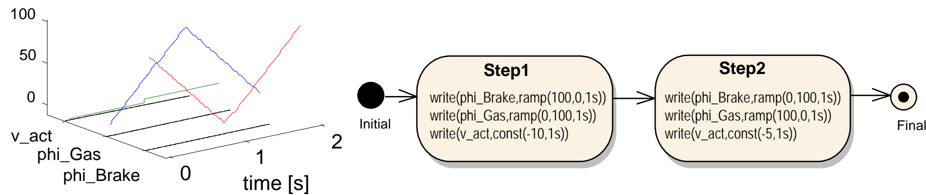

Fig. 5. Specification of synthetic stimulation sequences with TestML automatons

Every state of the **TESTML** automaton depicted above defines a period of time, with its length being determined by the duration of instructions within the respective state. The example mentioned above contains for the state Step1 the following three instructions:

– Write a ramp signal with the value course from 100 to 0 and the length of 1 second on the channel called `phi_Brake`.
  `write (phi_Brake, ramp(100, 0, 1 s))`
– Write a ramp signal with the value course from 0 to 100 and the length of 1 second on the channel called `phi_Gas`.
  `write (phi_Gas, ramp(0, 100, 1 s))`
– Write a constant signal with the value -10 and the length of 1 second on the channel called `v_act`.
  `write (v_act, const(-10, 1 s))`

After the statements have been executed a change into the next state, called Step2, takes place via the output transition. For the state Step2 we have the following three instructions:

– Write a ramp signal with the value course from 100 to 0 and the length of one second on the channel called `phi_Brake`.
  `write (phi_Brake, ramp(0, 100, 1 s))`
– Write a ramp signal with the value course from 0 to 100 and the length of one second on the channel called `phi_Gas`.
  `write (phi_Gas, ramp(100, 0, 1 s))`
– Write a constant signal with the value -10 and the length of one second on the channel called `v_act`.
  `write (v_act, const(-5, 1 s))`

The execution stops as soon as the final state has been reached.

4.2 Specification of timed, reactive test stimuli

In order to be able to show the use of **TESTML** automatons for the specification of reactive test behavior, the example mentioned above needs to be changed. Here, it is not the pedal interpretation which is tested, but the cruise control. The test interface is expanded by an input and an output channel. The port `v_Ziel` is an input port and describes the desired vehicle speed; `v_Fzg` is an output port and represents the current vehicle speed.

The reactive test behavior may be described as follows:

1. Set target speed `v_Ziel` at 18 m/s
2. Use gas pedal until vehicle speed is greater than or equals 18m/s.

3. Switch on cruise control.
4. Use brake pedal until vehicle speed equals 0 m/s.
The following chart shows the individual signal forms (left-hand side) as well as the description for the generation of the signal forms with (right-hand side).

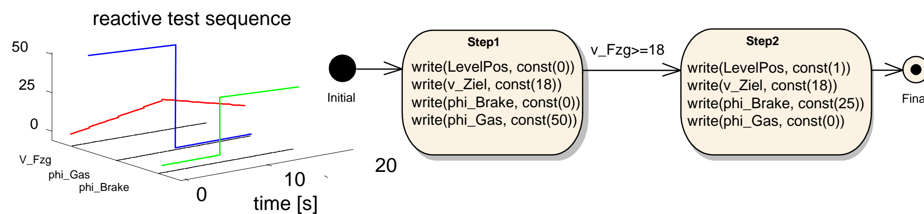

Fig. 6. Specification of reactive stimulation sequences with TestML automatons

In contrast to the example from the previous chapter, the transition between the states Step1 and the state Step2 is equipped with a condition. The condition defines the switching characteristics between Step1 and Step2. The instructions are executed until the transition condition for Step2 is fulfilled. Then the instructions from Step2 are e-xecuted.

– Write a constant signal with the value 50 on the channel called `phi_Gas` and a constant signal with the value 0 on the channel called `phi_Brake`. The cruise control is switched off.
```
write (LevelPos, const(0))
write (v_Ziel, const(18))
write (phi_Gas, const(50))
write (phi_Brake, const(0))
```
– If the channel `v_act` has taken on a value greater or equal 18 m/s, write a constant signal with the value 0 on the channel called `phi_Gas`, a constant signal with the value 70 on the `phi_Brake` channel and switch on the cruise control.
```
write (LevelPos, const(1))
write (v_Ziel, const(18))
write (phi_Gas, const(50))
write (phi_Brake, const(0))
```
With a basic set of signal primitives and their suitable combinations, the use of TESTML automatons makes it possible to write on signal forms however complex.

4.3 TestML Automatons for the specification of timeless test stimuli

Apart from systems working with timed test stimuli, there are frequently systems found in practice which are controlled solely by timeless stimuli, so-called messages. A test description for such a system consists of a set of actually timeless messages, which – by all means in a given order – are sent to different channels of the test system. The message sequence depicted below stands as an example for the discontinuous test case description, as it can be used in this case for the AutomationDesk tool from sSpace [6] company. Again, a cruise control function is tested. Besides the already known `phi_Brake` input the brake pedal flag `BrakePedal` is also read.

A possible hyteresis of the brake pedal recognition is tested. First, a rising edge from 0 to 100 for the `phi_Brake` input is created and then the `ped_min` value is set.

The ped_min value is the highest value in which the BrakePedal output again takes on the value 0, provided that there is no hysteresis.
The test description to be represented in the TestML looks as follows:
1. Writing 0.0 to phi_Brake
2. Waiting for time_step seconds
3. Writing 100.0 to phi_Brake, this way a rising edge from 0.0 to 100.0 is created
4. Waiting for time_step seconds
5. Calculation of ped_value according to ped_value = ped_min - tol
6. Writing of ped_value to phi_Brake
7. Reading BrakePedal and saving of the value in the brake_flag variable

The following automaton shows the implementation of the simulation by a TestML automaton. Reading the model output (last point of list mentioned above) is carried out by the capture element, which will not be depicted at this point.

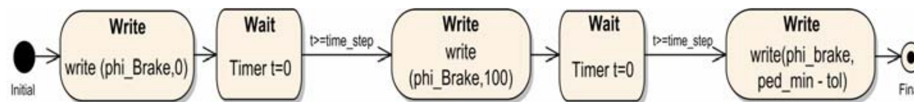

Fig. 7. Specification of message based test sequences with TestML automatons

Individual states of the automaton describe the activation of individual messages. Because of the use of timers, which can be difined locally on the automatons, the transitions between the states are time-controlled and define temporal distances between messages.

4.4 Storage and recording of data streams (measured data)

Apart from the specification of synthetic simulation sequences described in the previous chapter, TestML automatons can also be used for storing, managing and exchanging recorded data. For this, it does not make a difference, whether the data is used as stimulation sequences or as a reference for the test evaluation.

Continuous measuring and reference data are sampled and discretized, i.e. they are noted as a continuous flow of discrete values, which are assigned to a specified time period t with the same length of time Δt, the so-called sampling rate. During synthetic simulation of simple stimulation sequences by means of TestML automatons, which we have described in the previous chapters, something similar happens. The states of automatons form periods of time, whose length Δt is defined either explicitly for every state or implicitly by the length of the longest instruction. It is thus possible to define time periods with TestML automatons, which contain a defined length Δt. If we examine simple TestML automatons, i.e. automatons without loops and with states with only one outgoing transition each, every automaton represents one sequential composition of defined periods of time. A suitable instruction makes sure that exactly one value is assigned to every time period. If the length Δt is set for all states to one and the same value we have the requirements necessary to describe the sampled measuring and reference data with TestML automatons. The following chart shows a simple measured value, which has been written down as a TestML automaton.

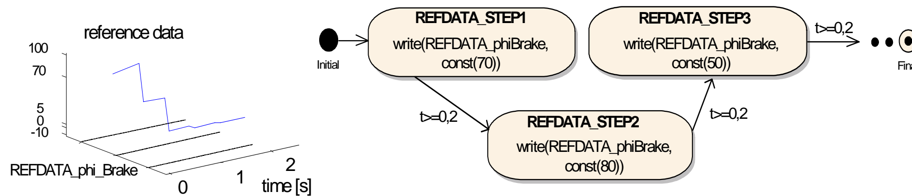

Fig. 8. Notation of scanned/sampled data streams with TestML automatons

For time control, `Timer t` is used, which is initialized for every state with the value 0. If `Timer t` reaches the value for the sampling frequency, the next state is used.

5. Summary and outlook

TestML is a XML-noted test exchange language that is fitted for the requirements of model-based testing of embedded vehicle software throughout the course of development. The reason for this development can be found in the need of defined tests which could be reused during the whole development process, both in different test phases and different test environments as well as in the exchange between suppliers and the OEM. Thus, the aim of the language design was to be able to map a spectrum as broad as possible of the test description languages established in the automotive industry. With this, an exchange of tests between different tool platforms for the MIL, SIL and HIL test is made possible.

As an extension to the majority of test description means prevalent, **TESTML** provides language constructs for tests under real time conditions. Test scenarios capable of real time use are an important functionality of HIL test beds. If one now wishes to cover the entire development process with the test exchange language, the corresponding description means must be provided.

Special importance was attached to flexible possibilities for test behavior description. The concept of hybrid automatons used to capture test behavior permits the mapping of common classes of automotive test descriptions, among them deterministic and reactive test stimuli with or without temporal references as well as the use of recorded data streams as they accrue out of test drives. The decision to map all test aspects on automatons made it possible to avoid an overloading of the exchange language with manifold constructs, which ultimately would have led to a superset of existing means of description. In order to more precisely define the semantics of hybrid automatons used in **TESTML** it is planned to define a mapping to a widely accepted automaton notation, e.g. Stateflow statecharts [11].

TESTML supports, besides classical functional tests, also comparative test approaches, such as regression testing and back-to-back tests, which are based on the existence of recorded reference data.

At this moment, in the frame of the IMMOS project (www.immos-project.de), a **TESTML**-based exchange of test descriptions between the MIL/SIL test environment MTest and the HIL test tool AutomationDesk is realized prototypically. In this context we would like to thank our participants in the IMMOS project for their contribu-

tion to TESTML, especially S. Sadeghipour and H.-W. Wiesbrock from ITPower Consultants, Prof. H. Schlingloff and M. Friske from Fraunhofer/FIRST.

6. References

[1] SCC20 ATML Group: IEEE ATML Specification Drafts and IEEE ATML Status reports; (2005) Available at http://grouper.ieee.org/groups/scc20/tii/
[2] M. Conrad: Modell-basierter Test eingebetteter Software im Automobil; Deutscher Universitätsverlag/GWV Fachverlage GmbH, Wiesbaden 2004
[3] R. Alur, C. Coucoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho,et al. The Algorithmic Analysis of Hybrid Systems. Theoretical Computer Science, 138:3–34, 1995.
[4] E. Broekman, E. Notenboom: Testing Embedded Software. Addison-Wesley, London (GB), 2002
[5] M. Conrad: Beschreibung von Testszenarien für Steuergerätesoftware – Vergleichskriterien und deren Anwendung. In: [8], S. 381-398
[5] M. Conrad, E. Sax: Mixed Signals. In: [4], S. 229-249
[6] Web pages of the dSPACE company; (October 2005) http://www.dspace.de/
[7] Web pages of the ETAS company; (October 2005) http://de.etasgroup.com/
[8] 10. Internationaler Kongress „Elektronik im Kraftfahrzeug“ (Tagungsband), VDI-Berichte, Band 1646, VDI Verlag, Düsseldorf (D), 2001
[9] Eckard Lehmann; Time Partition Testing, Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen; Dissertation, Berlin 2004
[10] Web page of MTest: Tool for model based tests of embedded systems (dSPACE [6]); (2005) http://www.dspace.de/ww/de/pub/products/sw/expsoft/mtest.htm
[11] Web pages of the The MathWorks Inc.; (October 2005) http://www.mathworks.com/
[12] TestML.xsd; XML schema of the TestML language. (2005) Available at http://www.immos-project.de/TestML/TestML.xsd