# Introducing MDML - A Domain-specific Modelling Language for Automotive Measurement Devices

**Christian Burghard, Gerald Stieglbauer, Robert Korošec**[1]

**Abstract.** The method of model-based mutation testing has recently been introduced in the AVL List GmbH to automatically test the various measurement devices within the company's portfolio. However, the initial approach of using UML as a modelling language turned out to be non-satisfying. In this paper, we introduce a textual domain-specific language for the sole purpose of modelling the behaviour of measurement devices. Furthermore, we examine the technical differences to UML as well as the usability-related aspects, which determine the acceptance of the language by its end users.

## 1 Introduction

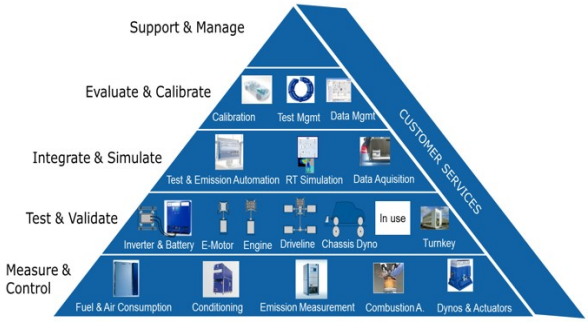


**Figure 1.** Portfolio AVL Instrumentation and Test Systems

AVL Instrumentation and Test Systems provides a product portfolio, which is applied in the automotive industry for integration and testing activities within the field of powertrain engineering (see Figure 1). Instruments (measurement devices, MDs) are an essential part of test systems, required for measuring specific quantities of the unit under test (air and fuel consumption, emissions, combustion, etc.). The measurement devices are integrated into a test automation system that orchestrates the different measurement devices participating in a test activity. An important part of the MD development process is testing of the control interfaces and of the devices' functional operation states. To make the development of testing procedures more effective and efficient, AVL initiated the introduction of model-based mutation testing into the process. The MoMuT test case generation tool [1, 2, 3] was used to generate test procedures, based on a model of the MD's functional behaviour. UML state diagrams were chosen as the modelling language for the device behaviour. However, this initial approach turned out to be non-satisfying in practice. An internal study revealed that UML was rejected for various technical and usability-related reasons. In this paper, we introduce a text-based domain-specific language (DSL) as an alternative to UML state diagrams. We examine technical aspects like semantics and expressiveness, as well as usability aspects like simplicity of model creation and intuitive understandability. We finally compare this DSL to UML in the context of our industrial case study and argue why the DSL seems to fit better and increases the acceptance rate by the end users.

The rest of this paper is structured as follows: Chapter 1.1 examines the MD testing process as it was before the introduction of model-based testing. Chapter 1.2 describes the former UML-based approach to model-based testing, as well as the problems of this initial approach. In Chapter 2 we describe the design approach and implementation of our DSL. Finally, the initial reactions of our test users, as well as a comparison to UML are summarized in Chapter 3.

### 1.1 Previous testing approach

AVL vehicle and powertrain testbeds are typically comprised of many different devices, such as measurement devices. The individual MDs are controlled by a test automation software called PUMA Open [4]. PUMA Open runs on a standard PC, which is connected to the devices. The MDs are tested by a group of test engineers. A server-sided abstraction layer allows them to handle each MD in the form of a software object which abstracts the device's functionality [5]. To test the integration of MDs into the automation system, simulation models virtualize the functionality of the MDs. This virtualization is required to avoid cost-intensive hardware and enable device variant testing. For this purpose the test engineers use a Testbed Simulator (TBSimu) which provides the same software interface as the actual devices. For the automated testing process, AVL uses an internal Test Automation Framework (TAF), which allows the execution of various NUnit Tests [6] on each MD's software interface. Traditionally, the NUnit tests are manually written by the test engineers. In most cases they use the MD's manual as a reference to define a set of test cases. Currently, there is no way to objectively measure the

[1] AVL List GmbH, Hans-List-Platz 1, 8020 Graz, email: {Christian.Burghard, Gerald.Stieglbauer, Robert.Korosec}@avl.com

test coverage, which has to be estimated empirically by the test engineer. In the event of a specification change, e.g. by creating a new MD variant, the test suite for a MD has to be manually revised. This process is error-prone and very time-consuming.

### 1.2 Introduction of a model-based testing approach

To make device testing more effective and efficient, a model-based testing approach was investigated in the TRUFAL project[2] [3]. The functional operation states of the MDs can generally be described in the form of state machines. As an initial trial, a UML State Diagram of the AVL489 particle emission measurement device was developed [5]. The definition of the necessary subset of UML model elements and their semantics [1] was reused from the MOGENTES project[3].

The UML-based device model of AVL489 was engineered by a test user with the help of various domain experts and served as an input for the MoMuT test case generator via an intermediate language called OOAS [7]. MoMuT generated a series of abstract testing procedures in the so-called Aldebaran file format [8]. These abstract test procedures were ultimately converted to concrete NUnit tests which are executable on the TAF. However, it turned out that the typical test engineer has a very weak background in UML semantics. In addition, currently available tools hardly provide any support in guiding the users in using the right UML semantics [1] for their intended purpose. On the other hand, even a UML-experienced modeller had a hard time obtaining the necessary information because UML's abstraction approach did not mirror the test engineers' intuitive way of thinking, which differs from standard UML state diagrams and turned out not to be sufficiently represented by UML elements.

## 2 Introducing a domain-specific language

As we faced these challenges, we decided to create a domain-specific language which is specifically tailored to MD behaviour modelling. This DSL is being designed in the ongoing TRUCONF project[4]. TRUCONF is a follow-up to TRUFAL and will extend the automated test case generation to incorporate the functional and non-functional aspects of MDs. An overview of the TRUCONF toolchain is given in Figure 2. The non-functional cost models (e.g. CPU load, memory consumption, data rates, etc.) will be mostly obtained via model learning. (This is currently subject of research and would thus go beyond the scope of this paper.) Therefore, the DSL currently focusses on the modelling of functional behaviour. Such a language has to satisfy the following requirements:

- The DSL has to be intuitively understandable to its end users, even if they have no prior modelling experience. This seems to be the most important requirement for the DSL's industrializability since the TRUFAL project has shown that ignoring this leads to a strong rejection of the suggested model-based testing approach by the test engineers.

[2] https://trufal.wordpress.com/ (2016.07.29)
[3] http://www.mogentes.eu/ (2016.07.29)
[4] http://truconf.ist.tugraz.at/ (2016.07.29)

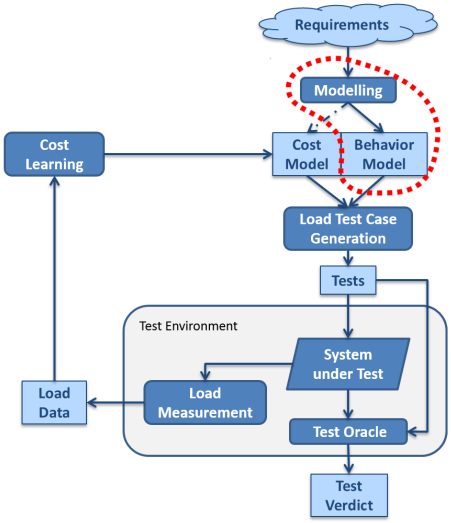


**Figure 2.** TRUCONF testing toolchain for measurement devices. The impact of our DSL will be mostly confined to the area indicated by the red dashed line.

- The DSL has to provide an efficient way to create MD models in a compact and expressive form with semantically well defined model elements.
- The DSL has to enable the generation of meaningful test cases based on partial MD models. More specifically, the test case generation needs to yield meaningful results, even if the model only contains a subset of all inputs allowed by the system under test (SUT). But for all allowed inputs the SUT's output has to conform to the output specified in the model. Compare the Input-Output Conformance (ioco) relation [9, 2]. (Currently, our test cases only test the presence of required outputs and not the absence of forbidden ones, but the testing mechanism can be modified to test for full ioco.) Furthermore, distributing an MD's functionality over several models should greatly facilitate variant testing.

### 2.1 Language design

In accordance with the test engineers an initial approach of developing a textual DSL based on Gherkin [10] was evaluated. This language is designed for behaviour-driven development and allows simply structured scenario descriptions in natural language. Colombo et al. have previously used Gherkin to generate test cases based on state machine models [11]. In their approach the authors describe each transition of their state machine with a Gherkin scenario. For each scenario they use the keyword `Given` to denote the current state, the keyword `When` for external triggers and the keyword `Then` for the next state. In our case the external triggers represent the AK commands [12] that are used to control the device remotely. For example:

```
Scenario: Go from Pause to Standby
  Given the device is in State Pause
  When I send the signal STBY
  Then the device should be in State Standby.
```

As a basis for the language design, a Gherkin model of AVL489 was developed. After that the Gherkin-based lan-

guage design was iteratively refined and optimized in close interaction with its end users to describe the MD in a concise and easily understandable way. The explicit naming of scenarios was omitted because it was deemed redundant. The natural-language formulation of current state, trigger and next state was replaced by a compact formal notation. For instance, since there could be many transitions originating from one current state, the scenarios could be grouped in blocks with a common `Given` statement. The semantics of the `Then` statement was slightly strengthened to describe the MDs reaction, rather than a postcondition. With these changes the model length was reduced by more than half in the very first design iteration. After some additional syntactic fine-tuning we arrived at the following notation:

```
given DeviceState = Pause {
  when Action = STBY then DeviceState -> Standby;
  when Action = SMES then DeviceState -> Measure;
  when ... }
```

This example includes the same transition as the previous one, as well as another transition to the state `Measure` and also hints at other possible transitions leaving the state `Pause`. If the state machine encompasses several orthogonal state variables, the `given` blocks can be cascaded to form a decision tree. To reflect the high degree of purpose orientation, we named our language "Measurement Device Modelling Language" (MDML). In this paper, we merely show a glimpse of MDML, as a detailed language description would go beyond the scope of this work.

## 2.2 Implementation

To complement our purpose-driven language design, we wanted to supply the test engineers with an editor that supports them in their concrete work as extensively as possible. Thus we decided to implement a plugin for the widely used Eclipse IDE [13] by means of the Xtext framework [14]. Xtext automatically creates an Eclipse-plugin from a grammar definition. This plugin provides a parser, as well as syntax highlighting and simple syntactic quick fixes out-of-the-box. Moreover, it highly facilitates the implementation of other features like auto-completion, code formatting, and tool tips. Such features greatly assist the test engineer in building a syntactically and semantically correct model and improve on the perceived lack of user-friendliness during the TRUFAL project.

As we did with our language design, we put the end users in the very center of our tool integration process. We encourage the test engineers to gain user experience and give feedback that will allow us to iteratively improve the IDE.

# 3 Results

## 3.1 User feedback

After its implementation, a prototype MDML IDE was made available to the test engineers, as well as a trial group who had no previous involvement in MD development and testing. All users agreed that MDML is very comprehensible and easy to learn. All test users were able to generate an initial measurement device model based on its manual within an hour. This is an important improvement, since the mere introduction to the UML IDE Visual Paradigm [15] and the used UML semantics took significantly longer. One test user was tasked with the modelling of several MDs over an extended period of time. Within a week, he created extensive models of all the devices that were assigned to him. The models were based on the behaviour specification in the MD manuals and cross-referenced with the behaviour of the virtual TBSimu devices. However, a final judgement of the model quality can only be made when the generated test suites are applied to the real devices.

On the other hand, the test users expressed the wish for several concrete IDE features: It would be helpful to provide the test engineers with some kind of graphical feedback to assess the degree of completion of the current model at first glance. Here several aspects of UML state diagrams could be reused, but in a use case-tailored manner and rather as an optional view with limited editing features, than a first class input language. Furthermore, the semantical correctness of an MDML model is not fully defined by its grammar. Thus there is a strong need for a model validation feature. For example it has to be checked if the decision tree allows outgoing transitions for all possible states and that no dead ends are produced. The validation result can be given back to the user in the form of a graphical or tabular view. Moreover the validator has to ensure that the model does not contain multiple `when ... then` statements for the same input that are allowed under the same conditions, which would produce non-deterministic behaviour.

## 3.2 Selected key differences to UML

MDML aims to improve on the practical and technical shortcomings of the previously used UML approach. However, MDML's language design is still in progress and it does not yet support timed behaviours. A selection of key differences between UML and MDML in the context of our use case is given below:

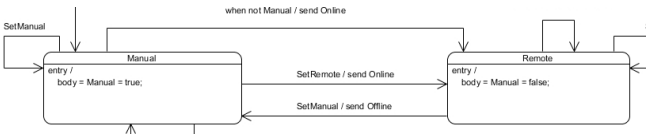


**Figure 3.** The `Manual/Remote` sub-state machine of AVL489.

### *3.2.1 Concrete examples*

1. If the state machine contains several parallel sub-state machines, the UML semantics requires the user to incorporate hidden boolean variables for communication between these sub-state machines. This issue is best described in the form of an example: Figure 3 shows the `Manual/Remote` sub-state machine of the AVL489 device. The MD can only receive commands from PUMA when it is in the state `Remote`. Thus all the inputs that change the states of a parallel sub-state machine (e.g. `Standby`, `Measure`, etc.) can only be accepted if this condition is fulfilled. However, the UML semantics does not allow for direct communication between the respective sub-state machines. Instead, the transition guards depend on boolean variables that communicate the

respective states. These variables have to be explicitly set in the states' `entry` actions. For example, in the `entry` action of the state `Manual`, the `Manual` variable is set to `true`, while it is set to `false` in `Remote`. This appears somewhat strange as it reads like "When you are in `Manual`, then become `Manual`". These internal variables are not visible at first glance, so they reduce the readability and somewhat undermine the graphical nature of the model. In MDML this shortcoming does not occur, as the decision tree can be made directly dependent on all state variables present in the model, for example:

```
given ConnectionState = Remote {
  given DeviceState = Standby {
    when ... }
  given ... }
```

2. In addition to the state diagram, the UML approach requires several class diagrams to describe the interface between the system under test and the environment, which explicitly defines all possible inputs and outputs. In MDML this information is replaced by a short header segment that defines the state variables and inputs in the following way:

```
public statevar DeviceState {Pause, Standby,
  ...} = Pause;
input Action {SPAU, STBY, ...};
```

State variables are defined by their name, their range and their initial state. The keyword `public` denotes that its current state is visible to the environment. Input symbols are explicitly defined and can be grouped into named input channels, e.g. "`Action`". Currently, MDML does not allow the explicit definition of outputs, but this feature can be added, should the need arise.

#### *3.2.2 Maintainability*

The debugging and fine-tuning of the UML model of AVL489 took up to 40 hours. This is due to the fact that the accurate representation of all small details turned out to be difficult and sometimes required a complete model revision. Due to the improvements mentioned above and to the guidance that our MDML IDE aims to provide, we hope to greatly reduce the necessary effort. It is also worth mentioning that all elements in an MDML model are immediately visible to the user while important aspects of a UML model can be hidden in various property menus of the respective editor. Hidden information can be a desirable feature under some circumstances, for example in a graphical representation designed to show an overview, but it can also pose a great challenge to an inexperienced user.

#### *3.2.3 Timing behaviour*

UML represents timing behaviour in a very intuitive manner. Although timing is an important part of most MDs' behaviour, MDML is currently lacking a representation of timed transitions. At the moment we treat timed actions as a kind of input (e.g. "the user waits for 20 seconds and the device reacts to the user's inaction"). However, this leaves a gap in MDML's semantics, as it is not defined at which point the time frame starts. An important part of our current work is to come up with a clearly defined representation that mirrors the intuitive timing semantics of UML state diagrams.

### 3.3 Conclusion and outlook

We have created a language design that proved to be highly understandable to test engineers and is very easy to learn. Their current feedback is quite promising and their involvement in the DSL design process has helped to increase the DSL's acceptance. While the textual model representation gives the user great control over small details, there is still a need for a graphical representation to show the big picture. The implementation of both a textual and a graphical representation are important goals of our ongoing TRUCONF project. Furthermore, the Eclipse-based IDE will be extended to encompass a validation of the MDML model in terms of coverage and unambiguousness. While MDML solved several issues of the UML approach, its language design is still incomplete. The most important addition will be a syntax and semantics for timing behaviour. This and other improvements are subject of our current work. Most importantly, though, MDML will ultimately have to prove its worth for industrial test case generation. An upcoming publication will describe the conversion of MDML into the intermediate language OOAS and the test case generation process.

## REFERENCES


[1] W. Krenn, R. Schlick, and B. K. Aichernig, "Mapping UML to labeled transition systems for test-case generation: A translation via object-oriented action systems," vol. 6286 LNCS, pp. 186–207, 2010.

[2] E. Joebstl, *Model-Based Mutation Testing with Constraint and SMT Solvers*. Dissertation, Graz University of Technology, 2014.

[3] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korošec, W. Krenn, R. Schlick, and B. V. Schmidt, "Model-based mutation testing of an industrial measurement device," vol. 8570 LNCS, pp. 1–19, 2014.

[4] "AVL PUMA Open Automation Platform." `https://www.avl.com/-/avl-puma-open-automation-platform`. (2016.07.29).

[5] J. Auer, *Automated Integration Testing of Measurement Devices*. Bachelors thesis, Graz University of Technology, 2013.

[6] "NUnit Framework." `http://www.nunit.org/`. (2016.07.29).

[7] S. Tiran, "The Argos Manual," tech. rep., 2012.

[8] "Aldebaran manual page." `http://cadp.inria.fr/man/aldebaran.html`. (2016.07.29).

[9] J. Tretmans, "Test Generation with Inputs, Outputs and Repetitive Quiescence," *Software-Concepts and Tools*, vol. 3, pp. 103–120, 1996.

[10] "Gherkin." `https://cucumber.io/docs/reference#gherkin`. (2016.07.29).

[11] C. Colombo, M. Micallef, and M. Scerri, "Verifying Web Applications: From Business Level Specifications to Automated Model-Based Testing," *Electronic Proceedings in Theoretical Computer Science*, vol. 141, no. Mbt, pp. 14–28, 2014.

[12] K. Jogun, "A Universal Interface for the Integration of Emissions Testing Equipment Into Engine Testing Automation Systems: The VDA-AK SAMT-Interface," *SAE Technical Paper*, 1994.

[13] "Eclipse IDE." `http://www.eclipse.org/`. (2016.07.29).

[14] "Xtext Framework." `https://eclipse.org/Xtext/`. (2016.07.29).

[15] "Visual Paradigm." `https://www.visual-paradigm.com`. (2016.07.29).