A Formal Behavioral Semantics for TestML

Jürgen Grossmann
DaimlerChrysler AG
Forschung und Technologie REI/SM
Alt Moabit 96a
D-10559 Berlin
Email: juergen.grossmann@daimlerchrysler.com

Wolfgang Müller
Universität Paderborn
C-LAB
Fürstenallee 11
D-33102 Paderborn
Email: wolfgang@acm.org

***Abstract*— TestML is an XML-based language for the exchange of test descriptions in automotive systems design and mainly introduced through the structural definition of an XML schema as an independent exchange format for existing tools and methods covering a wide range of different test technologies. In this paper, we present a rigorous formal behavioral semantics for TestML by means of Abstract State Machines (ASMs). Our semantics is a concise, unambiguous, high-level specification for TestML-based implementations and serves as a basis to define exact and well-defined mappings between existing test languages and TestML.**

I. Introduction

Test processes in automotive industry are tool-intensive and affected by technologically heterogeneous test infrastructures. In industrial designs, 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 imposes 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.

TestML was developed as a language for exchanging test descriptions between different test platforms and test systems in the context of model-based testing of embedded automotive software. TestML is based on a self-contained language definition that enables test descriptions of different levels of abstraction (such as test scenarios and test data), independent from the respective tool environment. Language requirements thus were derived from domain-specific demands on one hand and the general needs of an interchange language on the other hand. 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:

1) specification of discrete and continuous (analogue) stimuli,
2) a concept of time to describe time-dependent events,
3) specification of reactive (closed-loop) test cases,
4) description and management of measuring data as inputs as well as reference data for comparative tests,
5) expressions for test evaluations regarding the analysis of discrete and continuous signals.

Since real-time constraints exist for many applications in automotive industry, test evaluation is defined in TestML as an off-line process. Therefore, stimulation (test execution) and test evaluation are specified as separate processes.

This article mainly focus on TestML stimulation and capturing processes not covering test evaluation. For a concise, unambiguous, and high-level behavioral semantics to define the execution of TestML stimulation and capturing constructs, we use Abstract State Machines (ASMs), which have been successfully applied to comparable languages in other projects.

The article starts with a short overview and related work (section II), followed by a brief overview of Abstract State Machines (section III)and an informal introduction to TestML and its main elements (section IV). Section V provides a formal and precise semantics of TestML constructs by ASMs. Section VI closes the paper with our conclusions.

II. Related Work

Whereas Gurevich has originally defined Evolving Algebras (EAs) for considerations in complexity theory, multiple publications have demonstrated the applicability of EAs for formal specification in various fields [1]. Examples are *hardware architectures* (e.g., APE100 [2], DLX [3]), *software architectures* (e.g., Warren Abstract Machine [4], Constraint Logical Arithmetical Machine [5], Parallel Virtual Machine [6]), and *protocols* (e.g., bakery algorithm [7], group membership protocol [8], kermit [9]).

In addition, ASMs have been successfully applied to the definition of semantics of various programming, system, and hardware description languages. Examples for programming languages are Prolog, , and C++ [10], and Java [11]. The ITU standard SDL 2000 was the first standard where the complete dynamic semantics is described by means of ASMs [12]. For hardware description languages, ASM was applied to define the different behavioral semantics of their respective simulators. In 1995 the simulation semantics of VHDL'93 was also introduced by Abstract State Machines in [13] as a complementary precise, formal, but yet readable, documentation to capture the relevant principles of non-trivial interaction of the user defined VHDL processes and the VHDL simulation kernel. Later, the ASM approach was also applied to define the complete execution semantics of SystemC in [14], SpecC in [15] and SystemVerilog in [16].

Test languages are a special kind of programming languages. Their main focus is the definition of distinct application scenarios to test a system. They typically provide a set of expressions and statements to support the definition of messages or signals applied to a system. Additionally they provide a set of evaluation statements to evaluate the system's reaction. The levels of formalism used to define the expressions and statements of a certain test language are different [17] and strongly dependent on their application domain. Executable test languages in the field of real-time testing are particularly defined using formalisms with precise mathematical background. The Classification Tree Method for Embedded Systems (CTM/ES) is defined by use of set theory and the notion of mathematical functions (cf. [17]). The latter are used to define interpolations between the distinct supporting points acquired by the Classification Tree Method. In Time Partition Testing [18], a language to define reactive tests in the field of the automotive domain, mathematical functions are used to define signals. The notions of streams [19] and hybrid automata [20] were used to describe the overall control flow and the concatenation of simple signals to more complex ones.

In this paper, we have chosen the notion of simple value expressions to describe our most simple values, the notion of mathematical functions to describe signal primitives, and the notion of ASMs to provide a precise and unambiguous semantics of TestML control flow constructs. As TestML particularly serves as an exchange language, the language itself has to be completely free from ambiguities, easy to capture and to use. Using ASMs, we introduce a semantics definition, which provides the appropriate level of abstraction to be easily understandable, especially for programmers and software engineers who are used to think in terms of algorithms and state transitions. Further on, ASMs provide the necessary precision for unambiguous test exchange and reproducible test execution on different platforms.

III. Abstract State Machines

Gurevich initially introduced basic Abstract State Machines (ASMs) in 1991 [21]. A revised definition of ASMs with various extensions, commonly known as the Lipari Guide, was published in [22]. Whereas Gurevich has originally defined ASMs for considerations in complexity theory, multiple publications have demonstrated the applicability of ASMs for formal specification for various purposes. Examples come from hardware architectures, software architectures, communication protocols, and programming languages [1].

An ASM specification is a program executed on an abstract machine. The program comes in the form of guarded function updates (rules). Rules are nested if–then–else clauses with a set of function updates in their body. Based on these rules, an abstract machine performs state transitions with algebras as states. Firing a set of rules in one step performs a state transition. Only those rules are fired whose guards (*Condition*) evaluate to true. Rules are of the form

if *Condition* **then** <*Updates*> **else** <*Updates*> **endif** .

At each step the guards evaluate to a set of function updates (block) each of the form

$$f(t_1, ..., t_r) := t_0$$

where t_i are terms (including functions). A block is a set of function updates separated by a comma [1]. The individual function updates of each block are collected in a so-called update set. The individual updates of the update set are simultaneously executed in one step. That means that all updates in a block are simultaneously executed. Each function update changes a value at a particular location, which is given by the left hand side of the assignment. Functions are considered as global. In the classical ASM definition, two or more simultaneous updates of the same location in one update set define inconsistency and no state transition and no update in the update set are executed. In this chapter, we take a slightly modified ASM definition. In the case of more than one update on the same location we non deterministically choose one and remove the others from the update set.[2]

The following example illustrates a guarded update of a block with two update instructions:

if *Condition* **then** $A := B, B := A$ **endif**

It defines the simultaneous update of the 0-ary functions A and B. Since both updates are simultaneously executed, A becomes the value of B and vice versa. The rule fires when *Condition* evaluates to *true*.

ASMs are multi-sorted and based on the notion of universes. We assume the standard mathematic universes of Boolean, Integer, List, etc., as well as the standard operations on them without further mentioning. A universe can be dynamically extended with individual objects by

extend *Universe* **with** v <*Rule*> **endextend** ,

where v is a variable which is bound by the **extend** constructor.

The **choose** constructor defines an arbitrary selection of one element in a universe

choose v **in** *Universe* <*Rule*> **endchoose** ,

where v is non-deterministically selected from the given universe. The **choose** constructor can be qualified by a condition (**satisfying**). v is *undef* when the condition evaluates to *false*.

The **var** rule constructor defines the simultaneous instantiation of a rule:

var v **ranges over** *Universe* <*Rule*>

Executing the constructor means to spawn and execute the rule for each element in *Universe* simultaneously, i.e., the constructor basically spawns n rules where n is the number

[1] Note here that Gurevich [22] does not introduce a special symbol for separating updates in a block. We use a comma as an explicit block separator in this article.

[2] This modification has no impact on the theory of ASMs rather than helps us to simplify our definitions.

of elements in *Universe*. The following example demonstrates the application of this constructor. It defines a rule, which specifies that each non-empty l from the universe $LIST$ is replaced by the list's tail, i.e., deleting the first list element. l refers to any valid instance of $LIST$.

$$\textbf{var } l \textbf{ ranges over } LIST$$
$$\textbf{if } l \neq \langle\rangle \quad \textbf{then } l := tail(l)$$

Our definition of our TestML semantics is defined by distributed ASMs. Distributed ASMs consist of a collection of autonomously operating agents interacting with each other by reading and writing shared locations of global system states. The underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of partially ordered runs. Distributed ASMs partition rules into modules where each module is given by its module name ν. A module is instantiated to execute by setting $Mod(a) := \nu$ for an agent a. The symbol *Self* refers to a after the instantiation.

IV. TestML

The structural semantics of TestML is defined by means of an XML Schema (cf. [23]). A detailed description is available in [24] and [25]. The TestML schema [26] consists of a number of element definitions providing the fundamental basis to describe complete application scenarios to a system test.

The system on which these scenarios are applied is denoted as a system under test (SUTs). In TestML, a SUT is represented in terms of its interface only, the so-called test interface. A test interface is given by a set of input channels and output channels, which are represented by TestML elements of the type $port$. Each port can be characterized in more detail by defining its communication direction (input or output) and corresponding data type (e.g. boolean, integer, float).

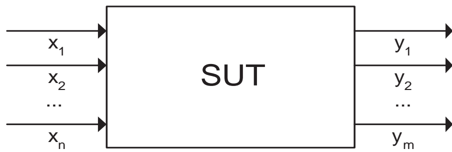

Fig. 1. The System Under Test is Represented by a Set of Channels (ports)

To describe concrete test scenarios, TestML provides the elements *testSequence* or *rtTestSequence*. While the element *rtTestSequence* can describe a real-time test scenario with an explicit sampling rate (element *sampletime*), the element *testSequence* represents a non real-time test scenario without an explicit sampling rate. In a real-time test scenario, each occurring event has a distinct relation to time. This means, that the timing behavior and duration of each statement is controlled by the execution environment during stimulation. Non real-time test scenarios, in contrast, do not provide timing control. They are for execution on non real-time execution environments where no timing constraints can be guaranteed.

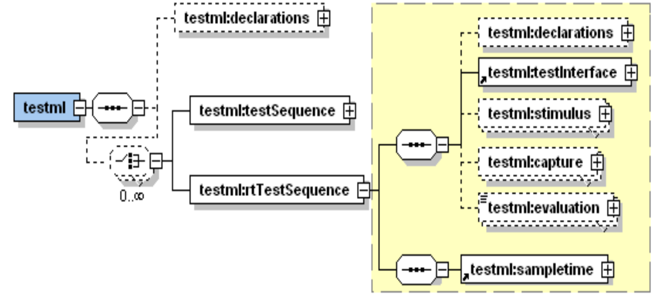

Fig. 2. Overview of TestML Basic Elements

The exact semantical difference between *testSequence* and *rtTestSequence* can be found later in section V.

Each TestML test sequence usually consists of *stimulate*, *capture* and *evaluate* as elements, which cover expressions and statements of a TestML test scenario. The *stimulate* element contains the complete behavior specification to generate the required stimulus signals. The behavior specification is modularly composed through *behaviour* elements. To capture the interaction between the SUT and our test sequences, we use capture variables. These variables serve as references to access recorded data. They are declared inside the *capture* element and are for test evaluation within the *evaluate* element later on.

A. Specifying Test Behavior with TestML

Specification of test behavior is one of the main aspects of describing tests and supporting the definition of test behavior is an essential part of a test description language. The focus of a such a behavioral description is how a test is carried out. According to this the term test behavior subsumes processes that define stimulation data and describe how these data are applied to the system under test (SUT). In practice, a variety of different methods, notations, and tools exist for the description of test behavior. For a comprehensive synopsis of the automotive sector see [17].

In order 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 have already defined some requirements in section IV. Below we take a closer look at different kinds of control flow complexity of different test languages.

Test languages, which support stimulation with discrete data only, often provide control flow statements (e.g., loops, if-statements etc.) to be completely Turing equivalent (e.g., [27]). Most of the languages that address continuous signal-based testing are less complex. With regard to these languages, we have to distinguish between different levels of behavior complexity. According to [17], we can state three levels here.

1) **Signal primitives:** The most simple form of test behavior complexity is realized by providing a set of primitive signal expressions to define basic wave forms (like sine waves, ramps etc.). Often parameterization is used to specify certain variants of a signal.

2) **Simple concatenation of signal primitives:** The next level of complexity is served when different signal primitives are combined through concatenation. This can be done by use of a sequential concatenation operator. These kinds of behavior complexity can often be found in simple signal generation frameworks (e.g., Simulink signal generation utility "signal builder" [28] , dSPACE Control Desk Stimulation Editor [29]).
3) **Programmed concatenation of signal primitives:** The most complex level is given when complex control flow structures are used, to define flexible concatenation during runtime. During specification time, we define multiple alternatives how signal primitives may be concatenated. Each alternative is annotated with predicates to be evaluated during runtime. The actual signal is processed as a result of this evaluation, respecting the ongoing interaction between the test scenario, the SUT, and the environment.

TestML supports all the levels of complexity mentioned above. The language provides a set of expressions to define simple values and a set of signal primitives. We take a closer look at this in the next section.

Furthermore, TestML provides an implicit sequential concatenation operator to compose more complex signals from signal primitives. To define the order and the timing of the concatenation, we use control flow structure defined through automata. This approach is well-known in hybrid automaton theory as well as from test methods in the automotive domain (cf. [18]). We will introduce our approach in section IV-C and section V.

B. Primitive Values and Signal Primitives

TestML provides a basic set of literals and operations to define simple values and signal primitives. A simple value is expressed through value expressions. In TestML we provide a set of numeric literals and operations (e.g., addition, multiplication, subtraction, division, sine) to define values for the most common numeric types (e.g., Integer and Double), as well as boolean literals and boolean operations (e.g., and, or, xor, not). For a detailed overview see [26], [25].

In addition to simple values, TestML supports the definition of signals, where a signal represents the change of a value over time. TestML provides statements to define deterministic signals with a well-defined and predictable value pattern and probabilistic signals with a certain amount of unpredictability (e.g., a noise wave with Gaussian probability).

To specify signals, we use simple mathematical functions as a basis and a set of predefined and function dependent parameters to define variants. The mathematical function states the type of the signal (e.g., constant signals, sine wave signals, ramps, exponential wave signals etc.), whereas the parameters are used to define certain signal characteristics (e.g., frequency, offset, amplitude). An additional and optional parameter *duration* exists for each signal to specify the exact length in time. If the parameter duration is left open the duration of a signal is infinite.

In the following, we provide formal definitions for signal statements used in TestML. As an example we provide the definitions of the sine wave signal statement, the ramp signal statement, and the exponential wave signal statement including their respective parameters. This only considers a small subset of signal primitives supported by TestML but should be sufficient to draw general conclusions:

1) **Sine Wave Signal:** A sine wave signal expression is defined by the sine wave function and the parameters *offset*, *amplitude*, *frequency*. The optional parameter *duration* specifies the length in time of the signal.

$$f(t) = amplitude * \sin(t * frequency) + offset$$

with

$$amplitude, frequency, offset \in \mathbb{R}$$

$$duration \in \mathbb{R}^+$$

and

$$t \in [0, duration[\ \subset \mathbb{R}^+$$

2) **Ramp:** A ramp signal is defined by a strict monotonic function with the parameters *slope*, *offset* and *limit*.

$$f(t) = t * slope + offset$$

and

$$(slope > 0 \wedge f(t) > limit) \vee (slope < 0 \wedge f(t) < limit)$$

$$\rightarrow f(t) = limit$$

with

$$slope, limit, offset \in \mathbb{R}$$

$$duration \in \mathbb{R}^+$$

and

$$t \in [0, duration[\ \subset \mathbb{R}^+$$

3) **Exponential Wave Signal:** An exponential wave signal is defined by an exponential function with the following parameters: *offset*, *limit*, and *tau*.

$$f(t) = offset + (limit - offset) * \left(1 - \exp^{(-t/tau)}\right)$$

with

$$tau, limit, offset \in \mathbb{R}$$

$$duration \in \mathbb{R}^+$$

and

$$t \in [0, duration[\ \subset \mathbb{R}^+$$

The following listing shows a TestML signal statement defining a sine wave signal with the duration of 10 seconds and the following characteristics: $f(t) = 100.0 * \sin(t * 1.0) + 5.0$

Specification of a Sine Wave Signal with TestML

```
<signal>
  <time>
    <unit>s</unit>
    <double>10.0</double>
  </time>
  <sine>
    <offset><double>5.0</double>
    </offset>
    <amplitude><double>100.0</double>
    </amplitude>
    <frequency><double>1.0</double>
    </frequency>
  </sine>
</signal>
```

In general signal primitives in TestML are enclosed by the element *signal*. Inside this element there are elements stating the duration of the signal (element *time*) and defining the kind of the signal (here element *sine*). Each signal kind provides its own parameters (e.g., offset, frequency, amplitude) to be able to define variants of a signal. The following graphic depicts the resulting signal plot defined in the listing above.

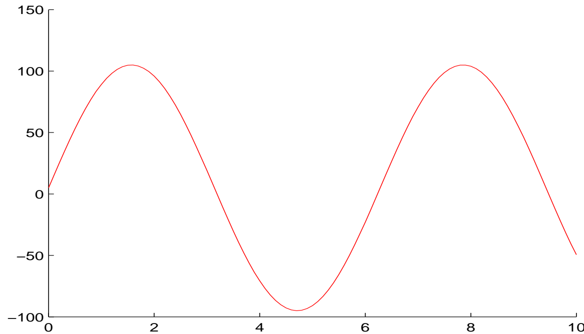

Fig. 3. Resulting Plot of the TestML Sine Wave Expression

C. Complex Stimulation Behavior

In TestML, complex stimulation behavior (cf. level 2 and 3 in section IV-A) is based on the primitive expressions we introduced in section IV-B, and a separate control flow structure, that we will introduce here. The control flow structure defines how and when partial defined signal primitives will be concatenated to obtain more complex stimulation signals.

This chapter gives an informal overview of the terms and concepts used in TestML to describe a control flow structure for signal concatenation. We informally describe how the so-called TestML automata are executed and we introduce all elements TestML automata are composed of. Based on the terms and concepts introduced here, we then define a precise and unambiguous behavioral semantics for TestML automata in section V.

To define our control flow structure, we use terms and concepts often used in the context of hybrid automata. Hybrid automata emanate from the theory of hybrid systems and are regarded as a mathematical model, which offers a sound basis for modeling timed applications and systems with discrete and analogue behavior (cf. [20]). The use of hybrid automata in the context of describing tests for embedded systems could be successfully shown in [18].

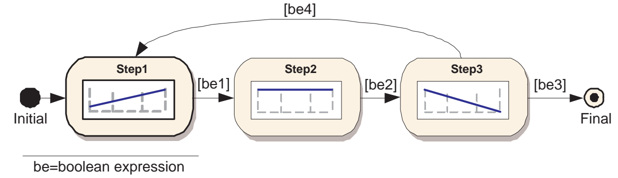

Fig. 4. Concatenating Signal Primitives Controlled through Automata

TestML automata consist of time consuming states – represented by elements called *step* – and time-less transitions – represented by elements called *switch*. A TestML automaton itself is represented through an element called *behavior*. Each step in turn is either defined by embedded TestML automata or contains one or more write statements (*write* element) to directly apply time-based signal primitives or simple values to the SUT. Each step contains at least one element called *switch*, that defines a transition that marks the passage between two steps. By use of the *cond* element the switching behavior of each switch can be constrained to certain time-dependent or value-dependent conditions. If a switch condition evaluates to true, the transition fires and the step referenced by the element *succ* (which is located inside the respective switch) will be processed next. For each step one transition without condition is allowed. A transition without condition fires after all time-dependent instructions defined within the respective step are terminated.

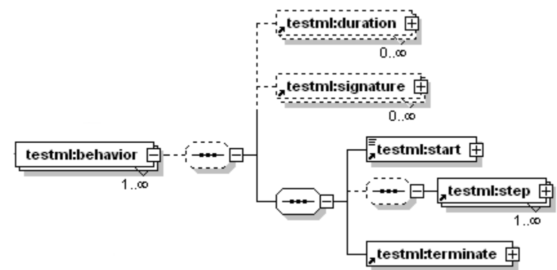

Fig. 5. The Behavior Element

In addition to steps and switches, each element *behavior* may include *duration* and *signature* elements. A duration constitutes the maximum duration of the behavior. A signature is an interface referring to internally specified signals. The element itself is composed of a number of declarations (much like the ports in the test interface) that can be individually accessed when reading or writing. The structure of the *behavior* element is depicted in Fig. 5.

V. FORMAL SEMANTICS

A TestML control flow structure is a set of parallel processes for real-time and non real-time stimulation and capturing. This directly applies a set of ASM agents with one agent for each test sequence. We further define the (i) domain of automata ATN. There we have one automaton for each real-time TestML behavior and each non real-time TestML behavior[3] and one additional for each capturing process.

We additionally have to introduce the concept of test sequence controller $\in TSC$, with exactly one test sequence controller for each test sequence agent $Self$. Its main task is to provide a consistent sampling rate and to observe the maximum duration defined for the stimulation and capturing processes. A test sequence controller thus determines the local time $t_{current}$ of the agent $Self$. In the case that $t_{current}(Self)$ is less or equal the individual $duration$ of the process $t_{current}(Self)$ is advanced by the individual $sampleTime(Self)$. Otherwise, the ASM agent is disabled by setting $Mod(Self)$ to $undef$. The execution of the controller can thus be simply defined by the following ASM rule.

if $t_{current}(Self) \leq duration(Self)$ **then**
  $t_{current}(Self) := t_{current}(Self) + sampleTime(Self)$
else $Mod(Self) := undef$
endif

Note here that a test sequence may have an explicit $duration$, which defines the maximum execution time of a test sequence. By default, $duration$ is set to the maximum testing time. For each test sequence there can be multiple automata with $type(a) \in \{realTime, nonRealTime, capture\}$ where $a \in ATN$. Each a can be in a $phase \in \{init, running, terminated\}$ with $init$ as a initial $phase$. In that phase a just initializes its local relative time t_{rel} to 0.0 and proceeds to $running$.

var a **ranges over** ATN
if $phase(a) = init$ **then**
  $t_{rel}(a) := 0.0$,
  $currentStep(a) := initialStep(a)$,
  $phase(a) := running$
endif

In $phase = running$, each a executes steps and step transitions by means of ASM macros $executeStep$ and $checkConditions$. Here, the different automata types perform a different execution. If $type(a) = realTime$ both are executed in parallel. Otherwise, a sequential execution is applied.

[3]TestML makes no explicit syntactical difference between real-time behavior and non real-time behavior. Here we use the term real-time behavior when the respective behavior definition is addressed from inside a real-time test sequence. We use the term non real-time behavior when it is addressed from inside a non-real-time test sequence.

var a **ranges over** ATN
if $phase(a) = running$ **then**
  if $type(a) = nonRealTime$ **then**
    $executeStep(currentStep(a))$;
    $checkConditions(currentStep(a))$
  elseif $type(a) = realTime$ **then**
    $executeStep(currentStep(a))$,
    $checkConditions(currentStep(a))$
  endif
endif

In the previous rule, $executeStep$ and $checkConditions$ refer to individual ASM macros, which are defined hereafter. The macro $executeStep(currentStep(a))$ defines the parallel execution of all write statements and all embedded $ATNs$ at $currentStep(a)$. Inside an ATN of type $realTime$ the maximum time needed to calculate all actual values expressed in statements, embedded automata and conditions is considered not to last longer than the specified sample time. Additionally, for type $nonRealTime$ $ATNs$ the maximum execution time is not specified and depends on the execution platform.

$executeStep(currentStep(a)) \equiv$
if $type(currentStep(a)) = statement$ **then**
  $executeStatement(currentStep(a))$
elseif $type(currentStep(a)) = realTimeATN$ **then**
  extend ATN **with** a
    $type(a) := realTime$,
    $phase(a) := running$
  endextend
elseif $type(currentStep(a)) = nonRealTimeATN$ **then**
  extend ATN **with** a
    $type(a) := nonRealTime$,
    $phase(a) := running$
  endextend
endif

The macro $checkConditions$ checks if the duration, i.e., maximal time, of an $a \in ATN$ is reached. If true, a terminates. Additionally, the duration of $currentStep$ and transition conditions of all $switches(currentStep(a))$ are checked. When t_{rel} reaches the duration of the individual step we check whether there is a default switch, i.e., *switch* without a condition. If there is one, we advance the current step to the next step. Otherwise, the execution hangs until the first valid condition hits and the execution continues with the next step.

$checkConditions(currentStep(a)) \equiv$
if $phase(currentStep(a)) = running$ **then**
  if $t_{current}(a) \geq duration(a)$ **then**
    $phase(a) := terminated$ **endif**
  if $t_{rel}(a) \geq duration(currentStep(a)$ **then**
   if $(exists(defaultSwitch(currentStep(a))))$ **then**
     $phase(currentStep(a)) = switching$ **endif**
   else $phase(a) := terminated$ **endif**
   endif
   if $evalSwitches(switches(currentStep(a))) = true$ **then**
     $phase(currentStep(a)) = switching$ **endif**
endif
if $phase(currentStep(a)) = switching$ **then**
  $currentStep(a) := nextStep(switches(currentStep(a)))$
endif

The definition of $nextStep$, selecting the next step from the first switch, which evaluates to true, should be intuitively clear. We thus just give the definition of the function $evalSwitches$ evaluating the different switches.

$$evalSwitches(switches) \equiv \\ \exists s \in switches : eval(condition(s)) = true$$

VI. Summary and Conclusions

In this paper, we have introduced a formal behavioral semantics for TestML by means of Abstract State Machines. Our semantics give a rigorous and unambiguous definition can be used for documentation and for the definition of mappings between existing test languages and TestML. We mainly focused on TestML stimulation and capturing processes not covering test evaluation since test evaluation is basically an off-line evaluation with little possibilities of control flow variations.

In [24] we have demonstrated that TestML is a suitable test language to express different kind of test behaviors covering the state-of-the-art test tools and languages. The used automaton notation 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 in test drives. The decision to map all test aspects on automata avoids an overloading of the exchange language with manifold constructs, which ultimately would have led to a superset of existing means of description.

By giving an unambiguous definition of the TestML means of description in this Paper, we advanced the TestML specification to be a suitable and reliable basis to exchange test definitions between different test platforms and languages used in the automotive industry (e.g., MTest, Time Partition Testing, Automation Desk). This can be used as a solid and trustworthy basis for test interchange between OEMs and suppliers and supports the increase of the level of test reuse during development.

At this moment, in the frame of the IMMOS project (www.immos-project.de), a TestML-based demonstrator is realized prototypically. The demonstrator shows the exchange of executable test descriptions between the MIL/SIL test environment MTest and the HIL test suite AutomationDesk, both from dSPACE.

Acknowledgements

The work described herein is funded by the BMBF through the IMMOS project. We would like to thank our partners in the IMMOS project for their contribution to TestML, especially S. Sadeghipour and H.-W. Wiesbrock (ITPower Consultants), Prof. Schlingloff and M. Friske (Fraunhofer/FIRST), Alexander Krupp (C-LAB), Klaus Lamberg and Christian Wewetzer (dSPACE), Ines Fey (DaimlerChrysler), and Mirko Conrad (The MathWorks). We also appreciate partial funds from SFB 614 (Selbstoptimierende Systeme des Machinenbaus).

References

[1] E. Börger, "Annotated Bibliography on Evolving Algebras," in *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[2] E. Börger, G. Del Castillo, P. Glavan, and D. Rosenzweig, "Towards a Mathematical Specification of the APE100 Architecture: the APESE Model." in *IFIP 13th World Computer Congress*, B. Pehrson and I. Simon, Eds., vol. I: Technology/Foundations, Elsevier, Amsterdam, the Netherlands, 1994, pp. 396–401.

[3] E. Börger, "Why use evolving algebras for hardware and software engineering?" in *SOFTSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, ser. Lecture Notes in Computer Science, M. Bartosek, J. Staudek, and J. Wiedermann, Eds. Springer-Verlag, 1996.

[4] E. Börger and D. Rosenzweig, "The WAM - Definition and Compiler Correctness," in *Logic Programming: Formal Methods and Practical Applications*, L. C. Beierle and L. Plümer, Eds. North-Holland, 1994.

[5] E. Börger and R. Salamone, "CLAM Specification for Provably Correct Compilation of CLP($\mathcal{R}$) Programs," in *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[6] E. Börger and U. Glässer, "A Formal Specification of the PVM Architecture," in *IFIP 13th World Computer Congress*, B. Pehrson and I. Simon, Eds., vol. I: Technology/Foundations, Elsevier, Amsterdam, the Netherlands, 1994, pp. 402–409.

[7] E. Börger, Y. Gurevich, and D. Rosenzweig, "The Bakery Algorithm: Yet Another Specification and Verification," in *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[8] Y. Gurevich and R. Mani, "Group Membership Protocol: Specification and Verification," in *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[9] J. Huggins, "Kermit: Specification and Verification," in *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[10] C. Wallace, "The Semantics of the C++ Programming Language," in *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.

[11] E. Börger and W. Schulte, "Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation," in *Mathematical Foundations of Computer Science, MFCS 98*, ser. Lecture Notes in Computer Science, L. Brim, J. Gruska, and J. Zlatuska, Eds. Springer Verlag, Berlin/Heidelberg/New York, 1998.

[12] U. Glässer, R. Gotzhein, and A. Prinz, "Towards a New Formal SDL Semantics Based on Abstract State Machines," *SDL '99 - The Next Millenium, 9th SDL Forum Proceedings*, 1999.

[13] E. Börger, U. Glässer, and W. Müller, "Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines," in *Formal Semantics for VHDL*, C. Delgado Kloos and P. T. Breuer, Eds. Kluwer Academic Publishers, 1995, pp. 107–139.

[14] W. Mueller, J.Ruf, and W.Rosenstiel, "An asm based systemc simulation semantics," in *SystemC - Methodologies and Applications*, W. Mueller, J.Ruf, and W.Rosenstiel, Eds. Kluwer, Dordrecht, 2003.

[15] W. Müller, R. Dömer, and A. Gerstlauer, "The Formal Execution Semantics of SpecC," in *International Symposium on System Synthesis, ISSS 02, Oct 2-4, Kyoto, Japan*, 2002.

[16] W. Mueller, M. Zambaldi, W. Ecker, and T. Kruse, "The Formal Simulation Semantics of SystemVerilog," in *Proc. of the Forum on Specification & Design Languages (FDL'04)*, Lille, France, 2004.

[17] M. Conrad, "Modell-basierter test eingebetteter software im automobil," Ph.D. dissertation, TU-Berlin, 2004.

[18] E. Lehmann, "Time partition testing systematischer test des kontinuierlichen verhaltens von eingebetteten systemen," Ph.D. dissertation, TU-Berlin, Berlin, 2004.

[19] M. Broy, "Refinement of Time," in *Transformation-Based Reactive System Development, ARTS'97*, M. Bertran and T. Rus, Eds., no. LNCS 1231. TCS, 1997, pp. 44 – 63. [Online]. Available: http://www4.informatik.tu-muenchen.de/publ/papers/arts97_boesswet_1997_Conference.pdf

[20] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in *Hybrid Systems*, 1992, pp. 209–229. [Online]. Available: citeseer.ist.psu.edu/alur92hybrid.html

[21] Y. Gurevich, "Evolving Algebras. A Tutorial Introduction," *Bulletin of EATCS*, vol. 43, pp. 264–284, 1991.
[22] ——, "Evolving Algebras 1993: Lipari Guide," in *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994.
[23] W3C, "Pages of the w3c dealing with xml schema and its specification," 2006. [Online]. Available: http://www.w3.org/XML/Schema#dev
[24] J. Grossmann, M. Conrad, I. Fey, C. Wewetzer, K. Lamberg, and A. Krupp, "Testml a language for exchange of tests," 2006. [Online]. Available: http://www.immos-project.de/site_immos/download/TestML_ASWSD.pdf
[25] J. Grossmann, C. Wewetzer, and A. Krupp, "Testml documentation," 2006.
[26] I. Project, "Testml schema defintion version 1.0.3," 2006. [Online]. Available: http://www.immos-projekt.de/TestML/TestML.zip
[27] ETSI, "Ttcn-3 on web pages of the european telecommunications standards institute (etsi)," 2005. [Online]. Available: http://www.etsi.org/ptcc/ptccttcn3.htm
[28] T. MathWorks, "The mathworks - simulink - simulation and model-based design." [Online]. Available: http://www.mathworks.com/products/simulink/
[29] D. AG, "Web pages of the dspace company," 2005. [Online]. Available: http://www.dspace.de/