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

Juergen Grossmann[1], Ines Fey[1], Alexander Krupp[3],
Mirko Conrad[4], Christian Wewetzer[2], and Wolfgang Mueller[3]

[1] DaimlerChrysler AG, Alt Moabit 96a, D-10559 Berlin
juergen.grossmann, ines.fey@daimlerchrysler.com
[2] dSPACE GmbH, Technologiepark 25, D-33100 Paderborn
christian.wewetzer@dspace.de
[3] Paderborn University/C-LAB, Fuerstenallee 11, D-33102 Paderborn
alexander.krupp, wolfgang.mueller@c-lab.de
[4] Member of the ACM
mirko.conrad@acm.org

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 tests at several levels of abstraction such as Model-in-the-Loop (MIL), Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL) tests. Different test systems are applied for this purpose (e.g. dSPACE MTest, dSPACE Automation Desk, National Instruments Teststand) and almost each test system requests its own proprietary test description language. The exchange of tests between different test systems and the reuse of tests between different test levels is normally not possible. Efforts to integrate these heterogeneous test environments, 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 we present in this article – is defined to overcome the technological obstacles (different test language syntax and semantics, different data formats and interface descriptions) that almost automatically accompany the application of heterogeneous test tools and test infrastructures. TestML supports the exchange of tests between different test notations in a heterogeneous tool environment. 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. Tool support is illustrated by an application with Simulink/Stateflow.

1 Introduction

Development processes in the automotive industry are highly distributed and fragmented. The Original Equipment Manufacturer (OEM) acts as the system

M. Broy, I.H. Krüger, and M. Meisinger (Eds.): ASWSD 2006, LNCS 4922, pp. 98–117, 2008.

integrator and solution provider. He is responsible for the development of high level specifications and the integration and quality assurance at system level. The software and hardware of the individual electronic control units (ECUs) are normally provided by different suppliers. Tools, methods and data formats used in the development processes of the suppliers and the OEMs are normally different.

Moreover, new development paradigms, such as model based development, have to be integrated into existing development processes and tool chains. Model-based specifications in development and the establishment of powerful code generators have led the development process to be noticeably more effective, and automated at a higher level of abstraction. Due to the availability of executable models, tests and analytical methods can be applied early and integrated into subsequent development steps. The positive effects — early error detection and early bug fixing — are obvious.

To keep the whole development and test process efficient and manageable, the definition of an integrated and seamless approach is required. Such an approach especially would address the subjects of test exchange, autonomy of infrastructure, methods, and platforms and the reuse of tests. The respective technological basis will be constituted by a domain specific test language, that is executable will unify the test infrastructure as well as the definition and documentation of tests. For this purpose, the BMBF project IMMOS (Integrated Methodology for Model-based ECU Development) was carried out by DaimlerChrysler AG, IT Power Consultants, dSPACE GmbH, Fraunhofer FIRST, FZI Karlsruhe and Paderborn University. We present the test exchange language TestML as a substantial project result.

In section 2 we give a short overview of the test processes in the automotive domain and address related work. Section 3 describes the overall purpose and ideas behind TestML whereas section 4 depicts the set-up and structure of the language itself. To illustrate the behavioral semantics of TestML we provide a mapping between TestML constructs and Matlab/Simulink constructs in Section 5. Section 6 provides a number of short test cases that exemplify the expressiveness of TestML. Section 7 summarizes the paper.

2 Related Work

In the industrial practice an automotive control system has to pass several kind of tests on different levels. Tests that go along with the integration of the complete vehicle system are mainly the responsibility of the OEM. These tests address the interaction between control units, the vehicle communication infrastructure and last but not least tests of the complete vehicle system. Tests on ECU level are mainly in the responsibility of the respective suppliers. They encompass the verification of the of the software driven functionality and the electronic characteristics of the ECU.

Actually a wide range of different test and simulation environments are used in the automotive domain. For tests on model level, so-called Model-in-the-Loop

(MIL) environments are used. To test the software itself, so-called Software-in-the-Loop (SIL) and Processor-in-the-Loop Environments are introduced[1]. In the end the integration between soft- and hardware (i.e. the complete ECU) is tested using Hardware-in-the-Loop- (HIL) Environments [1]. Besides software related tests (functionality, software integrity, software robustness), HIL environments allow to test the electronic characteristics and may simulate a complete network of interacting ECUs. Finally the OEM uses HIL Environments to test and simulate the complete electronic infrastructure of a vehicle.

Normally, different test systems are applied for the different simulation environments and almost each test system has individual requirements for methods, languages and concepts. The established test tools from National Instruments [2], dSPACE [3], Etas [4], Vector [5], MBtech [6] for example are highly specialized ones. They rely on proprietary languages and technologies and are mostly closed in respect to portability, extension and integration. In comparison, in the domain of Electronic Design standardization efforts have culminated in the definition of a set of verification and test languages: SystemVerilog [7], PSL [8], and e [9]. The languages and tool support facilitate the efficient creation of highly automated model-based testbenches [10]. This is achieved by a concise representation of interfaces including timing information, and by support for constraint based stimulus generation, and advanced (temporal) coverage and assertion facilities for evaluation. However, the current concepts focus on digital design, and they do not focus on testing of continuous domain models which are commonly applied in the automotive industry.

Efforts to address test exchange and test reuse in such heterogeneous environments are still in the beginning. The emerging IEEE standard ATML [11] is not finalized and not supported by the automotive tool chains. A new promising approach, that is based on TTCN-3 [12] and — among others — integrates the TestML concepts described in this paper, is already under definition but not yet available for industrial practice [13,14].

3 The TestML Principles

TestML is a tool-independent XML-denoted language, which was developed for the interchange of test descriptions. The language elements are represented by individual XML elements that are defined by an XML schema. The complete XML schema for TestML can be accessed in [15].

TestML 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

[1] A SIL environment allows to test the compiled target software using environment models both running on standardized personal computers. A PIL platform additionally simulates the targets processor environment to allow tests that address special target platform issues.

are supported. Our aim is to realize an interchange format for the large spectrum of test description languages established in the automotive industry. This section describes the purpose of TestML and expands on the background and the demands of the language design.

3.1 A Unified Format for Test Exchange

Our basic assumption is that semantic similarities as well as overlaps exist between the different test description languages (see [16] 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. This includes software from different sub-domains like telematics, body control, power train and driving assistance. Whereas communication related issues are crucial for telematics, the majority of body control and driving assistance software belongs to feedback control systems that rely on sensors and actuators and have to deal with a large amount of continuous real world data. The general conditions of the different automotive sub-domains 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:

- specification of discrete and continuous (analogue) stimuli
- a concept of time to describe time-dependent events
- specification of reactive test cases to test feedback communication and control systems
- 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 cover test descriptions at different levels of abstraction (such as test scenarios and test data) independent from the respective tool environment.

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 (see Fig. 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 quadratically.

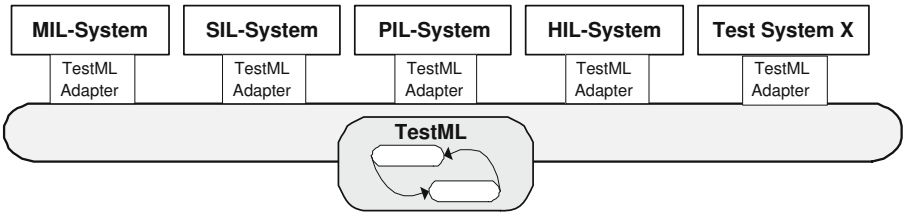

Fig. 1. Integration of different test descriptions using an intermediate language

3.2 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. Setup 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 [16].

In its function as an exchange language TestML enables operation of several 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 define one such abstraction termed TestML test system. A TestML test system consists of a combination of test components that we consider minimally necessary 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 setup 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.

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 is hidden 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.

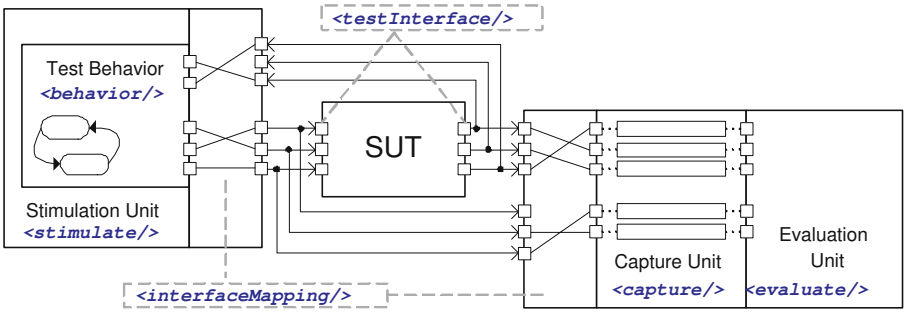

Fig. 2. Abstract test system for TestML

3.3 Test Behavior

The term test behavior subsumes processes that describe the stimulation of the test object at the moment of test execution. To serve its function as an exchange language, TestML has to cover and integrate the widest possible spectrum of different behavioral descriptions. 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 ([17], [16]). 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.

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 this purpose. For a good synopsis for the automotive sector see [16]. For TestML we decided to use hybrid timed automata as a basic concept to describe test behavior. Hybrid automata 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. [18]). The use of hybrid automata to describe continuous test behavior in the context of embedded systems could be successfully shown in [19]. To support the definition of exact and well-defined mappings between existing test languages and TestML we provided a rigorous formal behavioral semantics for TestML by means of Abstract State Machines (ASMs) [20].

4 Structure and Elements of TestML Test Descriptions

This section describes the elements and the setup 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 describe an operating scenario to be tested by means of test inputs.
- 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. Figure 3 shows the section of the TestML schema in which the basic elements are defined.

4.1 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 type safe 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.

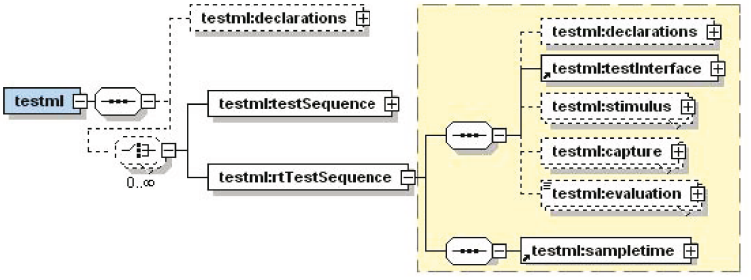

Fig. 3. The basic elements of a test description with TestML

Specification of a test interface

```
<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>
```

The listing above 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*.

4.2 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 section 3.3.

4.3 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 section 4.3. 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 section 3.3 for details). Moreover, the elements *stimulate* and *capture* contain mappings that map the input and output ports of a test interface (element *port*, see section 4.1) on the elements of a behavior signature (element *signature*, see section 3.3) 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*.

Mapping of channels to capture variables

```
<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>
```

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. These operators and command allow the comparison of signal values and complete signal shapes and are explained in the following section.

4.4 Data Types, Operators and Expressions

In most programming languages the basic data types are bool, integer, double, and 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 are referenced within TestML by their ID.

Instantiation of data types

```
<double ID="var1" name="Examples variable"/>
<write>
  <doubleRef IDREF="var1"/>
  <value><double>
    <value>100</value>
  </double></value>
</write>
```

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 waveform is represented by means of simple signal expressions or TestML automata, which are introduced in section 3.3. The next listing shows a simple signal expression specifying a ramp which rises from 0 to 100 within 10 seconds.

Instantiation of a signal

```
<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>
```

For expression evaluation a set of simple operators is provided. Table 1 shows a selection of operators. 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. They may be applied in an extended form to signals, which represent a time series of values.

Table 1. List of Operators

Operator	Meaning	Operator	Meaning
$< add/ >$	addition	$< and/ >$	logical and
$< sub/ >$	subtraction	$< or/ >$	logical or
$< mult/ >$	multiplication	$< not/ >$	logical not
$< div/ >$	division	$< xor/ >$	logical xor
$< abs/ >$	absolute value	$< equ/ >$	equality
$< max/ >$	maximum value	$< grt/ >$	greater
$< min/ >$	minimum value	$< geq/ >$	greater or equal

An example of a comparison expression is shown in the next listing, which compares two signals with the identifiers $Data1\_v\_act$ and $Data2\_v\_act$.

comparison expression

```
<cond>
  <grt>
    <signalRef IDREF="Data1_v_act"/>
    <signalRef IDREF="Data2_c_act"/>
  </grt>
</cond>
```

4.5 Behavioral Constructs

The structure and elements of the element *behavior* are depicted in Fig. 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*. 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.

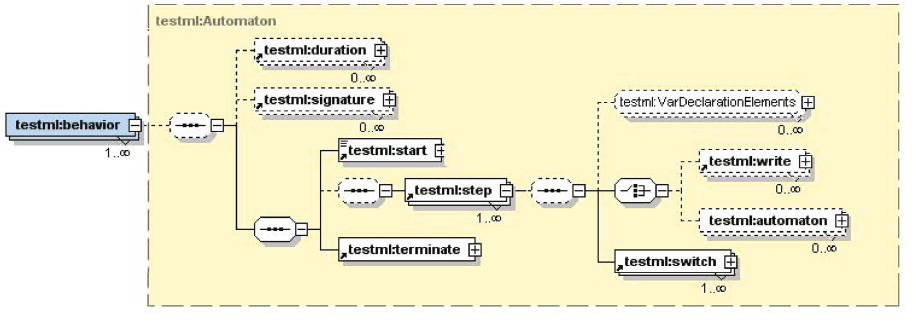

Fig. 4. The element behavior

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 by conditions for a time- or value-dependent control of automata. 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.

5 Mapping TestML Automata to Matlab Simulink

This section provides an executable TestML implementation using Simulink/Stateflow [21]. We mainly emphasize on behavioral aspects and introduce a detailed mapping that maps the TestML constructs defined in section 4.5 to the well-defined behavioral constructs of Simulink/Stateflow. For a concise TestML semantics refer to [20].

In TestML we generally distinguish between timed automata and non-timed automata. Timed automata reside inside TestML real-time test cases and non-timed automata reside inside non real-time test cases. Each top level TestML automaton, independent of its type, can be mapped to a Stateflow chart, that resides inside a Simulink Stateflow block. The signature of a TestML automaton is represented by the input and output ports of the Stateflow block. Hence the mapping between the automaton signature and an arbitrary test interface can be simply realized by drawing lines between Simulink ports. Figure 5 shows a Stateflow block called "TestML_Automaton", which provides four data output ports *phi_Brake*, *phi_Gas*, *v_des*, *leverPos* and one data input port *v_act*.

Each Stateflow chart that represents a TestML Automaton consists of a top level state with the same name as the TestML automaton. This top level state contains further states and transitions, which each represent either a TestML *step* or a TestML *switch*. For time control the top level state provides a local time property called *time*. We define local time with

$$time = t - startTime$$

in which *time* represents the local time, t holds the global time, and $startTime$ represents the point in time that the automaton has started. In general global time progress is realized by Simulink simulation time using a discrete simulation solver. Moreover, each TestML step — except a final or start step — is realized by a Stateflow state, which provides a local time property called $stepTime$. Local step time is defined in the same way as local automaton time. The property $startTime$ here represents the point in time when the step has started. As we will see later on, TestML steps have to control the running status of all embedded entities (write statements and embedded automata). Hence we

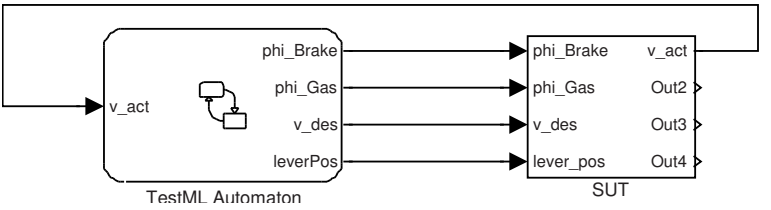

Fig. 5. A TestML automaton represented by Stateflow

introduce a step property called *runningEntities*. This property is initialized when a step is entered and holds the number of all embedded entities (e.g. $en: \ runningEntities = 2$). It is decremented whenever an embedded entity stops and sends the *EntityStopped* event.

$$on \ EntityStopped : runningEntities - -$$

TestML *start* steps are simply realized as Stateflow default transitions. TestML *final* steps are denoted as empty states with the name "final".

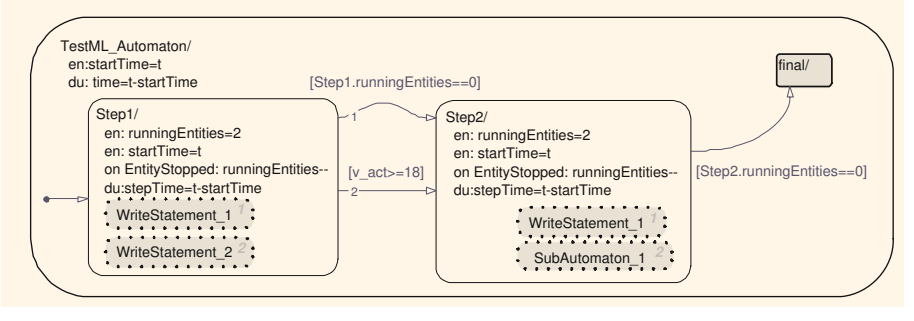

Fig. 6. Base structure of a TestML automaton in Stateflow

We have to provide an equivalent Stateflow transition for each TestML *switch*. A switch is composed of the elements *cond* and *next*. The element *next* can simply be interpreted as transition end, which refers to the next Stateflow state to be executed when the respective transition condition evaluates to *true*. The element *cond* expresses a switch condition that can in most cases be mapped directly to a Stateflow transition condition. The sole exception is the absence of a switch condition. In TestML the absence of a switch condition defines a so-called "default switch" which fires when all embedded entities of the current step have been finished. In contrast a Stateflow transition without any annotations fires immediately after its source state is activated.

We can not use the empty Stateflow transition to implement the TestML default switch. Instead we have to check the status of all embedded entities.

When all entities are finished the property *runningEntities* of a TestML step is zero. We can use this as a switch condition of a Stateflow transition.

$$[< Stepname > .runningEntities == 0]$$

Figure 6 shows the base structure of a TestML automaton implemented with Stateflow. The Automaton consists of two steps with two switches in between: a TestML default switch and a switch listening on the input port *v_act*. The main functional behavior of a TestML step is defined by its embedded entities. These are either embedded TestML automata or write statements which assign signals or simple values to ports. Embedded TestML automata are implemented almost in the same way as top-level automata. Sole difference: Embedded automata have to provide the *EntityStopped* event that will be fired when the execution of the automaton has finished. The event is triggered by the final state and can be realized by *en* : *EntityStopped* using the Stateflow Action Language. The concrete implementation of TestML write statements depends on the type of the enclosing automaton. Inside timed automata we assign timed signals to ports. Timed signals have a distinct duration. When executing write operations, one has to consider and control the duration of the signal that is written. In contrast, non-timed automata only provide simple value assignments that have no duration. Temporal control is not necessary here.

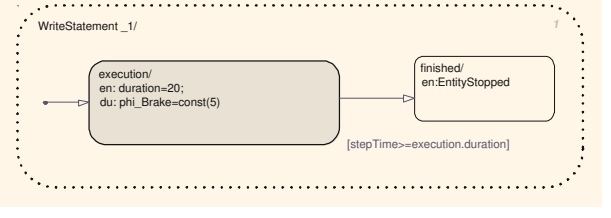

Fig. 7. Timed write statement which applies a constant signal to *phi_Brake*

We start with the implementation of write statements that reflect duration. For each write statement we provide a top level state that contains a controller structure that manages temporal and functional behavior. Multiple write statements that belong to the same TestML step are realized as multiple parallel executed top level states (see section 6). The enclosed controller structure consists of two states. The "execution state" is responsible for signal execution and its application to a port. When activated, an execution state calculates signal values as part of its during action and applies the calculated value to a specified port (e.g. $du : phi\_Brake := const(4)$). We may use graphical functions for value calculation that each represent a signal of a certain kind which is parameterized by a set of signal-specific parameters (e.g. $ramp(offset, slope, limit)$, $const(constval)$), $sinewave(offset, frequency, amplitude)$. The "finished" step

is triggered when the actual step time exceeds the signals duration. We implement this using the following transition condition:

$$[executer.duration > stepTime]$$

During its entry a finished step fires an *EntityStopped* event *en* : *EntityStopped* so that the enclosing step is informed about the signals end. Figure 7 shows the implementation of a write statement that applies a constant signal with the length of 20 seconds to a port called *phi_Brake*.

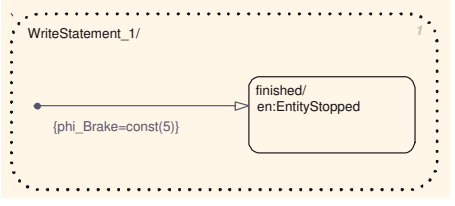

Fig. 8. Non-timed write statement which applies a constant signal to *phi_Brake*

Non-timed automata contain write statements that do not reflect duration. For this kind of write statements we adopt the structure of timed write statements and simply omit the execution state. The value calculation and its application to a port are realized as an action defined as part of the Stateflow default transition. Figure 8 shows a non-timed write statement that applies a constant value to a port called *phi_brake*.

6 Exemplary Use of TestML Automata

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 section 3.3. In the following examples, the TestML automata are not depicted in their XML representation but as annotated states in an UML alike notation[2].

6.1 Specification of Timed, Deterministic Test Stimuli

We 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 brake and gas pedal. For this

[2] We deliberately avoid to depict the XML representation here, since this quickly becomes too large even for short examples. The use of graphical representations makes a more compact visualization possible. In the following the TestML element *step* is depicted as a state and the element *switch* is presented as a transition. Statements which are used within the element *step*, either for the definition of signals or simple values or to assign these definitions to a port, are annotated in the form of pseudo code inside the states. For further information on XML representations, refer to the enclosed schema.

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 percent and *phi_Brake* represents brake pedal travel and also given in percent. 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%.

Figure 9 shows both the individual signal forms and the description used for the generation of signal forms with TestML. Every state of the TestML

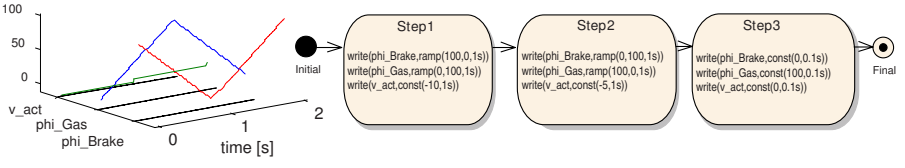

Fig. 9. Specification of synthetic stimulation sequences with TestML automata

automaton depicted above defines a time interval 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 one second on the channel called *phi_Brake* $[write(phi\_Brake, ramp(100, 0, 1s))]$.
- Write a ramp signal with the value course from 0 to 100 and the length of one second on the channel called $[phi\_Gas\ (write(phi\_Gas, ramp(0, 100, 1s))]$.
- Write a constant signal with the value -10 and the length of one second on the channel called *v_act* $[write(v\_act, const(-10, 1s))]$.

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. The complete execution stops as soon as the final state is reached.

- Write a ramp signal with the value course from 100 to 0 and the length of one second on the channel called *phi_Brake* $[write(phi\_Brake, ramp(0, 100, 1s))]$.
- Write a ramp signal with the value course from 0 to 100 and the length of one second on the channel called *phi_Gas* $[write(phi\_Gas, ramp(100, 0, 1s))]$.
- Write a constant signal with the value -10 and the length of one second on the channel called *v_act* $[write(v\_act, const(-5, 1s))]$.

6.2 Specification of Timed, Reactive Test Stimuli

In order to be able to show the use of TestML automata for the specification of reactive test behavior, the example mentioned above needs to be modified. Here, it is not the pedal interpretation which is tested rather than the cruise control. The test interface is expanded by an input and an output channel. The port *v_target* is an input port and describes the desired vehicle speed; *v_act* is an output port and represents the current vehicle speed. The reactive test behavior may be described as follows:

1. Set target speed *v_target* at 18 m/s
2. Use gas pedal until vehicle speed is greater than or equals 18 m/s.
3. Switch on cruise control.
4. Use brake pedal until vehicle speed equals 0 m/s.

Figure 10 shows the individual signal forms (left-hand side) as well as the description for the generation of the signal forms with (right-hand side).

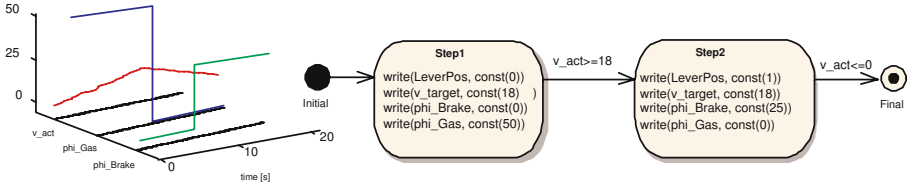

Fig. 10. Specification of reactive stimulation sequences with TestML automata

In contrast to the example from the previous section, 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 "Step1" is fulfilled. Then the instructions from "Step2" are executed.

- Write a constant signal of 50 on *phi_Gas* $[write(phi\_Gas, const(50))]$ and a constant signal of 0 on *phi_Brake* $[write(phi\_Brake, const(0)]$. The cruise control is switched off $[write(LeverPos, const(0))]$ and the velocity of the target vehicle is set to $18m/s$ $[write(v\_target, const(18))]$.
- If the channel *v_act* has taken on a value greater or equal 18 m/s, write a constant signal of 0 on the channel *phi_Gas* $[write(phi\_Gas, const(0))]$, a constant signal of 70 on the *phi_Brake* channel $[write(phi\_Brake, const(70))]$ and switch on the cruise control $[write(LeverPos, const(1))]$. The velocity of the target vehicle remains at $18m/s$ $[write(v\_target, const(18))]$.

With a basic set of signal primitives and their suitable combinations, the use of TestML automata supports the definition of complex signal forms.

6.3 TestML Automata for 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 dSPACE [3]. Again, we define the test of a cruise control function. Besides the already known phi_Brake input the brake pedal flag BrakePedal is also read.

A possible hysteresis 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.

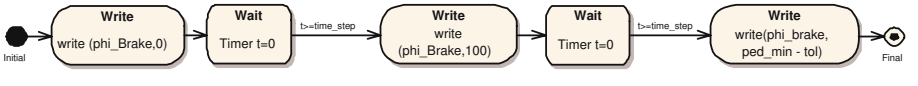

Fig. 11. Specification of message based test sequences with TestML automata

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

7 Summary and Outlook

TestML is a XML-noted test exchange language that is tailored towards 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 for test definitions, which can 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. 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.

Test scenarios capable of real-time use are an important functionality of HIL test beds. As a major extension over most existing means for test description, TestML provides language constructs for tests under real-time conditions. This enables support of the entire development process by the test exchange language.

Special care was taken to provide a flexible test behavior description language covering different levels of abstraction. The concept of hybrid automata used to capture test behavior permits the mapping of common classes of automotive test descriptions, including 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 automata made it possible to avoid an overloading of the exchange language with manifold constructs, which ultimately would have led to semantically redundant definitions. We would like to thank our participants in the IMMOS project for their contribution to TestML, especially S. Sadeghipour and H.-W. Wiesbrock from ITPower Consultants, Prof. H. Schlingloff and M. Friske from Fraunhofer FIRST.

References

1. Schäuffele, Zurawka (ed.): Automotive Software Engineering. Vieweg & Sohn Verlag, Wiesbaden (2006)
2. National Instruments: Web pages of the National Instruments corporation (2007)
3. dSpace AG: Web pages of the dSpace corporation (2005)
4. Etas Group: Web pages of the Etas Group (2007)
5. Vector Informatik GmbH: Web pages of the Vector Informatik GmbH (2007)
6. MBtech Group: Web pages of the MBtech Group (2007)
7. IEEE: IEEE Std.1800-2005 - Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language (2005)
8. IEEE: IEEE Std.1850-2005 - IEEE Standard for Property Specification Language (PSL) (2005)
9. IEEE: IEEE Std.1647-2006 - Standard for the Functional Verification Language 'e' (2006)
10. Bergeron, J., Cerny, E., Nightingale, A., Hunter, A.: Verification methodology manual for SystemVerilog. Springer, Heidelberg (2006)
11. SCC20 ATML Group: IEEE ATML specification drafts and IEEE ATML status reports (2006)
12. ETSI: ES 201 873-1 V3.2.1: Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 1: TTCN-3 Core Language (2007)

13. Schieferdecker, I., Großmann, J.: Testing of Embedded Control Systems with Continous Signals. In: Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme II, TU Braunschweig, pp. 113–122 (2006)
14. Schieferdecker, I., Bringmann, E., Grossmann, J.: Continuous TTCN-3: testing of embedded control systems. In: SEAS 2006: Proceedings of the 2006 international workshop on Software engineering for automotive systems, pp. 29–36. ACM Press, New York (2006)
15. IMMOS Project: TestML schema definition version 1.0.3 (2006)
16. Conrad, M.: Modell-basierter Test eingebetteter Software im Automobil. PhD thesis, TU-Berlin (2004)
17. Conrad, M., Sax, E.: Mixed signals. In: Testing Embedded Software, pp. 229–249 (2003)
18. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: hybrid systems, pp. 209–229 (1992)
19. Lehmann, E.: Time Partition Testing Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen. PhD thesis, TU-Berlin, Berlin (2004)
20. Grossmann, J., Mueller, W.: A Formal Behavioral Semantics for TestML. In: Proc. of IEEE ISoLA 2006, Paphos Cyprus, pp. 453–460 (2006)
21. The MathWorks: Web pages of Simulink - Simulation and Model-Based Design (2006)