Christian Burghard, BSc

Model-based Testing of Measurement Devices Using a Domain-specific Modelling Language

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematik

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig

Institute of Software Technology

Graz, April 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

Datum	Unterschrift

Abstract

The practice of model-based testing finds increasing application in industry, due to its potential to cope with the ever rising complexity of technical systems. For this reason, the AVL List GmbH is introducing a model-based testing methodology for the application to its portfolio of automotive measurement devices. In a previous project by AVL, the Graz University of Technology and the Austrian Institute of Technology, a model-based mutation testing approach has been developed. While this approach has been successfully validated in terms of functionality, it was rejected by AVL's test engineers as they deemed its UML-based modelling front-end too difficult to use in their specific industrial setting.

In the thesis at hand, we examine the tool composition-, usability- and experience-related reasons which have lead to the rejection of this modelling approach. We present a textual domain-specific language which we specifically tailored to the sole purpose of modelling AVL's measurement device state machines. To ensure the intended improvement in the user experience of the modelling formalism, we developed the language in close and frequent collaboration with the test engineers. The resulting domain-specific language, called MDML, turned out to be very easy to learn and to be able to efficiently encode measurement device state machine models.

In conjunction with MDML, we further developed a dedicated modelling tool, based on the well-known Eclipse-IDE. As we did with the language, we tailored this modelling tool to our use case and we also enriched it with a number of features providing user guidance and direct connection to AVL-internal data sources. Most importantly, we integrated a test case generation toolchain which we built around the pre-existing MoMuT test case generator. This toolchain involves a model transformation from MDML into object-oriented action systems to serve as input for the generator. It further involves the concretion of MoMuT's abstract test case into executable test code.

Lastly, we show that the capabilities of our model-based testing methodology are at least on-par with those of the previous UML-based one by means of a case study involving the generation and execution of tests for one of AVL's measurement devices.

Keywords: Model-Based Mutation Testing, Domain-Specific Language, State Machines, UML vs. DSL, Test Case Generation, User Experience, Model Transformation, OOAS.

Kurzfassung

Aufgrund ihrer Fähigkeit, die stetig wachsende Komplexität technischer Systeme beherrschbar zu machen, findet die Praxis des modellbasierten Testens zunehmende Anwendung in der Industrie. Demzufolge führt die AVL List Gmbh eine modellbasierte Testmethodik zur Anwendung auf ihr Portfolio an Messgeräten für den Automotivbereich ein. Aus einem früheren Projekt der AVL, der Technischen Universität Graz und des Austrian Institute of Technology ging ein modellbasierter Mutationstest-Ansatz hervor. Obwohl sich dieser Ansatz aus funktionaler Sicht bewährte, wurde er von AVLs Testingenieuren abgelehnt, da sie ihn in ihrem speziellen Umfeld als zu kompliziert empfanden.

In der vorliegenden Arbeit untersuchen wir die toolkompositionalitäts- usability- und erfahrungsbezogenen Gründe, die zu der Ablehnung des Modellierungsansatzes geführt haben. Wir stellen eine domänenspezifische Sprache vor, die wir speziell darauf zugeschnitten haben, die Zustandsmaschinen von AVL-Messgeräten zu modellieren. Um die angestrebte Verbesserung der User Experience sicherzustellen, haben wir die Sprache in naher und häufiger Zusammenarbeit mit den Testingenieuren entwickelt. Die daraus hervorgegangene Sprache, genannt MDML, erwies sich als sehr leicht erlernbar und fähig, Zustandsmaschinenmodelle von Messgeräten effizient auszudrücken.

Zusätzlich zu MDML haben wir, basierend auf der bekannten Eclipse IDE, ein spezialisiertes Modellierungstool entwickelt. So wie die Sprache haben wir auch das Tool auf unseren Anwendungsfall zugeschnitten und es außerdem mit Fetures ergänzt, die die Benutzerführung unterstützen sowie die direkte Kopplung mit AVL-internen Datenquellen erlauben. Das wesentlichste Merkmal ist die integrierte Testfall-Generierungs-Toolkette, die wir um den vorab vorhandenen Testfallgenerator MoMuT herum geschaffen haben. Diese Toolkette beinhaltet eine Modelltransformation von MDML zu objektorientierten Action-Systemen um Verwertbarkeit durch den Testfallgenerator herzustellen. Außerdem beinhaltet sie die Konkretisierung von MoMuTs abstrakten Testfällen zu ausführbarem Testcode.

Anhand einer Fallstudie zur Testfallgenerierung und -ausführung an einem der AVL-Messgeräte zeigen wir letztendlich, dass unsere modellbasierte Testmethodik funktionell zumindest gleichwertig zu der vorherigen UML-basierten Methodik ist.

Schlagworte: Modellbasiertes Mutationstesten, Domänenspezifische Sprache, Zustandsmaschinen, Vergleich UML zu DSL, Testfallgenerierung, User Experience, Modelltransformation, OOAS.

Acknowledgements

This work would not have been possible without the support of a number of people whom I would like to thank:

First of all, I want to thank my supervisor Prof. Bernhard K. Aichernig who, despite my frequent disappearance into AVL-internal business, reliably helped me in navigating the shallows of writing this thesis.

I would also like to thank Robert Korošec who constantly managed to keep the balance between AVL's business interests and my thesis-related needs. He always had an open ear for me and is a personal role model of mine. My gratitude also goes to Gerald Stieglbauer for providing me with an adequate learning curve throughout my work and for the occasional push beyond my previous limitations.

Furthermore, I would like to thank the team of AVL's *systems engineering laboratory* for their tireless and eager support in technical questions and other "geeky" subjects. Special thanks goes to Matthias Seidl who brought AVL's measurement device testing initiative to my attention in the first place.

Most of all, I want to thank my parents Gerlinde and Kurt for giving me the opportunity for a good education and for being my lovely mom and dad.

Christian Burghard
Graz, April 4th, 2018

Contents

List of Figures

Abbreviations

AK	Standardization of Exhaust Measurement (**Ger.**: Standardisierung Abgasmesstechnik)
AIT	Austrian Institute of Technology
BFS	Breadth-First Search
CDH	Configurable Device Handler
CNL	Controlled Natural Language
DKB	Device Knowledge Base
DSL	Domain-specific Language
DSML	Domain-specific Modelling Language
EBNF	Extended Backus-Naur Form
EGPML	Extensible General-Purpose Modelling Language
EMF	Eclipse Modelling Framework
IDE	Integrated Development Environment
IOCO	Input-Output Conformance
IOTS	Input-Output Transition System
LTS	Labelled Transition System
MBE	Model-based Engineering
MBT	Model-based Testing
MBMT	Model-based Mutation Testing
MDE	Model-driven Engineering
MDML	Measurement Device Modelling Language
NFR	Non-functional Requirement
OOAS	Object-oriented Action System
RRT	Rapidly exploring Random Tree
SMT	Satisfiability Modulo Theories

SSoT	Single Source of Truth
SUT	System Under Test
SysML	System Modelling Language
TAF	Test Automation Framework
TBSimu	Testbed Simulator
TFMS	Testfactory Management Suite
TCT	Test Case Transformator
UML	Unified Modelling Language
VNC	Virtual Network Computing

1 Introduction

1.1 Motivation for Model-Based Testing

Over the past decades, technical systems have exhibited a tremendous increase in complexity and will most certainly continue to do so in the foreseeable future. If left unaddressed, system complexity would eventually outgrow the capabilities of domain experts and of the tools at their disposal [44]. In spite of this trend, system complexity must still remain manageable to ensure product quality in a time- and cost-efficient manner. *Model-driven engineering*[1] (MDE) has established itself as a sustainable methodology to adequately tackle this problem [70]. It takes the burden of rising complexity off the engineers' shoulders by having them interact with abstract system models rather than with concrete systems directly. The information content within these models has been reduced to its very essence so that the task of translation between the abstract problem domain and the concrete implementation domain no longer has to be performed ad-hoc by the engineers.

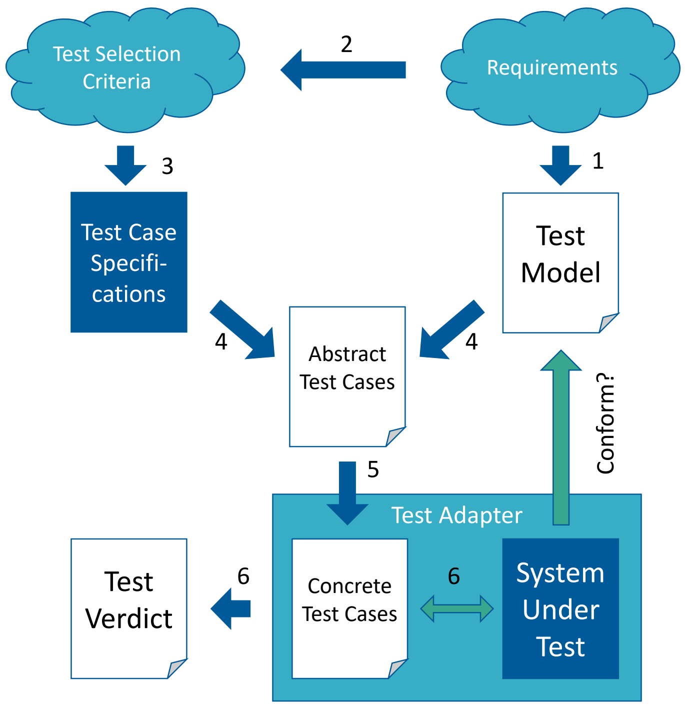


Figure 1.1: The general model-based testing process, based on [103, p.3].

[1] Also known as Model-based engineering (MBE)

All these demands also fully apply to the process of *system testing.* As a sub-discipline of MDE, *model-based testing* (MBT) is a wide research and development field, comprising a vast amount of test derivation strategies which are applied to an equally vast amount of model domains and applications [37, 103]. As varied as this field may be, the process of model-based testing can in many cases be generalized to the form depicted in Figure 1.1, comprising the following steps [103]:

1. The input of the model-based testing process is generally a set of *requirements* which apply to the *system under test* (SUT). These requirements are compiled into a *test model* which describes the SUT's intended behaviour. This modelling process is kept separate from the SUT's implementation process. In this way, a redundant specification is produced which has a lowered probability of sharing common faults with the SUT. Pretschner and Philipps [84] have pointed out that it would theoretically be possible to generate both the test cases and the system under test from the same model or to extract the testing model from the SUT. However, in both cases, the SUT would only be tested against itself which precludes the generation of meaningful verdicts.
2. In order to test the requirement conformance of the SUT to the extent prescribed by the overall *test policy*[2], well-suited *test selection criteria* must be chosen to guide the test case generation process. These criteria can be based on the structural coverage of the model (e.g. of states, transitions, statements, conditions, paths or different combinations thereof), on the coverage of previously defined requirements or data spaces or on the exclusion of predefined faults. Generally, tests could also be selected on a random basis or ad-hoc by a test engineer.
3. The test selection criteria are concretized into *test case specifications.* Depending on the criterion, these specifications include the coverage of a concrete set of states, transitions, requirements, data values, faults, etc.
4. A set of *test cases* - also known as a *test suite* - is generated from the test model and the test case specifications. Note that generated test cases and specifications generally exhibit an $m : n$ relationship. This means that one test case can cover multiple test case specifications and one test case specification can be covered by multiple test cases. A test suite of *high quality* will cover a high number of test case specifications with a low number of test cases. It is possible that the test suite contains no test case which covers a certain specification. This can be due to the specification being unfulfillable in principle which calls for the revision of the requirements and the test model. Alternatively, the problem of finding a test case for this particular specification could be computationally too expensive to solve.
5. Per definition [94, p.132], a model always constitutes an *abstract* representation of the original, merely including a limited amount of information. As yet, the generated test cases still reside on the high abstraction level of the test model. To be executable against the SUT, they must be transformed into *concrete* test cases which include the previously abstracted information - e.g. technical specifics of the SUT or the test environment which are of no direct relevance to the test selection criteria. Alternatively, the abstract test cases are forwarded directly to the *test adapter* which

[2] "*A high-level document describing the principles, approach and major objectives of the organization regarding testing.*" [50, p.61]

performs the concretion on-the-fly during test case execution.

6. The test adapter instruments the SUT to enable the execution of test cases. Depending on the nature of the SUT, the test adapter can be of a physical or purely virtual nature. Although the test execution process can be automated in many cases, the test adapter can also involve partial or exclusive human interaction (e.g. in the case of a test driver on a vehicle testbed). Each test case is executed against the system under test with the intent of falsifying the assumption that the SUT conforms to the test model. If the SUT passes all test cases of the test suite, this falsification has failed, thereby increasing confidence in the conformance of the SUT to the test model. As its end product, the model-based testing process yields a test verdict which includes information about passed and failed test cases and perhaps detailed failure reports. If a fault-based test selection strategy has been chosen, the test verdict also allows to reason about the absence of certain faults within the SUT.

The arguments for model-based testing boil down to a few key advantages: Through abstraction and elimination of redundancy, the effort to create and maintain a test suite for a given system is greatly reduced. It is generally easier to define the components of a complex system than defining a representative set of system traces. A localized change in a system specification can easily translate to numerous changes in the corresponding test suite. The manual application of these changes is a time-consuming (and therefore costly) and very error-prone process. However, one change in the specification will in most cases only result in a single change in a test model if the abstraction level is well-chosen. Through the application of model-based testing tools, the issue of test coverage is effectively reduced to the question of model completeness, the chosen coverage metric and the quality of the test case generation algorithm. A well-designed tool will eliminate the need for ad-hoc coverage analysis and provide the test engineer with a detailed coverage report of the test suite.

Nevertheless, the application of model-based testing also requires a distinct skill set on the test engineers' part - particularly, the knowledge of modelling notations [37]. Minimizing the required skill for the application of MBT (or MDE in general) means minimizing the semantic gap between the the test engineers' mental understanding of the problem domain and the modelling formalism at hand [97]. Over the past two decades, two schools of thought have emerged on how this minimization can be achieved [44, 70]:

Extensible general-purpose modelling languages (EGPMLs) are domain-independent but offer meta-modelling functionalities which allow the modellers to tailor the modelling notation to their specific problem domain. The prime example of such a language is the *Unified Modelling Language* (UML) which has become a de-facto standard for industrial applications [72]. In a survey on model-based testing approaches performed by Dias Neto et al. [37] in 2007, about one in four surveyed approaches was based on UML. Due to its high prevalence and standardization [48], UML has the advantage to be understandable to a large number of engineers across different application domains [44].

A fundamentally different approach is the creation of *domain-specific languages* (DSLs) [60, 44, 34] which are engineered from scratch to fit the individual requirements of their user bases and application domains. Such application domains can reach from functional and non-functional software testing [87, 25] over automotive battery development [82] to home automation [55], landscape dynamics [42] and marine biology [58], to name a few. Compared to EGPMLs, DSLs tend to require a higher effort for language and tool design and maintenance but offer the possibility to reduce the semantic gap to near-zero.

1.2 Research Problem

The AVL List GmbH is the leading developer of instrumentation and test systems for automotive applications and supplies companies in the automotive industry all over the world. Such instrumentation and test systems include testbeds for engines, powertrains, batteries, inverters, chassis and e-motors. All testbed types are complex systems of systems which include a certain set of measurement devices, specific to their respective application and customer needs.

In the recent past, AVL has made an effort to apply model-based testing methods to its measurement device portfolio. In cooperation with the Austrian Institute of Technology and the Graz University of Technology, a model-based mutation testing method [6, 56, 18] has been developed to facilitate the integration testing process of individual measurement devices into the testbed control system. While the testing method has been verified, the initial industrialization effort failed due to the poor reception of the corresponding modelling formalism.

In this thesis, we present a domain-specific modelling language (DSML) which we created as a front-end formalism for the previously existing test case generator to further the industrializability of the model-based testing approach. We tailored this DSML specifically to AVL's application domain. We developed it in close and frequent collaboration with its intended user base to ensure an appropriate learning curve through intuitive understandability, which in turn increased its user acceptance. Moreover, we integrated the language into a specialized tool environment and incorporated it into the test case generation toolchain. To reflect its domain-specific nature, we named the language "Measurement Device Modelling Language" (MDML).

1.3 Research Projects

The work presented in this thesis was conducted as part of - or influenced by - a series of different research projects. Specifically, the author was involved in the projects TRUCONF and DLUX, building on ground work laid in the MOGENTES and TRUFAL projects. All of these projects are described below:

1.3.1 TRUCONF

TRUst via COst function driven model based test case generation for Non-Functional properties of systems of systems[3] (2014-2018)
This recent research project was conducted by the Austrian Institute of Technology, AVL and the Graz University of Technology. It focused on increasing the ease of functional testing through the development and application of easily accessible modelling languages, as well as on the development of testing methodologies for non-functional properties of complex systems. The automated integration testing process of automotive measurement devices was chosen as a use case of this project, primarily focusing on the modelling language aspect. The work presented in this thesis was mostly conducted as part of this use case. The development of a non-functional testing methodology for load data of an automotive web-service was chosen as another use case for the project. More details on this use case can be found in Section 10.2.3. The work on this project was funded by the *Austrian Federal Ministry of Transport, Innovation and Technology* under the program "*ICT of the Future*".

1.3.2 DLUX

Domain-specific Language User eXperience[4] (2017-2018)
This project was started during the later phases of TRUCONF by AVL and the Vienna University of Economics and Business. It focused on the development of empirical methods to assess the *user experience* of software tools. These methods were applied to different in-house development tools within AVL. The modelling environment for MDML was chosen as one such use case. Some parts of this work - especially those described in Chapter 5 - were conducted as part of the DLUX project. Another use case of DLUX concerned itself with a graphical state machine editor which was used within AVL's measurement device development department. The work on this project was funded by the *Austrian Federal Ministry of Transport, Innovation and Technology* under the program "*ICT of the Future*".

[3]`http://truconf.ist.tugraz.at/`
[4]`https://dlux.wu.ac.at/`

1.3.3 MOGENTES

MOdel-based GENeration of Tests for dependable Embedded Systems[5]
(2008-2011)
MOGENTES was a large multinational project within the 7th EU framework program. Its goal was to improve the automatic testing and verification of embedded systems and increase confidence in safety-relevant components. It saw the initial development of the MoMuT::UML test case generator by the Austrian Institute of Technology and the Graz University of Technology. AVL was not involved in this project.

1.3.4 TRUFAL

TRUst via failed FALsification of complex dependable systems using automated test case generation through model mutation[6] (2011-2014)
This project constitutes the thematic link between the MOGENTES and TRUCONF projects. It was conducted by the Austrian Institute of Technology, AVL, the Technical University Graz and Thales. Here, model-based testing methodologies were first applied to AVL's measurement devices. The project resulted in a proven functional testing methodology but nevertheless a failed industrialization thereof. This lead to the inception of the following TRUCONF project. The project was funded by the *Austrian Federal Ministry of Transport, Innovation and Technology* under the program *FIT - IT - Trust in IT Systems*

1.4 Published Material

During his work on this thesis, the author has been involved in the creation of two thematically related papers:

Introducing MDML - A Domain-specific Modelling Language for Automotive Measurement Devices
This paper gives a coarse overview of the contents provided in Chapter 3 as well as selected information from the Chapters 4 and 5. It was co-written with Gerald Stieglbauer and Robert Korošec and presented by the author of this thesis at the *International Workshop on Digital Eco-Systems* co-located with the *28th International Conference on Testing Software and Systems (ICTSS) 2016* in Graz, Austria [28].

A Daily Dose of DSL - MDE Micro Injections in Practice
This paper describes an agile method for the introduction of model-driven engineering methodologies into an industrial environment. This introduction strategy - called *MDE micro injections* - is illustrated by example of the TRUFAL, TRUCONF and DLUX projects. The paper contains selected information from Chapters 3 and 5. Otherwise, this paper is only tangentially related to the contents of this thesis as it deals with the introduction of our MDE methodology rather than with the methodology itself. It was

[5]`http://www.mogentes.eu/`
[6]`https://trufal.wordpress.com/`

co-written with Gerald Stieglbauer, Stefan Sobernig and Robert Korošec and presented by the author of this thesis (filling in for Gerald Stieglbauer) at the *6th International Conference on Model-Driven Engineering and Software Development MODELSWARD 2018* in Funchal, Portugal [97].

1.5 Toolchain Overview

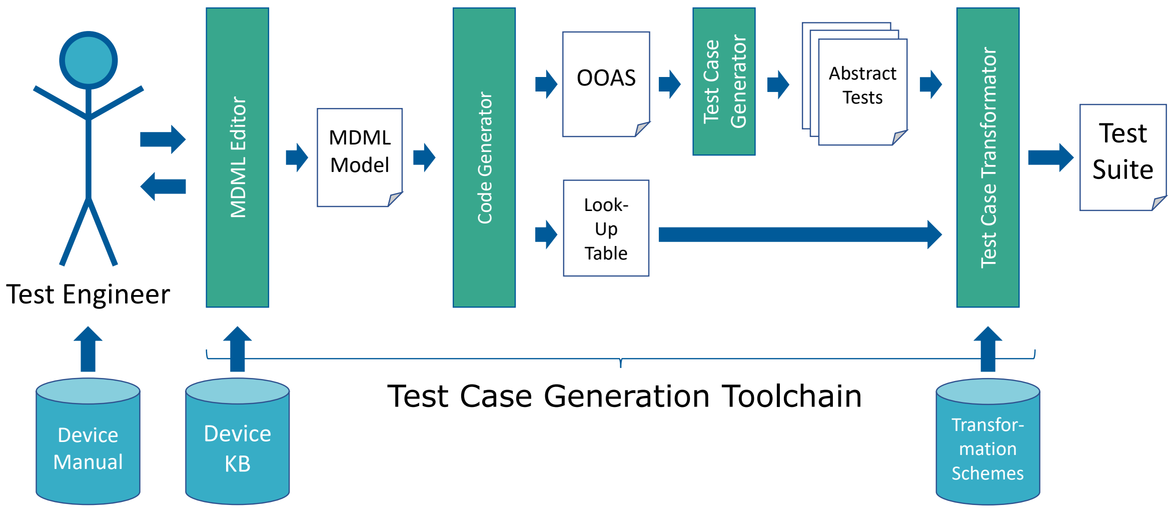


Figure 1.2: An overview of the test case generation toolchain.

The test case generation toolchain developed in the TRUCONF project builds upon the TRUFAL toolchain, described in detail by Auer [18]. The toolchain is depicted in Figure 1.2 and involves the following steps:

1. A test engineer creates an MDML model of a specific measurement device, using a dedicated MDML editor. The content of the model is based on an informal source of information - typically on the device handbook and, if available, development artefacts. In some cases, information about the testing interface can be directly imported from the *device knowledge base* (DKB), a database which is used and maintained by the measurement device development department.
2. A code generator converts the MDML model into an object-oriented action system (OOAS) [26, 99]. This representation encodes the functional contents of the MDML model and serves as the input to the test case generator. Additionally, a lookup table is created which serves as a link between the OOAS model elements and their counterparts in the MDML model. The OOAS formalism is described in more detail in Section 2.1.
3. The MoMuT::UML test case generator[7] [64] reads the OOAS and generates a series of abstract test sequences. The tool was initially designed to read input in the form of UML [48] models but it also uses OOAS as a second input language.

[7]`https://momut.org/`

4. The test case transformator reads the abstract test sequences and compiles them into a test suite. This test suite is a valid C# class and can be incorporated into AVL's test automation framework (TAF). The transformation to C# code is guided by a purpose-specific transformation scheme, which is selected by the test engineer. Information from the original MDML model (such as element names) can be easily retrieved via the previously generated lookup table.
5. The generated test suite is manually incorporated into the TAF. Automation of this and subsequent steps goes beyond the scope of the TRUCONF project and will be subject of future work.
6. Once incorporated, the TAF provides the necessary infrastructure to run the test suite on the corresponding measurement device.

1.6 Thesis Structure

The rest of this thesis is structured as follows: Chapter 2 provides an overview of the underlying concepts of object-oriented action systems and model-based mutation testing. Chapter 3 describes the lessons learned over the course of the TRUFAL project with focus on relevance to TRUCONF. It further describes the motivation for the use of a DSML and elaborates on concrete language requirements. It concludes with an example MDML model of a measurement device. The Chapters 4-8 roughly mirror the toolchain elements described above. Chapter 4 contains a full language specification of MDML with a focus on practical usage. Chapter 5 describes the Eclipse[8]-based integrated development environment (IDE) which has been developed in parallel with the language and serves as a user front-end. Chapter 6 states transformation rules between MDML and OOAS elements, resulting in a formal semantics for MDML. Chapter 7 describes the test case generation process with MoMuT. It focuses on the relevant mutation operators and describes the resulting fault modes in the device models. The quality of the generated test suites is examined in terms of mutation coverage. It also gives examples of the generated abstract test sequences. Chapter 8 describes the transformation process from abstract test sequences to a concrete test suite. It highlights the differences between various transformation schemes and describes the structure of the finished test suite. In Chapter 9, a case study of the test case generation process for the AVL489 *Particle Counter* [20] is presented. The capability of our testing method to uncover faults is directly assessed and compared to the results of the TRUFAL project. Finally, Chapter 10 gives an overview of related work with sections focusing on model-based testing toolchains involving domain-specific languages and textual state machine representations. The last section focuses on future work on our measurement device testing methodology.

[8]http://www.eclipse.org/

2 Preliminaries

2.1 Object-Oriented Action Systems

Object-oriented action systems are based on the action system formalism which was introduced by Back et al. [22, 23, 21] to describe distributed systems in a way that allows the application of a *refinement calculus*. Later, Bonsangue et al. [26] introduced an object-oriented version which can be translated back to simple action systems, thereby preserving their beneficial properties. Several other modifications were made by Krenn et al. [63] who used OOAS as an intermediate language in their MoMuT::UML toolchain [64]. These modifications include the addition of a prioritized composition operator and complex data types like maps or lists. Tiran [99] has provided a praxis-oriented manual for this version of the language. Parts of the formalism resemble Dijkstra's guarded command language [38] and its definition relies on the weakest precondition calculus. Event-B[1], created by Abrial, is another well-known formal method which is based on action systems [2, 3].

2.1.1 Formal Structure

Since we use MoMuT in our work, we adopt a simplified version of the object-oriented action system definition by Krenn et al. [63]:

$$\begin{aligned} AS =_{df} \; |[& \boldsymbol{var}\ \mathbf{V} : \mathbf{T}_V = \mathbf{I}_V \\ & \boldsymbol{methods}\ M_N^1 = M_B^1; \ldots ; M_N^m = M_B^m \\ & \boldsymbol{actions}\ A_N^1 = A_B^1; \ldots ; A_N^a = A_B^a \\ & \boldsymbol{do}\ X\ \boldsymbol{od} \\ ]| & \end{aligned}$$

A single action system AS consists of a vector of variables $\mathbf{V}$ with types $\mathbf{T}_V$ and initial values $\mathbf{I}_V$, a set of m methods and a set of a named actions. Each method consists of a method name M_N^i and a method body M_B^i. Analogously, each named action consists of an action name A_N^j and an action body A_B^j. Methods may have a return value, while actions have none. Furthermore, named actions are guarded commands and can be marked as controllable (`ctr`), observable (`obs`) or internal (no marking). This distinction will play a significant role regarding the conformance relation of the mutation testing approach described in Section 2.2. Since version 3.0 of MoMuT::UML, calls of methods and/or named actions can be arbitrarily cascaded, with the exception that direct or indirect

[1] `http://www.event-b.org/`

Action	**Notation**	$wp(Action, q)$
Sequential Cmp.	$A_1\,;A_2$	$wp(A_1, wp(A_2, q))$
Nondeterministic Cmp.	A_1 `[]` A_2	$wp(A_1, q) \wedge wp(A_2, q)$
Prioritized Cmp.	A_1`//`A_2	$wp(A_1, q)\wedge$ $(\neg gd(A_1) \rightarrow wp(A_2, q))$
Guarded Command	`requires` c`:` A `end`	$c \rightarrow wp(A, q)$
Assignment	y `:=` e	$q[y := e]$
Skip	`skip`	q

Table 2.1: Semantics of basic actions [63, p.191].

recursive calls are forbidden. Lastly, the action system contains a `do-od` block which is a variant of Dijkstra's guarded iteration statement or "repetitive construct" [38]. X denotes an arbitrary[2] action or composition thereof. The `do-od` block will execute this action until its execution guard $gd(X) =_{df} \neg wp(X, false)$ becomes false. Note that this definition of OOAS lacks the aspects of system distribution and object-orientation since the action systems presented in this work make no use of these features. Nevertheless, the overall structure of an OOAS is still dictated by the requirement to accommodate different objects:

$$\begin{aligned} OOAS =_{df} \; |[& \boldsymbol{types}\ \mathbf{T} \\ & \boldsymbol{system}\ SAB \\ ]| & \end{aligned}$$

Each OOAS consists of two parts. The first part is the type definition block which holds the set of type definitions **T**. These types are either derivations of basic data types like integers or enumerations or complex types. However, at least one of the types is a class definition C which is of the form of an action system AS as defined above: Furthermore, each class definition relies on the type definition block for the definitions of variable and method types. The second part of the OOAS is the system assembly block SAB. Generally, the system assembly block consists of a composition of different classes, which represents an analogue composition (and therefore prioritization) of their respective `do-od` blocks. In our case, each OOAS only instantiates a single object and therefore contains only one class definition.

As stated above, the semantics of object-oriented action systems is defined by the weakest precondition calculus. Thus, Table 2.1 shows the semantics of the subset of basic actions which are used in our work. Here, A, A_1 and A_2 denote other basic and/or named actions which enables recursive definition. The symbol $\rightarrow$ denotes logical implication. In case of the *sequential composition*, two actions are executed consecutively. Note that the preceding action can potentially affect the enabledness guard of the succeeding action which, in turn, can affect the guard of the whole composition. Therefore, the preceding action must be precomputed in order to compute the guard of the composition. The sequential composition operator ; exhibits the strongest binding of all composition operators. In case of the *nondeterministic composition*, the system may choose to execute either A_1 or A_2,

[2]Most of the cited sources only discuss nondeterministic composition of actions.

provided that both of them are enabled. While this can generally lead to nondeterministic system behaviour, the OOAS in our use case[3] will be set up in such a way that at most one action will be enabled at a given time. If, on the other hand, several controllable actions are composed in this way, the choice between these actions is up to the outside observer. The nondeterministic composition operator `[]` binds weaker than the sequential composition operator but stronger than the prioritized composition operator. The *prioritized composition* behaves similarly to the nondeterministic composition but will always prefer A_1 if it is enabled. The prioritized composition operator `//` exhibits the weakest binding of all composition operators. The *guarded command* is enabled when the action A is enabled and the boolean condition c evaluates to true. The *assignment* action and the *skip* action are always enabled.

2.1.2 Syntax Example

A short example of an OOAS, taken from Tiran [99, p.6] is shown below:

```
 1 types
 2   Greeter = autocons system
 3   |[
 4   var
 5     done : bool = false
 6   actions
 7     obs HelloWorld = requires done = false :
 8       done := true
 9     end
10   do
11     HelloWorld
12   od
13   ]|
14 system
15   Greeter
```

In this example, the lines 1 to 13 make up the action system definition of the class *Greeter*, which is instantiated in the system definition block in lines 14 to 15. The keyword `autocons` denotes that a single instance of this class is created at system start. Its variable definition block is introduced by the keyword `var` and contains a single variable *done* of type bool and initial value false. The variable definition block would generally be followed by the method definition block (denoted by the keyword `methods`) which is absent in this example. The action definition block (denoted by the keyword `actions`) contains a single observable action named *HelloWorld* which is also included in the `do-od` block. When the system *Greeter* is started, *HelloWorld* is enabled since *done* is initially false. Upon its first execution, the action sets *done* to true. The system terminates after the first iteration, since the execution guard of *HelloWorld* now evaluates to false. More information about this example and object-oriented action systems in general can be found in [99].

[3]Discounting mutated systems.

2.2 Model-Based Mutation Testing

Model-based mutation testing [5] is a powerful testing methodology. It is fault-based, which means that it is able to detect - and therefore guarantee the absence of defined sets of faults. Depending on the nature of these faults, it is able to subsume a number of different test coverage criteria [76]. Model-based mutation testing arose from the combination of model-based testing (described in Section 1.1) with *mutation analysis*.

2.2.1 Mutation Analysis

Mutation analysis [35, 51] is a method to assess the quality of a given test suite by testing its ability to distinguish faulty versions of a system - the so-called *mutants* - from the correct system. Each mutant has been injected with a syntactic change (*mutation*), generally resulting in deviant behaviour. If the mutation produces no observable difference in behaviour, the mutant is said to be *equivalent*. A test is said to *kill* a mutant if it is able to distinguish the mutant from the correct system. Therefore, equivalent mutants can never be killed. The overall quality of the test suite is reflected by its *mutation score* (*ms*):

$$ms =_{df} \frac{|\mathbf{M}_{killed}|}{|\mathbf{M} \setminus \mathbf{M}_{eq}|}$$

Here, $\mathbf{M}$ denotes the overall set of mutants, $\mathbf{M}_{killed} \subseteq \mathbf{M}$ denotes the subset of killed mutants and $\mathbf{M}_{eq} \subseteq \mathbf{M}$ denotes the subset of equivalent mutants. Informally, the mutation score denotes the percentage of killable mutants which are killed by the test suite. Computing the mutation score may be non-trivial since the equivalence of a mutant is generally non-decidable.

The set of mutants is produced by applying a set of *mutation operators* to the original system. Each mutation operator is a transformation pattern for a small syntactic change and generally produces a number of mutants when applied to different parts of the original system. Therefore, each of these simple or *first-order* mutants only implements a small localized fault. However, a *competent* programmer is assumed to primarily implement faults which do not exceed this magnitude. Also, while the test suite is only guaranteed to detect these simple mutants (provided that they have been killed), it is assumed to also detect the presence of more complex faults. These two assumptions were first formulated by DeMillo et al. in the form of the *competent programmer hypothesis* [35, p.34] ...

> “Programmers have one great advantage that is almost never exploited: they create programs that are *close* to being correct!”

..., as well as the *coupling effect* [35, p.35]:

> “Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.”

The coupling effect was empirically proven by Offutt and Jefferson [73] and formally proven for mutations on boolean formulas by Kapoor [59]. On the other hand, Devroey et al. [36] have efficiently performed mutation analysis with *higher-order* mutants which comprise multiple different mutations at once.

An extensive survey of mutation analysis has been provided by Jia and Harman [54].

2.2.2 Input-Output Conformance

The systems which are examined in mutation analysis do not necessarily need to be concrete implementations. Instead, mutation analysis has been applied to software specification models relatively early on by Gopal and Budd [27, 77] although their method required a working implementation, as well as a human tester who supplied the test cases.

Formally, the model domain used in our work can be described as an *input-output transition system* (IOTS) [101]:

$$IOTS =_{df} \langle S, L, T, s_0 \rangle$$

Here, S denotes a set of states, including the initial state s_0. $L = L_I \cup L_O$ denotes a set of labels, consisting of exclusive input labels L_I and exclusive output labels L_O. $T \subseteq S \times (T \cup \{\tau\}) \times S$ is a set of transition relations from an origin state to a target state, accompanied by a label. The label τ denotes an unobservable internal action. The distinction of L_O, L_I and τ directly corresponds to controllable, observable and internal actions of object-oriented action systems. Moreover, each IOTS is required to be *input-enabled*, which means that it must be able to accept any input label in any state.

Tretmans [101] has laid important groundwork for the derivation of test cases from specifications, which are used to test concrete implementations. Arguably, his main contribution is the definition of a number of implementation relations, defining the circumstances under which a concrete implementation conforms to a more abstract specification. The *input-output conformance* or **ioco** relation deserves special mention since it is utilized by MoMuT::UML and therefore strongly related to our work:

$$I \ \mathbf{ioco} \ S =_{df} \forall \sigma \in Straces(S) : out(I \text{ after } \sigma) \subseteq out(S \text{ after } \sigma)$$

Here, I ("Implementation") denotes an input-output transition system. S ("Specification") belongs to a superclass of IOTS called *labelled transition system* (LTS). $Straces(S)$ denotes the set of *suspension traces* of S - the set of all possible sequences of labels $(L \cup \{\delta\})^*$ which are allowed by the IOTS starting from its initial state. The symbol δ denotes the absence of output and can be emitted arbitrarily often in states which have no outgoing transitions labelled with L_O. $IOTS$ after σ denotes the state which is reached by the IOTS after it has traversed the trace σ[4]. $out(X) \subseteq L_O \cup \{\delta\}$ denotes the set of possible output symbols which the IOTS can produce in the state X, or $\{\delta\}$ if no outputs can be produced. Informally speaking, after any suspension trace included in the specification S,

[4]For nondeterministic IOTS, this expression denotes the set of possible states which are reached after σ.

the implementation I can only exhibits outputs which are allowed by S. Note that the distinction of input and output symbols and the requirement for input-enabledness do not apply to a general LTS. This makes the **ioco** relation well-suited for the application in our use case since it only takes traces of the specification into account and ignores traces which include unspecified inputs. This allows **ioco** to describe conformance to a specification which only partially covers the implementation. An in-depth description of **ioco** and other conformance relations has been provided by Tretmans [101].

2.2.3 Test Case Synthesis

The central idea of *model-based mutation testing* (MBMT) is to synthesize test case with the specific goal to kill all mutants of a given model-based specification, thereby maximizing the mutation score of the test suite. Aichernig [5] has developed a general formal framework which allows for the generation of test suites for an implementation, based on mutants of a specification model, thereby combining model-based testing and mutation testing. His theory is general in the sense that it can be applied using an arbitrary transitive preorder relation. Generally, **ioco** is not a transitive relation but if the specification is input-complete, **ioco** reduces to trace preorder, which is transitive [101, 105]:

$$I \leq_{tp} S =_{df} traces(I) \subseteq traces(S)$$
$$\text{with } traces(X) = Straces(X) \cap L^*$$

However, the requirement of input-completeness negates the primary advantage of **ioco** of supporting underspecification. One technique to reconcile both properties is called *demonic completion* [102, 12] and is achieved by making all unassigned input labels point to a newly introduced "undefined" state $s_u \in S$, from where all input labels perform self-loops. This operation preserves both input-enabledness and underspecification, thereby allowing the **ioco** relation to be incorporated into the general theory of model-based mutation testing. This further allows for the test case generation to be reduced to a constraint satisfaction problem concerning the reachability of a state in which the mutant exhibits a forbidden output [5]. As a side-effect of this reachability analysis, a trace to the state in question is discovered. This trace directly translates to a test case which kills the mutant. The solution of this problem in practice has been demonstrated on several tools [10, 12] including *Ulysses* [7], which served as a back-end for MoMuT::UML and used an enumerative exploration strategy. A substantial amount of ongoing research focuses on the efficient solution of this constraint satisfaction problem. To name a few examples, Jöbstl [56] has created a symbolic refinement checker which reduced the runtime by up to 90% compared to *Ulysses*. Later, Krenn and Schlick [62] have rebuilt MoMuT::UML to follow a search-based approach instead of solving a constraint satisfaction problem and to exploit concurrency in mutant exploration. These measures enabled the tool to support industrial-sized models. In a similar, but unrelated effort, Devroey et al. [36] have achieved a significant performance increase by combining all mutants into a *featured mutant model* which has the added benefit of exploring higher-order mutants.

3 A Measurement Device Modelling Language

Some parts of this chapter have already been published in DECOSYS 2016 [28], as well as in MODELSWARD 2018 [97].

3.1 Lessons Learned from the TRUFAL Project

This chapter recapitulates the final results of TRUFAL, as they have been published by Aichernig et al. [6] and/or presented in the TRUFAL evaluation report [61], with a strong focus on the learned lessons which are applicable to the TRUCONF project. These lessons can be divided into the categories of model creation, test case generation and test case execution.

3.1.1 Model Creation

The *AVL Particle Counter* [20] - also known as AVL489 - was used as a benchmark measurement device for the TRUFAL project. This device measures the concentration of particles in vehicle exhaust through their scattering of laser light. AVL's measurement device development process produces no artefacts in the form of device models which would be directly suitable for test case generation. Instead, the model of AVL489 had to be reverse-engineered from documentation and/or specification documents. This reverse-engineering process was experienced as tedious, time-consuming and error-prone. It is also worth noting that large parts of the modelling work were conducted off-site by our project partner, which introduced an additional time delay and the potential for communication errors. Nonetheless, the reverse-engineering process also had the side-effect of familiarizing the modellers with the measurement devices as well as furthering their understanding of the test acceptance criteria [61].

Once a prototype model had been created, it underwent a considerable evolution before it was suitable for test-case generation. An initial model could be created relatively fast but it deviated from the intended behaviour in multiple corner cases. Each deviation required several hours of investigation [61]. Many of these corner cases were not covered by the reference material on which the model was based. Eventually, the mature model covered all relevant corner cases, but severely lacked in readability and understandability.

Even with the the measurement device's intended behaviour fully known, translating it into UML became a challenge to the modellers, as illustrated by the examples in the rest of this section. Some of them have already been addressed in our previous work [28, 97]. The examples are further illustrated by the state machine diagram of the AVL489 measurement device in Figure 3.1.

State machines of AVL measurement devices are generally multi-dimensional. This aspect is translated to UML by subdividing the state machine diagram into several orthogonal regions. These regions are strongly intertwined in terms of their behaviour and the modeller had to take them all into account in order to understand the behaviour of the overall model. Flattening the regions into a one-dimensional state machine was not attempted due to the expected additional workload of recreating the model from scratch.

Furthermore, the model incorporated several internal variables to pass information between the different regions. This aspect can be seen in Figure 3.1 in the form of the variable *Manual* for the Control region (lower left), as well as the variable *Busy* for the Transition region (right). The value of the variable *Manual* is set in the entry actions of the states in the Control region and then used in transition guards of the Operating region (upper left). The value of *Busy* is set in the Operating region, triggering a state change in the Transition region. The usage of variables which need to directly mirror the state of a particular region adds a high and, arguably, unnecessary level of complexity and redundancy to the model. The modeller must be careful to always propagate all necessary information between variables and regions. Furthermore, the purely textual representation of these variables undermines the overall graphical nature of the model.

The semantics of UML regarding self-transitions did not mirror the intended behaviour. When a self-transition (a transition originating and ending in the same state) is performed, the input should be consumed but the exit and entry actions should not trigger. In the final model, nested sub-machine states were used to model this behaviour. This aspect could also have potentially been modelled using internal[1] transitions, but according to the initial definition of the modelling formalism to be used for the MoMuT front-end [65], they were not part of the used UML subset. The distinction regarding self-transitions is absent in the preliminary model depicted in Figure 3.1.

In addition to the state machine diagram shown in Figure 3.1, the UML model also needed to incorporate several other diagrams to specify the test interface: a profile definition for the used stereotypes, other class diagrams for signal definitions as well as test interface specification and a small Object diagram to instantiate the *system under test* (SUT) and the test environment.

At the conclusion of the TRUFAL project, it was proposed to directly create test models as part of the measurement device engineering process instead of having them reverse-engineered by the users. However, this goal has since been relativized, since using the same model both for engineering and testing of the SUT has implications which are detrimental to our use case. Pretschner and Phillips [84] have pointed out that such a setup would rather serve as a test to the model transformation toolchain itself, since the test suite will be

[1]Note that the definition of the word "internal" in the UML standard [48] does not correspond to the one established in Sections 2.1-2.2 and used in the rest of this work.

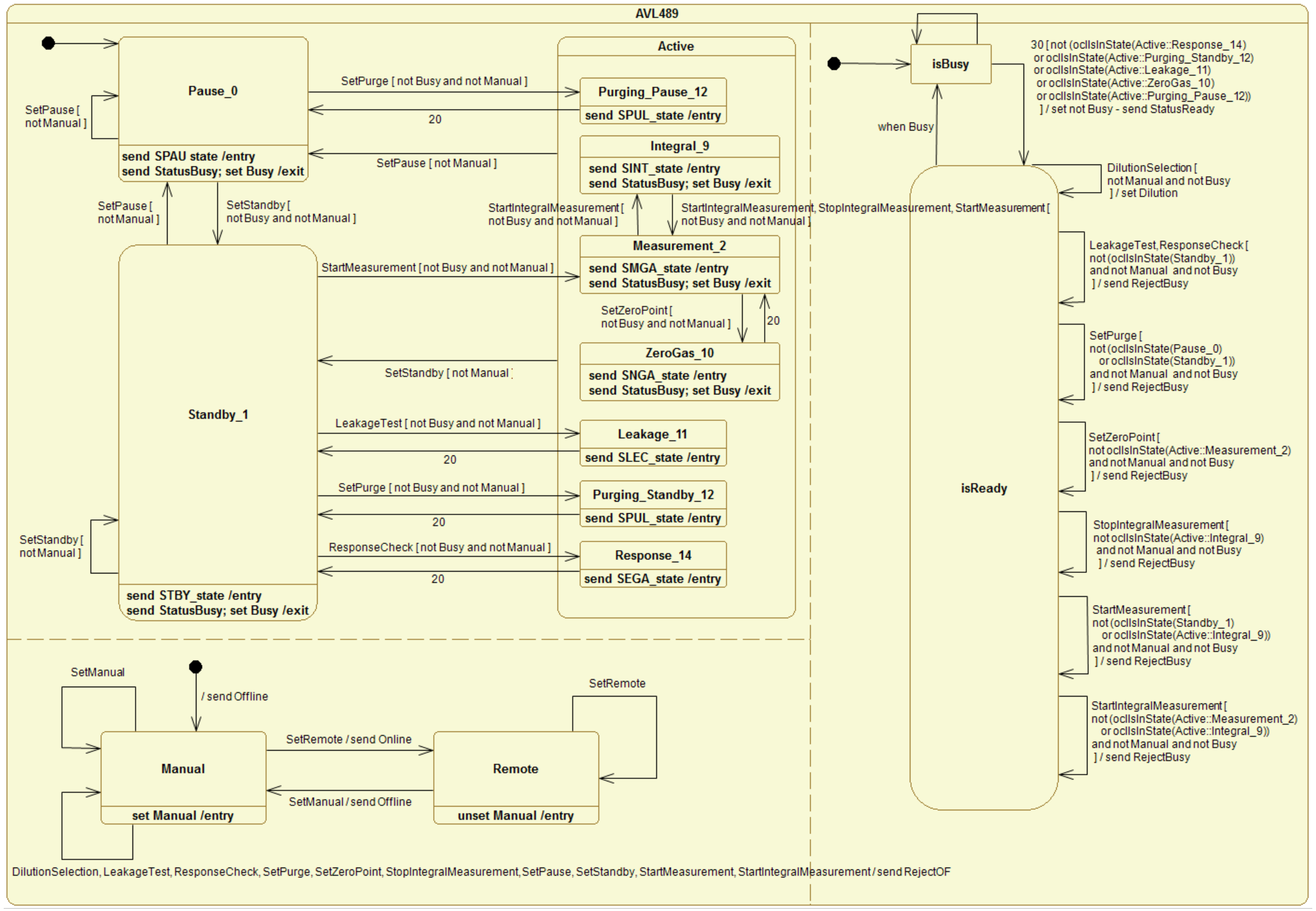

AVL489
Pause_0
send SPAU state /entry
send StatusBusy; set Busy /exit
SetPause [not Manual]
SetPause [not Manual]
SetStandby [notBusy and not Manual]
Standby_1
send STBY_state /entry
send StatusBusy; set Busy /exit
SetStandby [not Manual]
Active
Purging_Pause_12
send SPUL_state /entry
Integral_9
send SINT_state /entry
send StatusBusy; set Busy /exit
Measurement_2
send SMGA_state /entry
send StatusBusy; set Busy /exit
ZeroGas_10
send SNGA_state /entry
send StatusBusy; set Busy /exit
Leakage_11
send SLEC_state /entry
Purging_Standby_12
send SPUL_state /entry
Response_14
send SEGA_state /entry
SetPurge [not Busy and not Manual]
20
SetPause [not Manual]
StartIntegralMeasurement [not Busy and not Manual]
StartIntegralMeasurement, StopIntegralMeasurement, StartMeasurement [not Busy and not Manual]
StartMeasurement [not Busy and not Manual]
SetZeroPoint [not Busy and not Manual]
20
SetStandby [not Manual]
LeakageTest [not Busy and not Manual]
20
SetPurge [not Busy and not Manual]
20
ResponseCheck [not Busy and not Manual]
20
isBusy
when Busy
30 [not (oclIsInState(Active::Response_14) or oclIsInState(Active::Purging_Standby_12) or oclIsInState(Active::Leakage_11) or oclIsInState(Active::ZeroGas_10) or oclIsInState(Active::Purging_Pause_12))] / set not Busy - send StatusReady
isReady
DilutionSelection [not Manual and not Busy] / set Dilution
LeakageTest, ResponseCheck [not (oclIsInState(Standby_1)) and not Manual and not Busy] / send RejectBusy
SetPurge [not (oclIsInState(Pause_0) or oclIsInState(Standby_1)) and not Manual and not Busy] / send RejectBusy
SetZeroPoint [not oclIsInState(Active::Measurement_2) and not Manual and not Busy] / send RejectBusy
StopIntegralMeasurement [not oclIsInState(Active::Integral_9) and not Manual and not Busy] / send RejectBusy
StartMeasurement [not (oclIsInState(Standby_1) or oclIsInState(Active::Integral_9)) and not Manual and not Busy] / send RejectBusy
StartIntegralMeasurement [not (oclIsInState(Active::Measurement_2) or oclIsInState(Active::Integral_9)) and not Manual and not Busy] / send RejectBusy
SetManual
/ send Offline
Manual
set Manual /entry
SetRemote / send Online
SetManual / send Offline
Remote
unset Manual /entry
SetRemote
DilutionSelection, LeakageTest, ResponseCheck, SetPurge, SetZeroPoint, StopIntegralMeasurement, SetPause, SetStandby, StartMeasurement, StartIntegralMeasurement / send RejectOF

Figure 3.1: Preliminary state machine diagram of AVL489 [89, p.21].

effectively blind to all errors present in the model. While an existing *single source of truth* (SSoT) can be used to retrieve the test interface definition, the behavioural model must be a redundant human-made system interpretation to uphold the four-eyes principle.

3.1.2 Test Case Generation

With the test model in place, several test case generation trial runs were performed. During these trial runs, three different test-suites were generated: one test suite (henceforth called **M**) was generated by a computationally expensive mutation-based algorithm. The second one (**R**) was randomly generated. The third one (**C**) was a combined test suite with a number of random test cases and additional mutation-based test cases to cover remaining mutants. Test suite **C** was deemed the most effective in the a-priori evaluation, since it achieved the highest mutation score (81,25%) with the lowest number of test cases (57). The generation of the test suites roughly took 44h, 2h and 68h[2], respectively. Though slightly longer than expected, this result was acceptable since test case generation could be performed over the course of a weekend without human interaction and the process did not have to be repeated very often. Nevertheless, for the test suites **M** and **C**, more than 60% of this time was spent checking equivalent mutants. The trial runs were performed on a computer with two 6-core Intel Xeon processors with 3.47 GHz and 190 GB of RAM, which is far above the system specifications of standard-issue hardware within AVL.

3.1.3 Test Case Execution

In addition to the a-priori evaluation summarized above, all test suites were applied to a set of faulty simulation models of the *AVL Particle Counter* to assess their actual ability to uncover faults. The test suite **R** was deemed the most ineffective, since it exhibited the highest execution time of 1h 36min, as well as the lowest SUT mutation score[3] of 62.5%. Both **M** and **C** had execution times of 29 minutes and SUT mutation scores of 75% and 81.25%, respectively. In any case, the execution times of the test suites were a significant improvement over previously used testing methodologies within AVL. Combined with the results from the previous section, this makes test suite **C** (combined random and mutation-based) the most effective one overall.

Further details on the generation and evaluation of the test suites **M**, **R** and **C** are included in the corresponding paper by Aichernig et al. [6], as well as in Chapter 9 of this work, where test suites generated by the TRUCONF toolchain are evaluated against the same set of SUT faults.

[2]Generation times for **M** and **C** are not comparable due to different exploration depths. Both of them took advantage of 21 concurrent worker threads.

[3]Mutation score in terms of killed SUT mutants

3.2 Motivation for a Domain-Specific Modelling Language

3.2.1 MDE Application versus MDE Introduction

Even though the overall model-based testing methodology had been successfully verified, the modelling formalism was not accepted by the test engineers. Some of the reasons for this rejection have already been stated in Section 3.1.1. However, we argue that these previously mentioned difficulties are symptoms of a deeper underlying reason. To understand this reason, we examine why UML was chosen as the modelling formalism for the MoMuT front-end in the first place. The development of the MoMuT::UML test case generator began during the MOGENTES project in which AVL was not involved. The objective of this project was to improve the reliability of safety-critical systems through automated generation of test cases. During an early project stage, the project participants had to chose adequate modelling formalisms as front-ends for their test case generators. In a report about the chosen modelling formalisms, Kroening [65, p.5] wrote:

> "Instead of inventing new formalisms or forcing academic formalisms on the engineers, we rely on the existing and widely accepted modelling languages Simulink and UML. This approach allows to use the existing, mature tools readily available on the market today. The domain experts at the industrial partners participating in the MOGENTES project already have experience with at least one of these modelling formalisms."

In contrast to the initial users of MoMuT::UML, the test engineers at AVL had little or no previous experience with the UML language or UML tools. In our previous work [97] we used our experience from TRUFAL to argue that one of the root causes for a failed adoption of MDE methodologies is a missing distinction between *applying* MDE in an environment where it is already prevalent and *introducing* MDE in an environment where it has never or scarcely been applied before. During the TRUFAL project, the initial effort to introduce MDE methods had not been anticipated which caused a failure to adopt the model-based testing methodology despite its obvious advantages. However, it is also worth mentioning that extensive efforts to invent new modelling formalisms and/or to customize the modelling tool were well beyond the scope of the TRUFAL project.

To overcome the various technical and social difficulties of an industrial MDE adoption process, Stieglbauer and Rončević [98] proposed the application of a process called *MDE micro injections* - the iterative design, development and introduction of MDE formalisms and tools in close collaboration with the intended user base, executed in a series of many short sprints.

Domain-specific languages are especially well suited for an introduction via MDE micro injections since they generally have good chances of fulfilling their key characteristics, such as *customizability*, *adaptability* and various user experience aspects like *user guidance*, *comprehensibility* or adequate *learning curves*. A DSL can be tailored to the needs of its users, which helps to facilitate (or, in the best case, eliminate) their task of mentally mapping the semantics of the application domain to the semantics of the model domain [60]. In our previous paper [97], we further detailed the application of MDE micro injections as our means to introduce the MDML language and modelling tool in the AVL Test Center over the course of the TRUCONF project.

3.2.2 Textual versus Graphical Notations

The debate about the merits of graphical languages in comparison to textual ones has been going on since at least three decades. While results vary greatly over different problem domains [33, 46], the notion that both formalism types have their own complementary strengths and weaknesses has long been prevalent [83, 34]. While the two initially mentioned studies concluded in strong superiority of graphical and textual notations, respectively, recent eye tracking studies on UML diagrams vs. natural language [53], as well as a requirement modelling language with both textual and graphical representations [92] have found no significant difference between graphical and textual notations in terms of accuracy (efficiency in conveying the correct information). The wide variety of conclusions over different application domains suggests that the question about the merits of textual and graphical languages is not a trivial one and cannot be categorically answered, even with respect to concrete characteristics, such as accuracy or structural clarity.

Before we state the concrete reasons for our choice of modelling formalism, we present some arguments for both graphical and textual notations. Dejanović et al. [34] have compiled a list of advantages and disadvantages of both notation forms, based on their experience. We present a selection of these arguments, comment on some of them from our own experience and make some additional references to the studies mentioned above.

Arguments for Graphical Notations

Structural Clarity: Graphical languages seem to be better suited to convey static structural information about a model. This proposition is largely consistent with our own experience with measurement device state machine diagrams. While not categorically supported by Heijstek et al. [53], they concluded that diagrams can provide a good first impression of a model's structure.

Easy Navigation: Graphical editors tend to allow more navigational freedom, in terms of zooming and panning. Some text editors allow folding of specific text portions and navigation via a model outline while others are limited to scrolling. It is also generally easier to maximize, minimize or hide parts of the model.

Appropriate Learning Curve: Dejanović et al. make the claim that graphical languages are easier to learn, based on the assumption that the graphical editor offers a visible pallet of its model elements to the user and therefore allows for learning by trial and error while the textual editor initially only shows a blank screen. However, we argue that an intuitively comprehensible textual language, combined with well-designed user guidance features (e.g. importable templates, tool-tips, quick-fixes, etc.) of the editor can ensure an equally steep learning curve.

Arguments for Textual Notations

Familiarity: Graphical MDE methods have not been previously used at the AVL Test Center. While the test engineers are not guaranteed to have a background in software engineering, most of them have more experience with textual formalisms than with graphical ones, which should make a textual language more attractive to them.

No Serialization: Textual languages require no serialization since they consist only of a character string which can be directly saved to a file. This allows for the use of version control systems like Git[4], which is already in use at the AVL Test Center. Graphical models, on the other hand, must be converted to character strings in order to be saved to files. Depending on the serialization algorithm, this process may be incompatible with automatic file merging algorithms as used by version control systems, resulting in corrupted files.

Text Editors as Backup Tools: Simple text editors could be used as backup editing tools if no dedicated tools are available. We also point out the related aspect that all elements of a textual model are plainly accessible while some important elements of the UML model created during the TRUFAL project were buried deep within the menu hierarchy of the editor.

3.2.3 Choosing a Textual DSL

While all the previously mentioned factors impacted our use case to some degree, we arrived at our decision by observing the habitual workflows of our test engineers. One of them used a textual notation when designing his test cases with pen and paper. This textual notation turned out to be similar to a behaviour-driven development [93] language called Gherkin[5] [107]. Gherkin describes a system through a set of scenarios written in a subset of natural language. Each scenario reads like an English sentence, structured in the `Given-When-Then` format. The keyword `Given` is followed by a statement which specifies the scenario's precondition. Statements introduced by the keyword `When` specify a trigger action. The scenario definition ends with the keyword `Then`, followed by a postcondition and/or the system's reaction. Colombo et al. [31] have previously used Gherkin to model a state machine representing a web application. They used the `Given` statement to encode the current state of the system, the `When` statement to encode input symbols which trigger a state transition and the `Then` statement to encode the target state and other postconditions, if necessary. When applied to our use case, a single scenario in a measurement device model would look like this:

```
Scenario: Switch from Standby to Measurement
  Given the OperatingState is Standby
    And the TransitionState is Ready
    And the ControlState is Remote
  When the command SMES is received
  Then the OperatingState switches to Measurement
    And the TransitionState switches to Busy.
```

[4]https://git-scm.com/
[5]https://cucumber.io/docs/reference

We decided to follow the same general approach as Colombo et al. regarding the application of the `Given-When-Then` structure to encode state machines and used Gherkin as a baseline for the development of MDML.

To develop the language prototype in parallel with the editor prototype, we used a DSL framework for Eclipse called Xtext[6] [108]. This framework turned out to be well-suited to provide the test engineers with rapid prototypes of both the language and the editor on a regular basis, as prescribed by the MDE micro injection process. More information about the implementation of our dedicated MDML editor can be found in Section 5.2.

Although we chose to follow a purely textual approach for the time being, the idea to incorporate additional graphical model views and thereby combining the best aspects of both worlds soon took shape. We expect this measure to significantly improve the structural clarity of our model representation for the reasons outlined in Section 3.2.2. Such graphical views could be created using compatible Eclipse frameworks such as Sirius[7] or by customizing the existing UML modelling environment for Eclipse, called Papyrus[8].

3.3 Requirements for the Modelling Language

This section contains a list of requirements for MDML, as we have stated them at the beginning of the TRUCONF project. Each requirement is enclosed in quotation marks and followed by a short statement on its fulfilment at the time of project conclusion. The requirements 1, 6 and 7 have already been addressed in our previous work [28].

1. *"The modelling language has to provide an efficient way to create device models in a compact form."*

   Since MDML is textual, all model features are plainly visible and accessible. The language design provides numerous degrees of freedom in terms of model structure and allows the user to choose the most appropriate alternative. The size of a preliminary model rarely exceeds one screen page in the dedicated IDE (see Chapter 5).

---

2. *"The modelling language has to be composed of a textual and a graphical representation. The textual representation facilitates the editing of model details while the graphical representation makes it easy to get an overview of the model. The graphical representation also has to be editable."*

   At the time of this work, no industrializable version of a graphical representation has been created yet. A first attempt at a visualization concept for MDML has been made by Altenhuber [16] who studied the impact of non-editable graphical visualizations on the understandability of otherwise textual models. With minimal cooperation by the author, he created several different visualizations for MDML, including a flattened state machine representation, separate state diagrams for individual state

[6]https://www.eclipse.org/Xtext/
[7]http://www.eclipse.org/sirius/
[8]http://www.eclipse.org/papyrus/

dimensions as well as interactive explorative views. A series of interviews with AVL test engineers showed a slight preference for the former two visualization methods as well as the importance of offering different visualization options according to the individual user's needs. While Altenhuber's study laid important groundwork for a visualization of MDML models, more work is needed to improve the visualization concept according to the test engineers' varied feedback and to create an industrializable implementation. This, however, goes beyond the scope of this thesis.

---

3. *"The modelling language has to be of a modular structure so that the individual devices can be modelled individually (and possibly be composed of several modules) and then combined to create test cases for parallel running measurement devices on one PUMA system."*

   The idea of composing device models of several interacting parts has eventually been dropped. Passing the state variables from one module to another made the language feel too much like a programming language and less like a black-box specification, which would contradict requirement 6. Currently, the language is also not able to describe any kind of inheritance hierarchy between models.

---

4. *"The semantic of the modelling language has to be clearly defined, especially regarding possible parallel executions and priority of individual statements over others."*

   The model transformation from MDML to object-oriented action systems (OOAS) implicitly fulfils this requirement. OOAS has a clearly defined semantics which is described in Section 2.1, as well as in [26, 99]. By relating all language elements to OOAS elements, as described in Chapter 6, the semantics of MDML is clearly defined. The language has been especially designed to exclude prioritization or conflicting statements. This can lead to a slightly increased model complexity but it also spares the user the effort to learn potentially complicated precedence rules similar to the ones applying to UML state machine diagrams [48]. See Sections 4.1.5 and 4.2 for concrete examples of this paradigm.

---

5. *"The modelling language should be iteratively developed in close collaboration with its final users."*

   The language design for MDML was formally conducted from January to September 2015. During this time period, the language has been iteratively developed in close collaboration with test engineers and domain experts. After several prototypes, we arrived at a design which satisfied the expectations of the end users. This process followed the idea of introducing the modelling method to its end users in the form of MBE micro injections, as suggested by Stieglbauer and Rončević [98]. We thoroughly described the application of MDE micro injections to our use case in [97].

---

6. "*The modelling language has to be intuitively understandable to their users (e.g. testing engineers) who do not necessarily have experience in software engineering. For*

example, syntactic structures such as `object.method();` *should be avoided since they are not intuitively understandable to non-software engineers.*"

The set of operators and separators within the language syntax was kept to a pragmatic minimum. Otherwise, the syntax is mainly comprised of meaningful keywords. The intuitive understandability of MDML in practice has been informally evaluated by the author during its design phase, as well as formally evaluated by our project partners from the Vienna University of Economics and Business within the DLUX project, which aims to improve the user experience of MDE tools within AVL.

---

7. *"The modelling language has to support test case generation from partial device models. The initial state must be present in the model. Every model trace originating from the initial state should be available for testing."*

   The system under test is related to the model via the **ioco** relation (see Section 2.2.2) which restricts the required conformance of the SUT to only those traces which are present in the model. Any additional traces, which might be implemented in the SUT, are not examined.

---

8. *"The modelling should be facilitated by a plugin for the Eclipse IDE."*

   An Eclipse-based IDE has been developed and is described in Chapter 5. For IDE-specific requirements see Section 5.1.

---

9. *"The semantics of the modelling language should make it easy to convert it into UML or SysML for compatibility with other models."*

   The models can be structured according to the state normal form (see Section 4.2.1) which groups all transitions by their originating states. This is analogous to the UML state machine diagram notation where states are represented by boxes and transitions are represented by outgoing arrows. This similarity facilitates bidirectional conversion to both UML and SysML [49] state machine diagrams.

---

10. *"Up to this point it has not been defined whether the modelling language should provide means to express non-functional requirements, in addition to expressing the required behaviour."*

    The development of a general testing methodology for non-functional requirements has eventually grown out-of-scope for the measurement device use case of the TRUCONF project. However, the MDML language is able to model timed transitions and the test environment imposes limits on the response times of commands sent to the measurement device, both of which can arguably be classified as non-functional requirements.

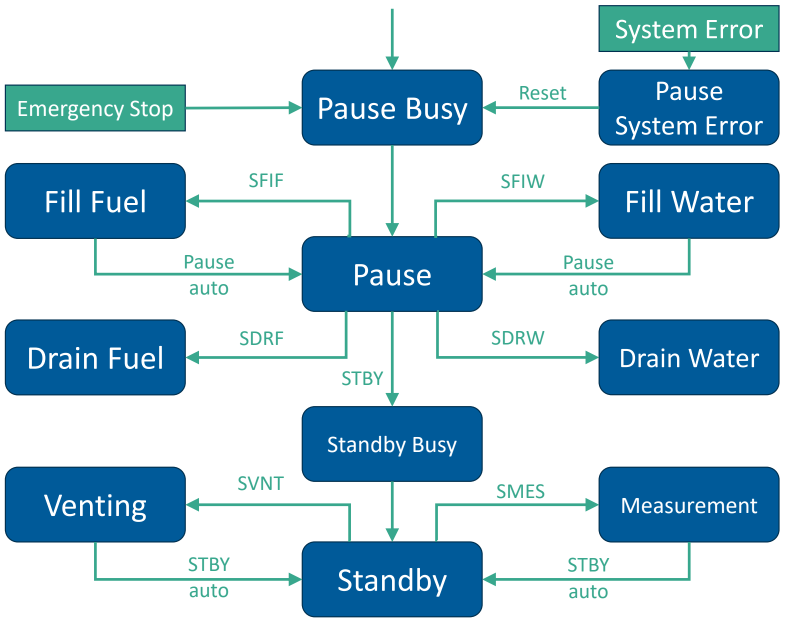


Figure 3.2: State machine diagram of *AVL FuelExact PLU* (AVL740), based on [19, p.132].

3.4 MDML Model Example: AVL740

Having laid out all the necessary background and reasoning for the introduction of MDML, we present a small preliminary model of the *AVL FuelExact PLU* measurement device (also known as AVL740) which measures the fuel mass flow to a combustion engine while simultaneously regulating the fuel temperature. This MDML model shall provide a first impression of the modelling language before each language element is presented in detail in Chapter 4. Later stages of the TRUCONF toolchain will also be explained by means of this MDML model. The model was created from the information present in its official product guide [19], which constitutes a common workflow for a test engineer. The product guide offers a list of AK commands [57] which are used to control the device over Ethernet, as well as an informal state machine diagram (Figure 3.2). The state machine diagram was chosen as the primary information source with the command list as a backup. In this diagram, the Operation and Transition state variables which were separated in the state machine diagram of AVL489 (see Figure 3.1) have been folded into a one-dimensional state machine. The Control state variable is absent from the diagram and will therefore be ignored in the model. It is assumed that the device is in the state *Remote* at the beginning of each test sequence. The representation of system errors and emergency stops has been omitted. The remaining arrow pointing to the state configuration *Pause/Busy* is assumed to originate in all other state configurations except *Standby/Busy*, *FillFuel* and *FillWater* with the latter two having explicit transitions to *Pause/Ready*. All transitions with the trigger “auto” are timed transitions. Since this is a preliminary device model and reliable information about the time-outs of timed transitions tends to be difficult to come by, all transition times in this model are assumptions made by the author. When C# test suites are generated from an MDML model, all state variables and states must be mapped to enumerations and enum symbols of the test automation framework object model. Lacking a strict naming convention, their names cannot be automatically inferred and must therefore be annotated in the MDML model. All states were annotated with their corresponding enum symbols. For more information on the test automation framework and necessary annotations, see Chapter 8, as well as Section 4.4.5 for the concrete annotation syntax. The MDML model, which was based on this information, is presented on Page 27.

The first line identifies the model by device name and version number. Lines 3-12 contain the definition of the state variable *DeviceState*. Each possible state has been annotated by its corresponding enum symbol. The state variable definition ends with the definition of its initial state (*Pause*). Lines 14-17 contain the definition of the state variable *Status* which is structured in the same manner. Line 19 contains the definition of an input channel *UserAction*, containing eight different AK commands which can be received by the device. The rest of the model is structured as a decision tree. State transitions are performed, based on the current values of the state variables *DeviceState* and *Status*, as well as the received *UserAction*.

3.4.1 Model Life Cycle

Generally, fully consistent and complete information about the behaviour of a particular measurement device is hard to obtain from the product guide alone. Information from different sources has to be pieced together and reconciled with the intended behaviour. This data inconsistency continues to be a persistent obstacle in the measurement device testing process. The connection of the modelling environment to a *single source of truth* SSoT can mitigate this problem to a certain degree (see Section 5.3 for an example). These circumstances dictate a life cycle for measurement device models. The model starts as an initial draft which is then continuously compared to a prototypical system under test. All observed cases of non-conformance have to be evaluated and classified as either modelling errors or SUT bugs. As previously stated in Section 3.1, UML-based device models exhibit the same life cycle. Nonetheless, MDML aims to reduce the model editing effort to the point of negligibility.

```
device AVL740 version 4.00 {
  // Definition of Operating states, annotated with corresponding C# enum symbols
  public statevar DeviceState {
    Pause(AVL740.States.Pause),
    Standby(AVL740.States.Standby),
    FillFuel(AVL740.States.Fill_Fuel),
    DrainFuel(AVL740.States.Drain_Fuel),
    FillWater(AVL740.States.Fill_Water),
    DrainWater(AVL740.States.Drain_Water),
    Venting(AVL740.States.Venting),
    Measurement(AVL740.States.Measurement)
  } = Pause;
  // Definition of Transition states, annotated with corresponding C# enum symbols
  public statevar Status {
    Ready(AVL740.TransitionStates.Ready),
    Busy(AVL740.TransitionStates.Busy)
  } = Ready;
  // Definition of AK commands which can be received by the device
  input UserAction {SPAU, STBY, SFIF, SDRF, SFIW, SDRW, SVNT, SMES};

  // Decision tree

  given Status = Ready {
    // Transitions between Operating states
    given DeviceState = Pause {
      when UserAction = STBY then DeviceState -> Standby and Status -> Busy;
      when UserAction = SFIF then DeviceState -> FillFuel;
      when UserAction = SDRF then DeviceState -> DrainFuel;
      when UserAction = SFIW then DeviceState -> FillWater;
      when UserAction = SDRW then DeviceState -> DrainWater;
    }
    given DeviceState = Standby {
      when UserAction = SVNT then DeviceState -> Venting;
      when UserAction = SMES then DeviceState -> Measurement;
    }
    given DeviceState = Venting when UserAction = STBY then DeviceState -> Standby;

    given DeviceState = Measurement when UserAction = STBY then DeviceState -> Standby;

    //Transitions to Pause
    given DeviceState in {FillFuel, DrainFuel} {
      when UserAction = SPAU then DeviceState -> Pause;
    }
    given DeviceState not in {FillFuel, DrainFuel} {
      when UserAction = SPAU then DeviceState -> Pause and Status -> Busy;
    }
  }

  // Return to Ready; Timeouts assumed
  given Status = Busy {
    given DeviceState = Pause when 10 sec elapsed then Status -> Ready;
    given DeviceState = Standby when 15 sec elapsed then Status -> Ready;
  }

  // Timed states; Timeouts assumed
  given DeviceState = Venting when 30 sec elapsed then DeviceState -> Standby
    and Status -> Ready;
  given DeviceState = Measurement when 30 sec elapsed then DeviceState -> Standby
    and Status -> Ready;
  given DeviceState = FillFuel when 30 sec elapsed then DeviceState -> Pause
    and Status -> Ready;
  given DeviceState = FillWater when 30 sec elapsed then DeviceState -> Pause
    and Status -> Ready;
}
```

4 MDML Specification

Some parts of this chapter have already been published in DECOSYS 2016 [28].

This Chapter contains a thorough description of the MDML language with focus on practical usage, as well as a full grammar. The description will initially focus on the most basic language features necessary to produce a simple model. Thereafter, it expands on multidimensionality and advanced language features. In some cases, parallels are drawn to common programming languages (mainly those of the C family) to better illustrate certain concepts.

4.1 Basic Model Structure

4.1.1 Device

Each MDML model encodes the description of a particular measurement device. The model is framed by an all-encompassing block which represents a named device:

```
device AVLDEMO {
  ...
}
```

Currently, the parentheses serve a purely visual purpose: they make it easier to distinguish a full MDML model from a mere MDML snippet. Optionally, the device can be annotated with the firmware version number to distinguish between different model versions:

```
device AVLDEMO version 1.00 {
...
}
```

The content of a device can be roughly separated into a header and a body section although they are not explicitly declared as such. The header section contains all declarations and initializations in the form of state variable and input definitions. The body section is composed of a hierarchy of `given`, `when` and `then` statements which describe the intended behaviour of the device. As such, the body section can be thought of as a decision tree, embedded in an implicit infinite loop. This decision tree determines the next state, based on the current state and the current input.

4.1.2 State Variables

The state variables define the state space of the device. Although all state machines could possibly be described with only a single state variable [104], MDML was specifically designed to handle multi-dimensional state machines.

Each state variable declaration begins with an access specifier:

public: If the state variable is declared public, its changes are considered to be visible to the environment. Translated to an OOAS, a public state variable will propagate its values via an observable action. This is the general case for state variables.

private: The value of the state variable is not visible to an outside observer. This is used in special situations, as described in Section 4.4.3.

The access specifier is followed by the keyword `statevar` and the unique variable name. It is followed by a list of all possible states of this state variable, enclosed in curly brackets. The declaration ends with the assignment of an initial value, which must be one of the states declared beforehand. The declaration is terminated by a semicolon. Consequently, a state variable definition works similar to a combination of an enumeration type definition (as they occur e.g. in languages of the C family) and an enumeration variable initialization. In some cases, the declared states must receive annotations. For more information on this matter, see Sections 4.4.5 and 8.2.

```
public statevar DeviceState {Pause, Standby, Measurement} = Pause;
```

4.1.3 Inputs

An input event is a signal which can be received and processed by the device. As such, it directly translates to a controllable action in OOAS. Input events are grouped into input channels which help to keep the models more orderly. The definition of an input channel roughly resembles the declaration segment of a state variable definition. It starts with the keyword `input`, followed by the channel name and a list of possible input symbols, enclosed in curly brackets. The declaration is terminated by a semicolon.

```
input UserAction {STBY, SMES, SPAU};
```

Ideally, the communication over input events is synchronous, with input events being processed at the same time at which they are issued. However, this hinges on the implementation of the underlying communication drivers and device firmware. As mentioned before in the discussion about the **ioco** relation (Section 2.2.2), measurement devices are input-output transition systems and accept any input in any state. Therefore, transitions associated with undefined inputs must be ignored during test case generation.

4.1.4 Given Statements

A `given` statement encodes a comparison expression (e.g. equality) on the value of a specific state variable:

```
given DeviceState = Pause ...
```

It defines an enabling condition for a specific portion of the decision tree. As such, it is functionally equivalent to the condition in an `if` statement, as it is known from various programming languages. As opposed to many popular programming languages, MDML uses a single equals sign rather than a double equals sign as a relational operator since value assignments are handled by the transition operator (`->`, see Section 4.1.6). A `given` statement can be either directly followed by a `when` statement (see Section 4.1.5) or by a block which allows multiple `when` statements and/or cascading with other `given` statements:

```
given DeviceState = Pause when ...
given DeviceState = Standby {
  when ...
  when ...
  given ...
}
```

Additionally, `given` statements can be formulated as inequality- and set-based conditions on state variables. An exhaustive list of different types of expressions is given below:

```
1 given DeviceState != Pause ...
2 given DeviceState in {Pause, Standby} ...
3 given DeviceState not in {Pause, Standby} ...
```

If a new state is added to the state variable *DeviceState* in the above example later on, it will implicitly enable the `given` statements in Line 1 and 3 which can potentially lead to unintended behaviour. Nevertheless, such negative expressions can still be useful since they are more intuitively understandable than long inclusive lists.

4.1.5 When Statements

A typical `when` statement is composed of the keyword `when`, which is followed by an input channel name, an equals sign and a specific input event from the range of the channel.

```
when UserAction = SPAU ...
```

This statement makes the device react to an input event or other type of specified event (see Sections 4.3, 4.4.1). In practice, many input events are only available in a specific state or *state configuration*[1]. This is encoded by the position of the `when` statement in the hierarchy of `given` statements which forms the decision tree. If a `when` statement is not enclosed by any `given` statements it is enabled in all possible state configurations.

[1] *The overall state of a state machine, consisting of the assigned states of all its state variables.* Compare with the UML specification [48, p.306].

Additionally to the equality expression described above, `when` statements on input channels can take various other shapes. An exhaustive list of them is given below:

```
1 when UserAction != SPAU ...
2 when UserAction in {SPAU, STBY} ...
3 when UserAction not in {SPAU, STBY} ...
4 when UserAction = any ...
```

An explanation of the previous code snippet, enumerated by line number:

1. This statement is triggered by all input events in the channel *UserAction* other than *SPAU*.
2. This statement is triggered by the input events *SPAU* and *STBY*. The set of input events cannot be empty.
3. This statement is triggered by all input events in channel *UserAction* other than *SPAU* and *STBY*. The set of input events cannot be empty.
4. This statement is triggered by all input events in channel *UserAction*. If the device contains only one channel, this statement encompasses all input events.

Separating the input events in different input channels can be useful when combined with these features. Nevertheless, if a new input event gets added to the channel *UserAction* in the above example later on, it will implicitly trigger statements in Line 1, 3 and 4, which could potentially lead to unintended behaviour.

Note that a measurement devices must always behave deterministically. Therefore, it is forbidden to have a specific input event handled multiple times in a specific state configuration. This design choice was made to prevent the need for potentially confusing precedence rules[2], but it could also result in slightly longer and more redundant models. If an input event would trigger two or more different `when` statements in a specific state configuration, a *Duplicate Path Error* is reported.

4.1.6 Then Statements

A typical `then` statement describes the reaction of the device to an event. Most commonly, this reaction is a state transition. A state transition is encoded by the keyword `then`, followed by the name of the state variable, an arrow-shaped transition operator and the name of the target state. A semicolon denotes that the current branch of the decision tree is terminated and all subsequent statements are part of a different branch:

```
...then DeviceState -> Pause;
```

If the source state is identical to the target state, the state transition has no effect. Such a transition is only meaningful if the enclosing `given` statements are enabled for a broader state range which makes the identity implicit. Diagonal state transitions with several state variables changing at once are described in Section 4.2.

[2]Contrast the UML state machine diagram semantics [48, pp.315-316], which incorporates precedence relations for conflicting transitions.

4.1.7 Code Comments

Before more complex language elements are described, the commenting syntax of MDML needs to be addressed. MDML adopts its commenting syntax from C. Thus, it offers two different comment styles, illustrated in the example below. Single-line comments begin with the character string // and end at the next line break. In-line or block comments begin with the character string /* and end with */. This style allows the comment to span multiple lines and/or to be placed within code lines.

```
given DeviceState = /*in-line comment*/ Standby { // single-line comment
  ...
```

4.1.8 A Functioning Model

Utilizing the previously described language elements in their simplest forms, it is now possible to build a functioning model of a measurement device, as illustrated by the fictitious example below. An equivalent UML state machine diagram of the model is given in Figure 4.1.

```
device AVLDEMO {
  public statevar DeviceState {Pause, Standby, Measurement} = Pause;
  input UserAction {STBY, SMES, SPAU};

  given DeviceState = Pause {
    when UserAction = STBY then DeviceState -> Standby;
  }
  given DeviceState = Standby {
    when UserAction = SPAU then DeviceState -> Pause;
    when UserAction = SMES then DeviceState -> Measurement;
  }
  given DeviceState = Measurement {
    when UserAction = SPAU then DeviceState -> Pause;
  }
}
```

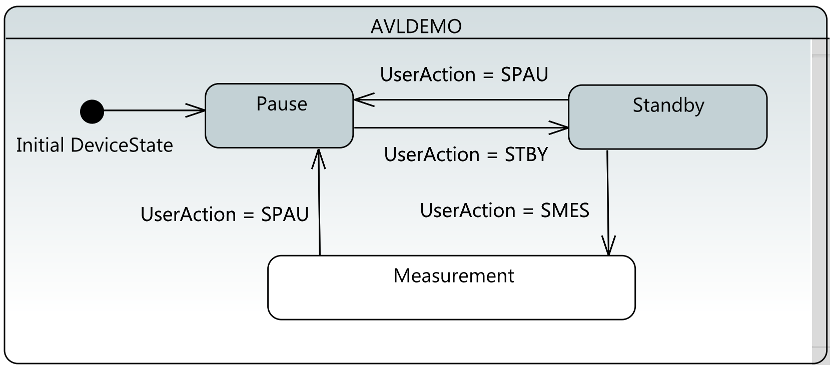


Figure 4.1: A UML state machine diagram of the fictional *AVLDEMO* model.

4.2 Multidimensional Models

Most measurement devices within the AVL portfolio are multidimensional, which means that they require more than one state variable. As stated above, it would be possible to describe every state model with just one state variable. This is rarely done in practice since this measure tends to produce a large number of states. The most common state variables in AVL measurement devices, as well as their general behaviours are listed below:

DeviceState (Operating State): The "main" state variable which denotes the operation which is currently being performed by the device. Common states include *Pause*, *Standby* and *Measurement*.

Status (Transition State): If a device needs to perform some internal mechanical action (like e.g. opening and closing valves during the transition between Operating States), it will not respond to any commands until this action is completed. This behaviour is reflected in the *Status* state variable which contains the states *Ready* and *Busy*. In some cases, a change of *DeviceState* will also result a change of *Status* to *Busy*. While *Status* is *Busy*, the device cannot receive any commands. *Status* will later revert back to *Ready* via a timed behaviour (see Section 4.3).

UserLevel (Control State): This state variable denotes whether the device receives its commands remotely or by direct manual input. Like *Status*, it usually is a binary state variable which contains the states *Manual* and *Remote* or some variation thereof. While in the state *Manual*, the device can usually only receive the command to switch to *Remote*.

If a `then` statement causes the change of two or more state variables, the keyword `and` is used. The order of different state transitions concatenated by `and` only has an effect on the model's behaviour if secondary actions (see Section 4.4.1) are involved.

```
...then DeviceState -> Purging and Status -> Busy;
```

Note that the repeated assignment of a particular state variable within one `then` statement is forbidden and results in a *Contradictory State Transition Error*.

With the above mentioned language elements, it is now possible to build a simple multidimensional model, as exemplified below. A UML state machine diagram of the example model is given in Figure 4.2.

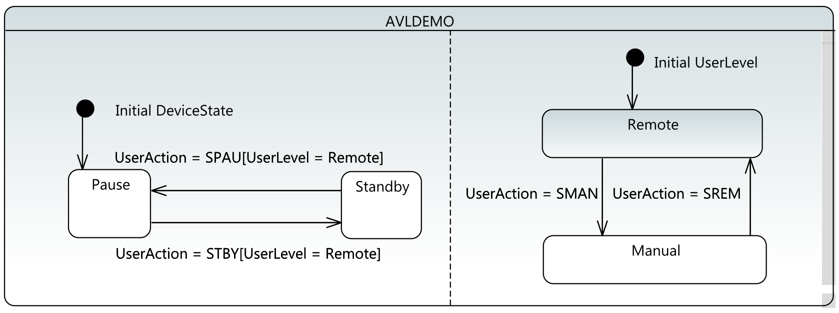


Figure 4.2: A UML state machine diagram of the multidimensional model example.

```
device AVLDEMO {
  public statevar DeviceState {Pause, Standby} = Pause;
  public statevar UserLevel {Manual, Remote} = Remote;
  input UserAction {SPAU, SMES, SMAN, SREM};

  given UserLevel = Remote {
    when UserAction = SMAN then UserLevel -> Manual;

    given DeviceState = Pause {
      when UserAction = STBY then DeviceState -> Standby;
    }

    given DeviceState = Standby {
      when UserAction = SPAU then DeviceState -> Pause;
    }
  }

  given UserLevel = Manual {
    when UserAction = SREM then UserLevel -> Remote;
  }
}
```

No measurement device within the AVL portfolio contains final states (states without any outgoing transitions). The following model provides several examples of such unwanted final states:

```
device AVLDEMO {
  public statevar DeviceState {Pause, Standby, Measurement} = Pause;
  public statevar UserLevel {Manual, Remote} = Remote;
  input UserAction {SPAU, STBY, SMES};

  given UserLevel = Remote {
    given DeviceState = Pause {
      when UserAction = STBY then DeviceState -> Standby;
    }
    given DeviceState = Standby {
      when UserAction = SPAU then DeviceState -> Pause;
      when UserAction = SMES then DeviceState -> Measurement;
    }
  }
}
```

For this model, the IDE would yield an *Unhandled State Configuration Warning* of the following form:

```
This model contains unhandled state configurations:
  Measurement/Remote
  ---/Manual
```

This warning message alerts the user to two unhandled state configurations:

1. The state configuration *Measurement/Remote* is unhandled. This is a problem, since this state can be reached from the initial state.
2. None of the state configurations with *UserLevel = Manual* are handled. However, this is a false positive, since there is no way to reach these configurations.

Nevertheless, due to the properties of the **ioco** relation (see Section 2.2.2), models with unhandled state configurations can still be used for the generation of meaningful test cases. Each test sequence reaching an unhandled state configuration will simply terminate.

4.2.1 Common Decision Tree Structures

Our experience has shown that certain decision tree structure patterns tend to occur during practical application. These patterns are by no means required model features and they do not need to be implemented in the exact way described below. During the rest of this section, it is assumed that the common state variables (*DeviceState*, *Status*, *UserLevel*) are present and implemented as described in Section 4.2. The code example below reflects the most common structure of a decision tree:

```
given Status = Ready {
  given UserLevel = Remote {
    given DeviceState = Pause ...
    given DeviceState = Standby ...
    ... // given statements regarding DeviceState go here
  }
  given UserLevel = Manual {
    ... // handling of Manual state goes here
  }
}
given Status = Busy {
  ... // handling of Busy state goes here
}
```

Note that the outermost level distinguishes by *Status*. If the device is *Busy*, all inputs tend to be disabled an the *Busy* state is left via a timed behaviour (see Section 4.3). The second layer distinguishes by *UserLevel*. The state *Manual* tends to only permit one particular input which causes the device to switch to *Remote*. Most of the model code is usually contained within the `given` block for the state configuration *Ready/Remote* where the distinction by *DeviceState* is located. This distinction tends to assume one of the following forms:

State Normal Form

```
given DeviceState = Pause {
  when UserAction = STBY then DeviceState -> Standby;
  when UserAction = SMES then DeviceState -> Measurement;
}
given DeviceState = Standby {
  when UserAction = SPAU then DeviceState -> Pause;
  when UserAction = SMES then DeviceState -> Measurement;
}
given DeviceState = Measurement {
  when UserAction = SPAU then DeviceState -> Pause;
  when UserAction = STBY then DeviceState -> Standby;
}
```

In the state normal form, all transitions are grouped by their originating *DeviceState*. This normal form tends to be intuitively understandable and can also be easily related to corresponding UML state machine diagrams where states and transitions are represented by boxes and outgoing arrows, respectively. Nevertheless, it tends to produce longer models since transitions with the same trigger and target state still require full `when` and `then` statements for each origin state.

Input Normal Form

```
given DeviceState in {Standby, Measurement} {
  when UserAction = SPAU then DeviceState -> Pause;
}
given DeviceState in {Pause, Measurement} {
  when UserAction = STBY then DeviceState -> Standby;
}
given DeviceState in {Pause, Standby} {
  when UserAction = SMES then DeviceState -> Measurement;
}
```

In the input normal form, all transitions are grouped by their input event and target state. This implicitly assumes that input events and target states always correlate. The possible origin states for each transition are defined as list expressions in the `given` statements. This normal form makes an intuitive understanding of the model more difficult but it tends to be significantly shorter and to contain less redundancy. It also allows for fast modelling if the user obtains the underlying information from AVL's official measurement device manuals. These documents contain a list of possible commands which is structured analogously to this normal form.

4.2.2 Unordered Decision Trees

It is generally advisable to create models with ordered decision trees. This means, for example, that the outermost level of the tree exclusively distinguishes by the state variable *Status*, the next level distinguishes by *UserLevel* and the innermost level exclusively distinguishes by *DeviceState*. Usually, this practice results in more intuitively understandable models. However, in some cases it is reasonable to create models which depart from this structure. From a purely functional point of view, this is never strictly necessary, but it should be done to avoid redundancy within the model. For example, a hypothetical measurement device contains a timed behaviour (see Section 4.3) in the state *Measurement* which is activated regardless of the *UserLevel* being *Remote* or not. This situation would warrant the following kind of decision tree:

```
given DeviceState = Measurement {
  ... // timed behaviour goes here
}

given UserLevel = Remote {
  given DeviceState = Pause ...
  given DeviceState = Standby ...
  given DeviceState = Measurement ... // everything else goes here
}
```

The timed behaviour would be modelled in Line 2 which is enabled regardless of the value of *UserLevel*. In this case, the device can act autonomously in a state configuration where it cannot receive any commands from the user. All commands which are enabled in the state configuration *Remote/Measurement* are modelled in Line 8. If the decision tree was well ordered, the timed behaviour would have to be implemented twice, for the configurations *Manual/Measurement* and for *Remote/Measurement*.

However, this freedom in decision tree structure also comes with several possibilities for pitfalls. From a syntactical point of view, the following construct would be allowed:

```
given DeviceState = Pause {
  given DeviceState = Standby {
    ...
  }
}
```

Since both `given` statements are mutually exclusive, all statements within the inner bracket are always disabled. If such constructs exist, an *Infeasible Path Error* is reported. However, the following construct would be semantically correct and, under certain circumstances, advisable:

```
given DeviceState in {Pause, Standby} {
  when ...
  when ...
  given DeviceState = Standby {
    ...
  }
}
```

4.3 Timed Behaviours

Nearly every measurement device within the AVL portfolio contains some sort of timed behaviour. MDML is able to handle state transitions which occur after a specified time. For example, the following `when` statement triggers after 3 seconds:

```
when 3 sec elapsed then ...
```

Currently supported units of time are `min`, `sec` and `ms`. The timer starts running as soon as a state configuration is reached which enables the timed behaviour. If the device changes to a state configuration which disables the timed behaviour before it triggers, the timer is aborted and resets. For instance, the timed behaviour in the following example will trigger 3 seconds after *Flip1* and *Flip2* have been received at least once. However, *Flop1* can abort the timer by resetting *Switch1*.

```
public statevar Switch1 {Off, On} = Off;
public statevar Switch2 {Off, On} = Off;
input UserAction {Flip1, Flip2, Flop1};

when UserAction = Flip1 then Switch1 -> On;
when UserAction = Flip2 then Switch2 -> On;
when UserAction = Flop1 then Switch1 -> Off;

given Switch1 = On {
  given Switch2 = On {
    when 3 sec elapsed then Switch1 -> Off and Switch2 -> Off;
  }
}
```

It is important to note that a generated test will also yield a pass verdict if the system under test exhibits the timed behaviour *before* the timer expires. There are currently no mechanisms in place which assert that no transition has occurred before the specified time.

Currently, there is no measurement device known to the author which would require two or more concurrent timed behaviours. To keep the complexity of MDML models to a minimum, this limitation is strictly enforced in the language. If two or more timed behaviours are enabled in a specific state configuration, a *Multiple Time Triggers Error* is reported.

4.4 Advanced Features

4.4.1 Secondary Actions

In Section 4.2 it has been stated that state transitions which affect multiple state variables are written as:

```
... then DeviceState -> Purge and Status -> Busy;
```

However, if a transition of one state variable into a specific state always results in a transition of another state variable into a specific state, this can alternatively be written as a secondary action:

```
when DeviceState -> Purge then Status -> Busy;
```

These secondary actions can also include `and` statements and result in further state transitions. Any direct or indirect multiple assignment of a state variable will result in a *Multiple Assignment Error*. Cascades of secondary actions are evaluated *breadth-first*. For example, consider a number of state variables $V_a, V_b, \ldots, V_l$ which are interlinked by a hierarchy of secondary actions. The transition of V_a causes a transition of V_b and V_c, each of which cause subsequent secondary actions on their own:

```
when Va -> Sa then Vb -> Sb and Vc -> Sc;
when Vb -> Sb then Vd -> Sd and Ve -> Se;
when Ve -> Se then Vh -> Sh;
when Vh -> Sh then Vk -> Sk;
when Vc -> Sc then Vf -> Sf and Vg -> Sg;
when Vg -> Sg then Vi -> Si and Vj -> Sj;
When Vi -> Si then Vl -> Sl;
```

Independently of the order of the individual `when` statements, the ensuing cascade of secondary actions, depicted in Figure 4.3, would evaluate exactly in alphabetical order, due to breadth-first traversal of the cascade. Knowledge of the evaluation order of secondary actions is important if one considers placing them within `given` statements which can be affected by previous transitions.

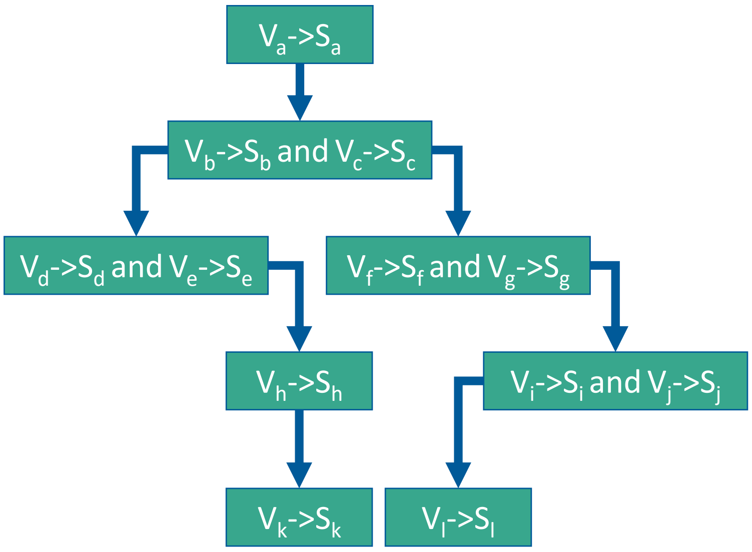

V_a->S_a
V_b->S_b and V_c->S_c
V_d->S_d and V_e->S_e
V_f->S_f and V_g->S_g
V_h->S_h
V_i->S_i and V_j->S_j
V_k->S_k
V_l->S_l

Figure 4.3: An example cascade of secondary actions.

4.4.2 Last-Transitions

Certain states behave in such a way that, after the operation associated with the state has finished, the device returns to its previous state. In order to model this kind of behaviour, MDML offers a very simple history in the form of the `last` keyword. A last-transition will cause the specified state variable to return to its previous state. For instance, if the state *Measurement* is entered from *Pause* in the following example, the state variable will return to *Pause*. If the state is entered from *Standby*, it will return to *Standby*. As a result, the word “last” is disallowed as a state name.

```
given DeviceState in {Pause, Standby} {
  when UserAction = SUMS then DeviceState -> Measurement;
}
given DeviceState = Measurement {
  when UserAction = SUME then DeviceState -> last;
}
```

If a last-transition is entered repeatedly, the state variable will alternate between two states. If a last-transition is entered from the initial state, it has no effect.

In order to keep the complexity of MDML models to a minimum, the state in which a last-transition can occur should be as clearly defined as possible. If a `then` statement containing a last-transition is enabled in more than one state of the specified state variable, an *Unclear Last-Transition Warning* is reported. This is no impediment to the functionality of the model but it greatly diminishes its understandability.

4.4.3 Private State Variables

Sometimes, looking at the observable state of a device is not enough to determine, what it is going to do next. Last-transitions, for example, model a specific variety of this behaviour and should be used if applicable. But as a general solution to this problem, MDML allows the definition of private state variables. In the following example, the inputs *SMES* and *SUMS* both cause the *DeviceState* to change to *Measurement*. But only when entered via *SMES*, the *DeviceState* automatically reverts back to *Standby*. This behaviour is modelled by introducing a private state variable. This, however, is a last resort because adding an additional state variable can lead to the emergence of many meaningless state configurations.

```
public statevar DeviceState {Standby, Measurement} = Standby;
private statevar TM {On, Off} = Off;
input UserAction {STBY, SMES, SUMS};

given DeviceState = Standby {
  when UserAction = SMES then DeviceState -> Measurement and TM -> On;
  when UserAction = SUMS then DeviceState -> Measurement and TM -> Off;
}

given DeviceState = Measurement {
  when UserAction = STBY then DeviceState -> Standby;
  given TM = On when 20 sec elapsed then DeviceState -> Standby;
}
```

4.4.4 Self-Transitions

It is common for measurement devices to accept inputs which do not alter the current state. This behaviour can be expressed through the following construct:

```
... then DoNothing;
```

In our initial implementation, the system stayed quiescent when enacting the `DoNothing` command (see δ, Section 2.2.2). There was no way to test whether the state configuration actually remained unchanged. This changed when we made a last-minute alteration to our algorithm which made self-transitions verifiable (see Section 9.4).

4.4.5 Annotations

In all currently known cases, input events and state transitions are mapped to method calls of measurement device PageObjects in the test automation framework. These method calls require parameters which can be annotated in the state- and input channel definition, respectively.

```
public statevar DeviceState {Standby(1), Purging(12)} = Standby;
input UserAction {Standby, Purge(20,false)};
```

Here, the state *Purging* would be replaced by the state code *12* while the method *Purge* would be called with an integer argument and a boolean argument. In the context of MDML, this feature is merely an annotation and therefore completely optional. However, it might be required by the testing environment, depending on the used test case transformation scheme. The allowed types of parameters are integers, strings within double quotation marks, IDs (a letter or underscore, followed by any number of letters, numbers or underscores) and hierarchical IDs (An ID, followed by an arbitrary number of IDs, separated by periods). For more information about annotations, we refer to the description of the various test case transformation schemes in Chapter 8.

4.5 MDML Grammar

To conclude our description of MDML, we present a full grammar of the language in extended Backus-Naur form (EBNF, [106]) in Table 4.1. Curly brackets signify the 0-to-∞-fold occurrence of their contents while square brackets signify 0-to-1-fold occurrence. Terminal symbols are enclosed by single quotation marks. The terminal rules **ID**, **INT** and **STRING** are predefined in the Xtext framework. According to the Chomsky hierarchy, the grammar of MDML can be classified as context-free since the left side of each production rule consists of a single non-terminal while the right side consists of a non-empty string of terminals and non-terminals [29].

Each device is composed of a header and a body section. Both header and body are not explicitly syntactically marked but are distinguished by their different types of content. The header resolves into a list of state variable definitions and input channel definitions which can alternate in arbitrary order. To establish a connection between the grammar of MDML and the formal considerations in the upcoming chapters, we define a series of symbols which will be used later on. For example, the sets of all state variables and input channels defined in the header of a device D will be referred to as $\mathit{StateVariables}_D$ and $InputChannels_D$, respectively. As stated in Section 4.1.2, the definition of a state variable $v \in \mathit{StateVariables}_D$ includes a set of possible states States_v, as well as the declaration of its initial state $InitialState_v$. Analogous to state variables, the definition of an input channel $c \in InputChannels_D$ includes a list of possible input events $Events_c$, as stated in Section 4.1.3.

The body of a device contains a set of behaviours, henceforth referred to as $Body_D$. The recursive definition of a behaviour is the basis for MDML's decision tree structure. It can resolve to `given` or `when` blocks, the former of which either contains a behaviour or a behaviour block which, in turn, contains a list of behaviours. A `when` statement defines an event which triggers appropriate reactions, defined in the form of a `then` statement. Such events can either be input-based (see Section 4.1.5), secondary actions of state transitions (see Section 4.4.1) or timed behaviours (see Section 4.3). The set of all timed behaviours which occur within a device is called $TimeTriggers_D$ and will also be of importance later on.

Device	:=	‘device’ **ID** [‘version’ VNr] ‘{’ Header Body ‘}’
VNr	:=	(**VP** | **INT**) {‘.’ **INT**}
VP	:=	(‘A’...‘Z’) (‘0’...‘9’) {‘0’...‘9’}
Header	:=	{StateVariable | InputChannel}
StateVariable	:=	(‘public’ | ‘private’) ‘statevar’ **ID** ‘{’ State {‘,’ State} ‘}’ ‘=’ **ID** ‘;’
State	:=	**ID** [‘(’ Parameter ‘)’]
Parameter	:=	([‘-’] **INT**) | **STRING** | (**ID** {‘.’ **ID**})
InputChannel	:=	‘input’ **ID** ‘{’ InputEvent {‘,’ InputEvent} ‘}’ ‘;’
InputEvent	:=	**ID** [‘(’ Parameter {‘,’ Parameter} ‘)’]
Body	:=	{Behaviour}
Behaviour	:=	GivenBlock | WhenBlock
GivenBlock	:=	GivenStatement (BehaviourBlock | Behaviour)
GivenStatement	:=	‘given’ (StateEqualsExp | StateListExp)
StateEqualsExp	:=	**ID** (‘=’ | ‘!=’) **ID**
StateListExp	:=	**ID** [‘not’] ‘in’ ‘{’ **ID** {‘,’ **ID**} ‘}’
BehaviourBlock	:=	‘{’ Behaviour {Behaviour} ‘}’
WhenBlock	:=	WhenStatement ThenStatement
WhenStatement	:=	‘when’ (StateTransition | InputEqualsAct | InputListAct | TimeTrigger)
StateTransition	:=	**ID** ‘->’ **ID**
InputEqualsAct	:=	**ID** (‘=’ | ‘!=’) (**ID** | ‘any’)
InputListAct	:=	**ID** [‘not’] ‘in’ ‘{’ **ID** {‘,’ **ID**} ‘}’
TimeTrigger	:=	**INT** UnitOfTime ‘elapsed’
UnitOfTime	:=	‘min’ | ‘sec’ | ‘ms’
ThenStatement	:=	‘then’ (‘DoNothing’ | Reaction {‘and’ Reaction}) ‘;’
Reaction	:=	StateTransition
ID	:=	[‘^’] **Letter** {**Letter** | **Digit**}
Letter	:=	‘A’...‘Z’ | ‘a’...‘z’ | ‘_’
Digit	:=	‘0’...‘9’
INT	:=	**Digit** {**Digit**}
STRING	:=	‘"’ (‘\\’ | !(‘\\’ | ‘"’)) {‘\\’ | !(‘\\’ | ‘"’)} ‘"’ | ‘'’ (‘\\’ | !(‘\\’ | ‘'’)) {‘\\’ | !(‘\\’ | ‘'’)} ‘'’

Table 4.1: EBNF grammar of MDML.

5 An Eclipse-Based IDE for MDML

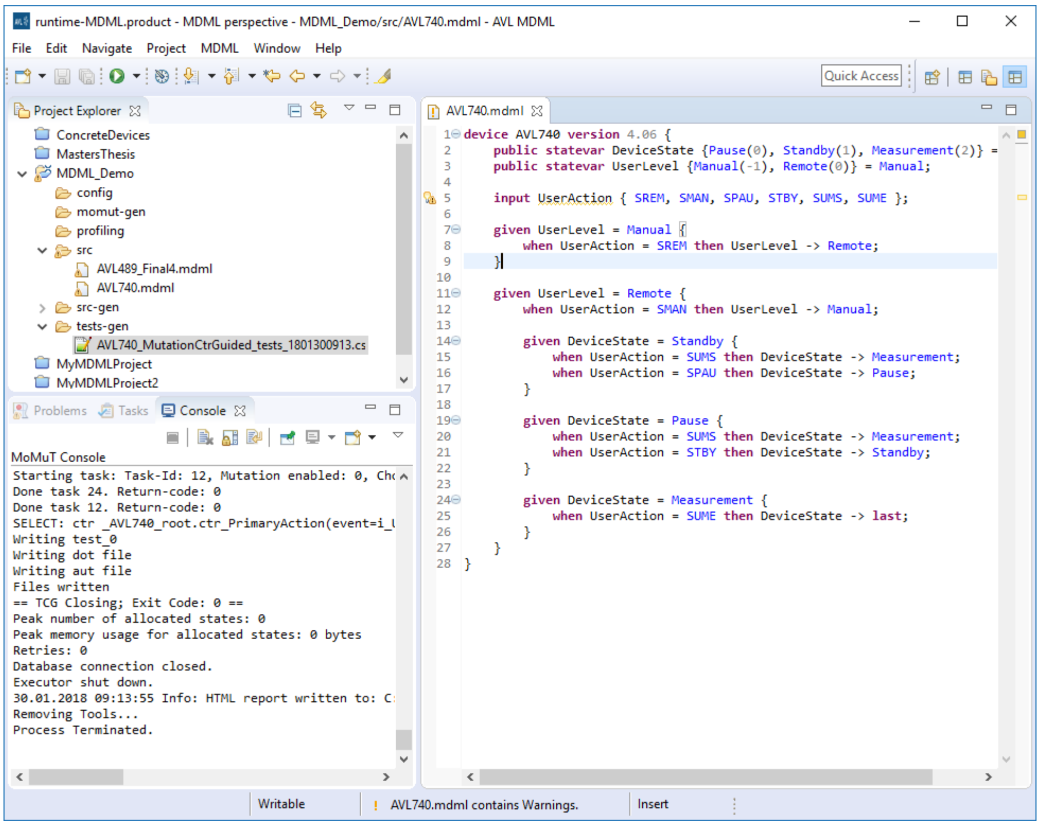
Some parts of this chapter have already been published in DECOSYS 2016 [28], as well as in MODELSWARD 2018 [97].

As domain-specific languages need dedicated tooling support to be used to their full potential [44], we developed MDML in parallel with a dedicated modelling tool. While MDML models could be theoretically developed with an off-the-shelf text editor alone, the benefits of a dedicated tool justify the implementation effort [47]. According to Abrahão et al. [1], a model editor must be tailored to the application domain in a number of different ways: it must offer guidance for workflows which are specific to the application domain, thereby reducing the training effort [97]. It must be designed in a minimalistic way so that it only offers features which are relevant to the application domain. Ideally it would also offer some sort of domain-specific graphical representation of the models, even if the modelling language is of a textual nature. These criteria are valid for all modelling tools but are especially important when designing a domain-specific tool from scratch.

With these basic guidelines in mind, we developed an MDML editor based on the well-known Eclipse IDE[1] which is easily extensible through a flexible plugin API. During the later stages of tool development, we received support from the Vienna University of Economics and Business as part of the DLUX project. This support was mainly given in the form of implementation assistance and the planning and execution of user interviews which provided additional development guidance.

Our tool provides a dedicated MDML modelling environment which integrates relevant data sources from within AVL and encapsulates the test case generation toolchain (see Section 1.5). This chapter mainly focuses on the editor and the data source integration. Figure 5.1 gives a representative impression of the MDML IDE.

[1]`https://www.eclipse.org/`



```
device AVL740 version 4.06 {
    public statevar DeviceState {Pause(0), Standby(1), Measurement(2)} =
    public statevar UserLevel {Manual(-1), Remote(0)} = Manual;

    input UserAction { SREM, SMAN, SPAU, STBY, SUMS, SUME };

    given UserLevel = Manual {
        when UserAction = SREM then UserLevel -> Remote;
    }

    given UserLevel = Remote {
        when UserAction = SMAN then UserLevel -> Manual;

        given DeviceState = Standby {
            when UserAction = SUMS then DeviceState -> Measurement;
            when UserAction = SPAU then DeviceState -> Pause;
        }

        given DeviceState = Pause {
            when UserAction = SUMS then DeviceState -> Measurement;
            when UserAction = STBY then DeviceState -> Standby;
        }

        given DeviceState = Measurement {
            when UserAction = SUME then DeviceState -> last;
        }
    }
}
```

```
Starting task: Task-Id: 12, Mutation enabled: 0, Ch
Done task 24. Return-code: 0
Done task 12. Return-code: 0
SELECT: ctr _AVL740_root.ctr_PrimaryAction(event=i_
Writing test_0
Writing dot file
Writing aut file
Files written
== TCG Closing; Exit Code: 0 ==
Peak number of allocated states: 0
Peak memory usage for allocated states: 0 bytes
Retries: 0
Database connection closed.
Executor shut down.
30.01.2018 09:13:55 Info: HTML report written to: C
Removing Tools...
Process Terminated.
```

Figure 5.1: A screenshot of the MDML IDE.

5.1 Requirements for an Eclipse-Based IDE

Analogous to the DSL-based testing formalism as a whole, a set of initial requirements was defined for the MDML IDE at the beginning of the TRUCONF project. As in Section 3.3, we quote the initial requirements and comment on them, based on the results achieved at project conclusion.

1. *"The Eclipse plugin has to provide a modelling environment for both the textual and graphical modelling language."*

   As previously stated in Section 3.3 - req. 2, the graphical representation of MDML models has grown out of scope for the TRUCONF project. All further requirements concerning the graphical representation are omitted.

---

2. *"The Eclipse plugin should provide ways to quickly generate code structures in the textual representation like, for example, wizards, code completion, quickfixes and module templates which can then be filled by the users."*

   The Eclipse-based MDML IDE was created using the Xtext framework (see Section 5.2) which provides the developer with the opportunity to easily implement tooltips, quickfixes, content assist, etc. These functionalities were used to provide the users with as much information and guidance as possible. Moreover, a series of wizards was implemented to import information from the *device knowledge base* (DKB, see Section 5.3) as well as the *model zoo* (see Section 5.4).

---

3. *"The Eclipse plugin should provide syntactic and semantic coding tips to the user."*

   Syntactic content assist is largely provided out-of-the-box by the Xtext framework. Furthermore, descriptive tool tips of syntactic elements have been implemented. To the extent of its availability within the DKB, semantic information has been integrated into these tooltips.

---

4. *"The Eclipse plugin has to provide information from AVLs device database [sic!]. (The nature of this information is currently subject to debate, but state and signal names will most likely be provided automatically)."*

   The MDML IDE supports the import of outward device interfaces - specifically, input events and states - from the DKB. While the knowledge base would also contain behavioural information like state transitions, it is intentionally left out of the test model to uphold the four-eyes principle. A redundant behavioural specification prevents the masking of faults within the original specification [84].

5.2 Implementation Using the Xtext Framework

Xtext[2] [108] is an open-source framework for the creation of domain-specific language editor plugins for the Eclipse IDE. It is mainly being developed by the *itemis AG*[3] in cooperation with the *Eclipse Foundation*. Our decision to use Xtext for the development of an MDML editor is strongly linked to our decision to create a textual domain-specific language. Xtext was suggested by a member of the TRUCONF steering board who already had positive experiences with the framework as well as good relations to the Eclipse community. As it turned out, Xtext is well-suited for the rapid prototyping of a domain-specific editor [97]. It proved to be sufficiently robust during the early stages of language development when we were faced with frequent and substantial changes of the grammar definition. An initial formal implementation of the MDML grammar could be completed within a single working day. Afterwards, this initial grammar gradually evolved while we generated tool prototypes on a regular basis. These tool prototypes gave an early impression of what the finished product would eventually look like. As the grammar definition settled into a largely stable state, the implementation of usability-related features gained momentum. Once a valid grammar file has been created, Xtext uses it as a basis for the automatic creation of a plugin for the Eclipse IDE. This auto-generated plugin comprises a multitude of features with different degrees of out-of-the-box maturity and openness to customization. The most relevant of these features are described in the rest of this section.

[2]`https://www.eclipse.org/Xtext/`

[3]`http://www.itemis.com`

5.2.1 Domain-Specific Editor

Most prominently, Xtext creates a dedicated editor view for MDML which interfaces with *Eclipse modelling framework*[4] (EMF). An auto-generated parser and serializer convert between the textual representation of the MDML model and an Ecore model which is kept in memory. Basic syntax highlighting is supported out-of-the-box but more intricate highlighting rules can be defined by implementing the corresponding method stub. The same is true for auto-formatting. By default, the serializer creates a textual representation as a monolithic block of text. A dedicated method stub allows for the definition of whitespacing rules for the serializer and the auto-formatter.

5.2.2 Code Generator

In model-driven development, models are generally used to easily create artefacts which reside on a lower abstraction level and are considered too complex to be handled directly. For this purpose, Xtext offers a method stub for code generation, based on the contents of the current model. Per default, the code generator takes the Ecore model as input and writes to an output file within the currently opened project in the Eclipse workspace. This segment of the IDE contains the transformation from MDML to object-oriented action systems as it is described in Chapter 6.

5.2.3 Content Assist

Content assist[5] - also known as *intelligent code completion* or, in Microsoft products, *IntelliSense* - is the functionality of providing the user with a list of context-aware suggestions for model elements and keywords during the modelling process, based on syntax, predefined scoping rules and specialized rules for certain language elements [108, p.200]. In most cases, Xtext provides syntactic content assist out-of-the-box. However, when referencing a language element in a place other than its definition (e.g. referencing a state from within a `given` statement) the scope of this reference has to be defined by the DSL engineer. Afterwards, the appropriate content assist options will be given to the user. The MDML IDE provides a few cases of semantic content assist which are described in Section 5.3.

5.2.4 Model Validation

The structural freedom of MDML is bought with a large potential for modelling mistakes and ad-hoc validation of MDML models can be a difficult and tedious task. Thankfully, Xtext provides a validator class which can be extended with individual validation methods for different fault modes, e.g. having multiple state variables of the same name. Depending on the severity of the violation, the testing methods can emit a warning or an error,

[4] `http://www.eclipse.org/modeling/emf/`
[5] `https://www.eclipse.org/pdt/help/html/working_with_code_assist.htm`

the latter of which precludes code generation. The corresponding erroneous parts of the textual model are underlined in yellow or red, respectively. Yet, MDML demands more sophisticated validation operations, e.g. ensuring that an input event is handled only once for a given state configuration. For this purpose, a Prolog-based model validator was created which allows for the declarative specification of validation rules. The Prolog validator was implemented by the author as a project work for his master's studies and was thus kept separate from the work which is presented in this thesis. It is able to diagnose the following errors and warnings, relating to different sections of Chapter 4:

Duplicate Path Error: Two or more different reactions are defined for a certain input event in a certain state combination (see Section 4.1.5).

Infeasible Path Error: The decision tree contains unreachable branches due to contradictory transitions. This is not to be confused with a full state reachability analysis (see Section 4.2.2).

Multiple Time Triggers Error: Multiple timed behaviours are enabled in a certain state configuration (see Section 4.3).

Contradictory State Transition Error: A state variable is assigned multiple times in a state transition, taking secondary actions into account (see Section 4.2).

Redundant Conditions Warning: A condition on the decision tree does not alter the enabling state space of its child branch.

Redundant Condition Information Warning: A condition on the decision tree explicitly grants degrees of freedom which are prohibited by subsequent conditions.

Unclear Last-Transition Warning: A particular definition of a last-transition is enabled in more than one state configuration (see Section 4.4.2).

Unhandled State Configuration Warning: There are state configurations which possess no outgoing transitions. This validation function may yield false positives if said state configurations are unreachable in the first place (see Section 4.2).

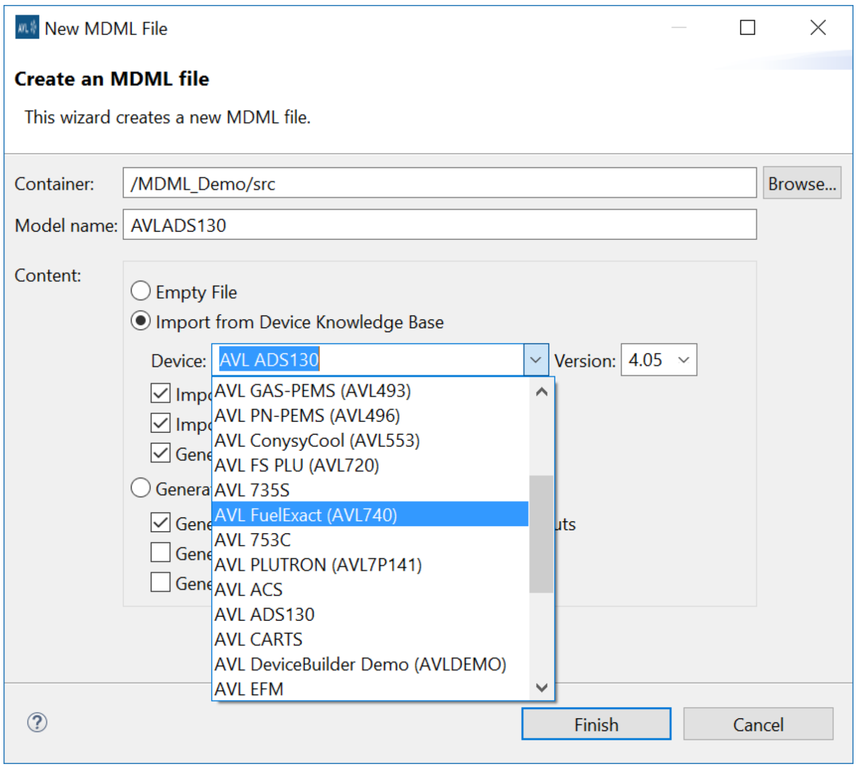
5.2.5 Tooltips

Per default, tooltips simply show the name of the grammar rules (see Section 4.5) which correspond to the model element over which the cursor currently hovers. We extended this information by a short static documentation text for each type of model element. Xtext tooltips support text formatting by HTML tags which can be used to support their readability. In addition to syntactic information, the MDML IDE in a few cases offers semantic tooltips which are described in Section 5.3.

5.2.6 Quickfixes

Quickfixes are offered as mitigation for specific validation violations. For example, Xtext supports the correction of misspelled state references in `given` statements under the condition that their scope has been defined. More sophisticated quickfixes have to be implemented by the DSL developer. See Section 5.3 for examples.

5.3 Device Knowledge Base Integration



New MDML File

Create an MDML file

This wizard creates a new MDML file.

Container: /MDML_Demo/src

Model name: AVLADS130

Content:
Empty File
[✓ Import from Device Knowledge Base]

Device: AVL ADS130 Version: 4.05

[✓ Imp]
[✓ Imp]
[✓ Gene]
Genera
[✓ Gene] uts
Gene
Gene

AVL GAS-PEMS (AVL493)
AVL PN-PEMS (AVL496)
AVL ConysyCool (AVL553)
AVL FS PLU (AVL720)
AVL 735S
AVL FuelExact (AVL740)
AVL 753C
AVL PLUTRON (AVL7P141)
AVL ACS
AVL ADS130
AVL CARTS
AVL DeviceBuilder Demo (AVLDEMO)
AVL EFM

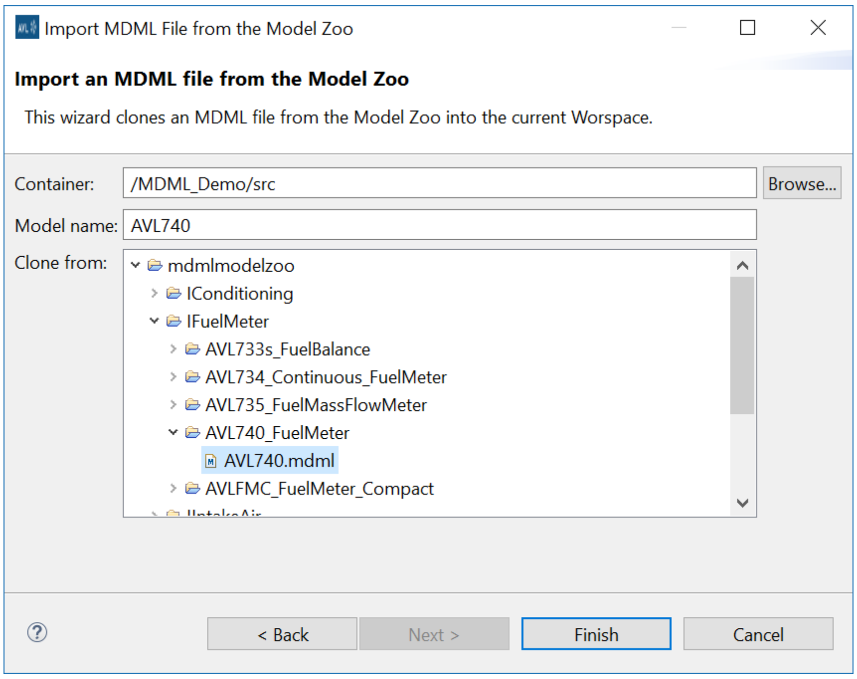
Figure 5.2: A screenshot of the new-file wizard which uses information from the DKB.

The information on which MDML models are based - e.g. states, input events, etc. - is originally generated by the device firmware developers. Until recently, however, the best source for the test engineers in terms of availability were the measurement device handbooks which are written by the documentation department. Even while these handbooks are not always kept perfectly up-to-date, the test engineers prefer them over other more complete or correct information sources because they are easily available. An effort to increase data transparency between different departments of AVL has recently led to the establishment of the so-called *device knowledge base* (DKB). While this database can be accessed from anywhere within the company, the measurement device development department holds the sole authority over its contents, thereby establishing a *single source of truth* (SSoT). The DKB contains the names and firmware versions for each device as well as the corresponding states and AK commands [57] which are used as input events. For the latter, each entry contains a short semantic description of its meaning or purpose. The MDML IDE uses this information in several different ways:

- Figure 5.2 depicts a wizard for the creation of new MDML files which offers the option to import device headers (name, firmware version, AK commands, states) from the DKB. The range of imported states and AK commands may vary depending on the selected firmware version.
- If the name of an existing MDML model can be matched to a device within the DKB, a quickfix offers the option to import the previously mentioned header information.

If, e.g. an input channel contains some of the device's AK commands, the remaining commands can be added via a quickfix.
- When adding AK commands to an input channel per hand, a full list of them is provided for the current device via content assist, including the semantic description of each command.
- The same semantic information is added to the tooltip text of each AK command.

5.4 Model Zoo Integration


Import MDML File from the Model Zoo

Import an MDML file from the Model Zoo

This wizard clones an MDML file from the Model Zoo into the current Worspace.

Container: /MDML_Demo/src

Model name: AVL740

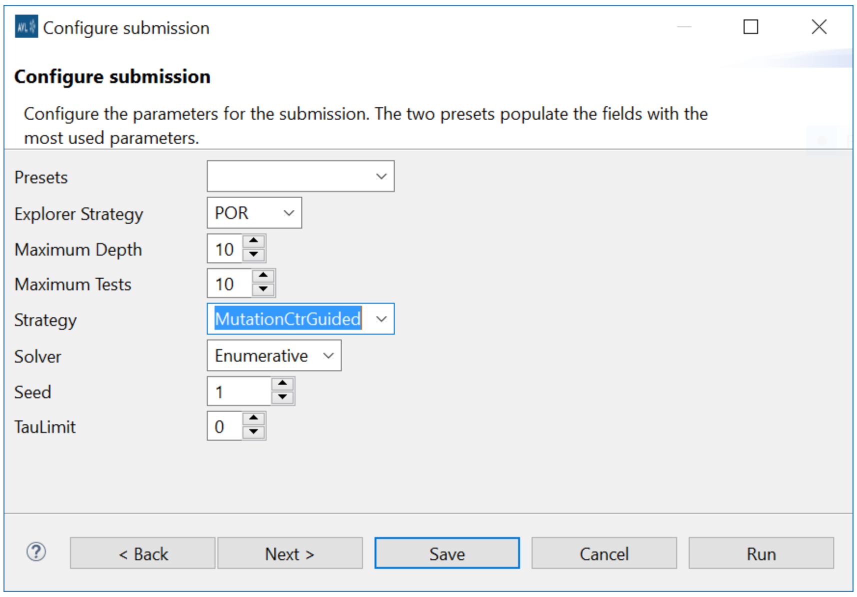
Clone from:
```
mdmlmodelzoo
  IConditioning
  IFuelMeter
    AVL733s_FuelBalance
    AVL734_Continuous_FuelMeter
    AVL735_FuelMassFlowMeter
    AVL740_FuelMeter
      AVL740.mdml
    AVLFMC_FuelMeter_Compact
```

Figure 5.3: A screenshot of the model zoo import wizard.

The so-called *model zoo* is the most recent addition to the list of features of the MDML IDE. As it turned out, our test engineers found it easier to create a model based on pre-existing content rather than entirely from scratch. Consequently, we established a centralized repository which we populated with all MDML models which are currently at our disposal. The MDML IDE offers the option to clone this repository via an encapsulated Git[6] implementation. As can be seen in Figure 5.3, a dedicated import wizard allows the user to browse the local model zoo repository and copy prototypical models to the Eclipse workspace. The model zoo was designed to be an SSoT under the authority of one or few administrators. While the work on an export functionality to the model zoo is ongoing, the exact details of the model submission and review process have not yet been defined.

[6] `https://www.eclipse.org/jgit/`

5.5 Test Case Generator Integration


Configure submission

Configure submission

Configure the parameters for the submission. The two presets populate the fields with the most used parameters.

Presets	
Explorer Strategy	POR
Maximum Depth	10
Maximum Tests	10
Strategy	MutationCtrGuided
Solver	Enumerative
Seed	1
TauLimit	0

Figure 5.4: A screenshot of the model submission wizard which starts the test case generator.

The TRUFAL toolchain was realized by three different tools, namely a UML editor, MoMuT::UML and the test case transformator, the latter two being the command line tools. By contrast, the TRUCONF toolchain is entirely encapsulated by the MDML IDE to uphold the design principle of *separation of concerns* [97]. The test case generation process can be configured by a dedicated wizard which gives the user control over several selected parameters of MoMuT (see Figure 5.4 for a screenshot of the wizard, as well as [68] for more details on the parameters). The test case generation process can be started either from within the wizard or by pressing a single button on the toolbar of the IDE if the configuration has been completed beforehand. The process starts by executing the code generator which transforms the MDML model into an OOAS. Then, MoMuT is started in an external process and generates a set of abstract test cases. The output of the MoMuT process is fed to the console view of the IDE to let the user monitor the progress. After MoMuT has terminated, the abstract test cases are read by the test case transformator which outputs a single .cs file containing the completed test suite into the currently opened project folder within the workspace. For an overview of the test case generation toolchain, see Section 1.5 as well as the Chapters 6, 7 and 8 for further details on the previously mentioned operations. In Figure 5.4, it is evident that the wizard for test case generation has been named "*Configure submission*". This is due to the fact that it is planned to move the test case generator to a centralized location in future versions of the toolchain. More details on this matter can be found in Section 10.2.2.

5.6 User Experience Evaluation

The notion of *user experience* (UX) goes far beyond the mere usability of a software tool [1, 98, 97]. Instead, a good user experience entails a minimization of disturbance to the users during the tool introduction phase as well as the general minimization of all negative emotions that the users might associate with the tool - e.g. the fear of losing control over ones own workflows or working conditions.

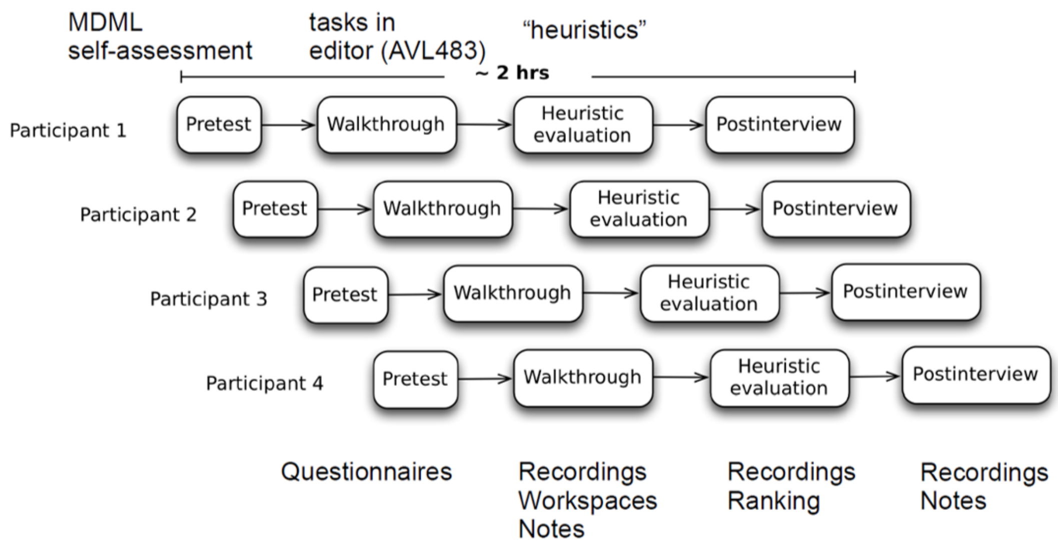


Figure 5.5: The structure of the heuristic walkthroughs [97, p.649].

Hence, the DLUX project was launched with the goal to develop an empirical user experience evaluation method kit and to apply it to AVL internal software tools - among others, the MDML IDE [97]. In addition to our regular feedback sessions, we conducted a series of so-called *heuristic walkthroughs* with our test engineers. A heuristic walkthrough is a usability inspection method which combines scenario-based and heuristic-based inspection processes [67]. First, our test engineers were handed a short questionnaire about their previous experience with MDML. Afterwards, they performed a series of short but practically relevant tasks with the MDML IDE while narrating their cognitive processes. Afterwards, they were asked to rate both the language and the IDE according to different criteria, e.g. user guidance, minimalistic design, efficiency, documentation, error mitigation, and others. The overall process, including a short wrap-up, took approximately two hours (see Figure 5.5). Afterwards, the transcripts of these interviews were analysed and the aggregated information was used as a guideline for the planning and prioritization of further development steps (e.g. the model zoo).

Aside from the heuristic walkthroughs, the user experience of MDML has been tested multiple times. The formalism was used by regular employees, student trainees and summer interns - both with and without prior experience of the application domain. In one notable case, a student trainee taught herself MDML without any supervision but solely based on the information presented in Chapter 4. The estimated effort for the relevant tasks from the first contact with the modelling formalism to creating a mature test model are given in Table 5.1. Our modelling formalism has been shown to exhibit an adequate learning curve - even without prior domain knowledge. Both the initial learning time and the time to create a first model draft have been reduced to 1-2 hours. The high debugging effort during TRUFAL was mainly caused by communication overhead due to the modeller being located off-site. As previously mentioned in Section 3.4.1, we believe that a bug-fixing and fine-tuning phase will always be necessary. Nevertheless, though we are still lacking long-term data, we expect to significantly reduce the necessary effort since - in addition to the improved user experience - the test engineer and the modeller will no longer be separate individuals.

Activity	**TRUFAL**	**TRUCONF**
Learning the modelling formalism	6	1-2
Implementing the first draft implementation	4	1-2
Debugging and fine-tuning	40	$\ll$40

Table 5.1: Compared effort in h of different tasks for the modelling methodologies used during the TRUFAL and TRUCONF projects [61, 97].

6 Model Transformation from MDML to OOAS

This chapter describes the model transformation from MDML models to object-oriented action systems which, in turn, can be processed by the MoMuT test case generator. From now on, we refer to the MDML-to-OOAS transformation as *model transformation* when focusing on the algorithm or *code generation* when focusing on the implementation. The requirements to the code generator have been defined as follows:

1. "*The code generator needs to be designed in such a way that it can be easily maintained and that functionality can be added or changed with little effort.*"

   Due to its modular implementation as by the *strategy pattern* [45], individual functionalities can be added or removed in the form of strategies.

   ---

2. "*The functionality of the code generator must be specified and verified according to the practice of test-driven development [24].*"

   During the implementation process of the code generator, the practice of test-driven development has been observed, which resulted in a verifiable specification for the transformation logic. As the time of writing this thesis, the test base for the code generator comprises 33 different unit tests.

   ---

3. "*The code generator must be able to handle complicated m:n relationships between model elements.*"

   The modular structure mentioned in Req. 1 keeps the individual strategies short and relatively simple. Together with a rigorous test-driven development practice as in Req. 2, this gave us the ability to handle the complexity of the code generator.

   ---

4. "*The model transformation needs to be designed in such a way that it produces as little unreachable code as possible.*"

   It was deemed important to limit the number of equivalent mutants since they consumed about 60% of test case generation time during experiments within the TRUFAL project [6]. However, the introduction of MoMuT's search-based back-end is also less susceptible to performance loss by equivalent mutants, depending on the chosen search strategy (see Section 7.1.3)

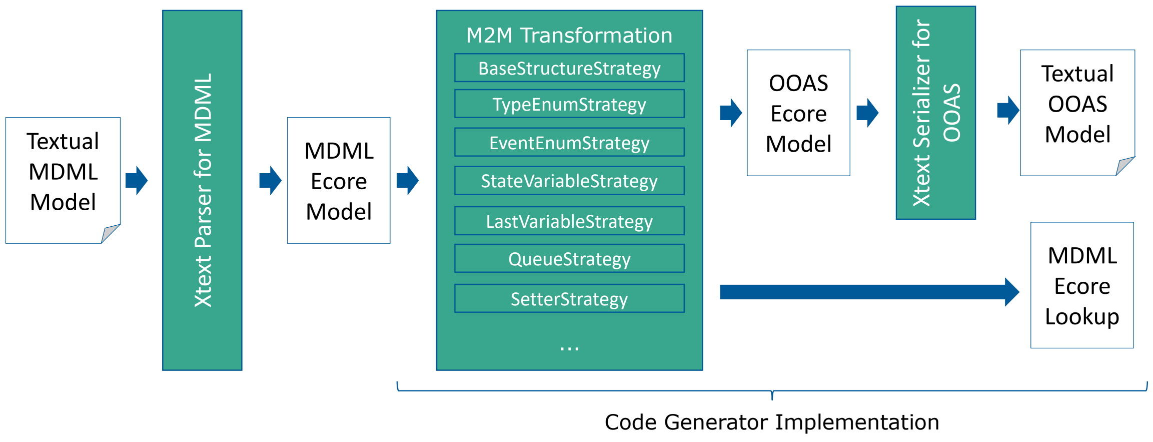


Figure 6.1: Software architecture of the OOAS code generator.

The rest of this chapter describes the architecture of the code generator, followed by a thorough description of the model transformation logic.

6.1 Code Generator Architecture

Since many MDML and OOAS model elements exhibit a complex m:n relationship, early attempts to produce a linear stream of output code while traversing the MDML model resulted in failure. As the need for an object model of the target representation became clear, we chose to implement an Xtext grammar for a subset of OOAS and to exploit the ability of the auto-generated serializer to transform Ecore models back into their textual representations, as depicted in Figure 6.1.

The remaining task of translating between the Ecore representations of MDML and OOAS was split up into a series of strategies as by the strategy pattern [45]. The first of these strategies creates the basic model structure (type definition block, system, variable block, etc.). Thereafter, each of the remaining strategies adds a particular feature to this structure (e.g. the variable setter methods). With the exception of the base structure strategy, all strategies can be executed in arbitrary order. Each strategy is described in detail in Section 6.2. Additionally, a lookup table is created to map named references of MDML model elements (like OOAS state variable names, event enum values, etc.) to their corresponding MDML Ecore objects. This lookup table is an artefact of the code generation process and serves as an additional input to the test case transformator.

When the OOAS Ecore model is complete, Xtext saves it as a resource. During this process, each Ecore model element is mapped to its representation in the abstract syntax tree of the OOAS grammar. The resulting textual representation is formatted according to some simple predefined rules (concerning line breaks, indents, etc.). This approach grants the ability to add, change or remove functionality by simply adding or removing individual strategies.

6.1.1 Ecore Model

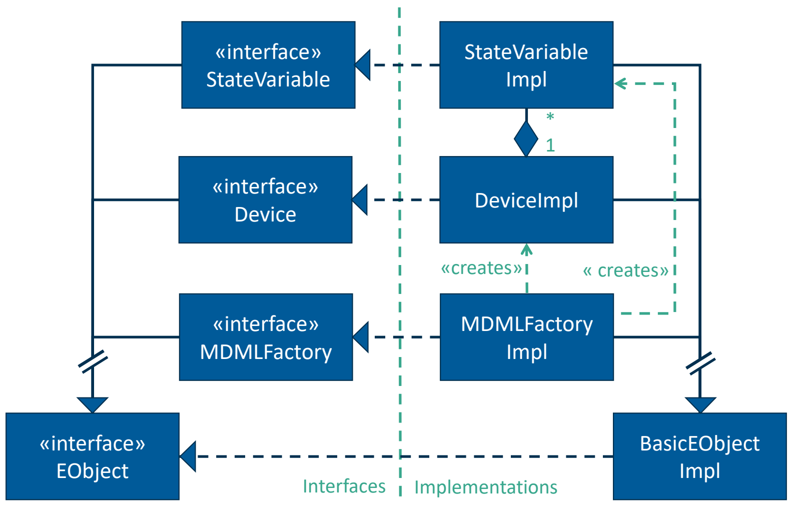


Figure 6.2: Excerpt from the Ecore-based object model of MDML.

We conclude the implementation-focused description of the code generator by giving a small impression of MDML's Ecore object model. This description is limited to the use case at hand and merely scratches the surface of this vast topic. A comprehensive reference on the Eclipse modelling framework, which includes the Ecore meta-model, has been provided by Steinberg et al. [96].

Figure 6.2 shows an excerpt of the class hierarchy which was generated from the MDML grammar. On first glance, it is evident that the object model strictly separates interfaces and implementations as per the *bridge pattern* [45]. The creators of the EMF imposed this design choice to adhere to best practice and to enable multiple inheritance [96, p.22]. All model elements indirectly derive from the *EObject* interface and its implementation, respectively. *EObject* provides basic functionality like resource handling, navigation through the model hierarchy or cross-referencing. Specific model elements like *Device* and *StateVariable* are generated from the MDML grammar and manage their respective attributes like device version numbers or initial states.

When interacting with objects from a parsed MDML model, the DSL engineer handles instances of the implementations but always addresses them through their interfaces. Therefore, new model elements cannot be created by constructor calls when building a model from scratch with *EObjects*. Instead, EMF uses the *abstract factory pattern* [45]. A factory interface declares creation methods for all types of model elements and the concrete factory provides the respective implementations.

6.2 Model Transformation Logic

6.2.1 Notation

Before describing the different transformation strategies, the formalisms used within these descriptions must be properly defined. These include rewrite rules, conventions on the usage of sets and vectors, as well as the definition of several Boolean predicates.

Rewrite Rules

The notation we use to describe the various rewrite rules was inspired by a model transformation notation used by Johannes Eriksson in his PhD thesis [41]:

$$P \xrightarrow{R_1} T$$

The above statement describes the rewrite rule R_1 which transforms a pattern P into the term T. If P contains further variables, they can be used on the right-hand side within T and be subjected to other rewrite rules. The instantiation of a rewrite rule is indicated by pointed brackets, followed by its superscript name:

$$P_1 \texttt{ or } P_2 \xrightarrow{R_2} \langle P_1 \rangle^{R_1} \;||\; \langle P_2 \rangle^{R_1}$$

In this way, all named rewrite rules of the model transformation are described, with the following exceptions: The rules *Name*, *EName*, *LName*, *OCName*, *SATName*, *SName* and *TName* map to character strings identifying different OOAS model elements which are derived from the same MDML model element. A full list of them has been given in Section 4.5. Different types of names and names of different MDML element types receive different prefixes to prevent naming collisions within the OOAS. These names are also used as index for their corresponding MDML model elements within the lookup table.

Vectors, Sets and Families

The description of the model transformation involves both vectors $\vec{\mathbf{X}}$ and sets $\mathbf{X}$ of elements. Both vectors and sets are written in boldface. Additionally, MDML element attributes like $\mathit{InputChannels}_D$, $\mathit{StateVariables}_D$, etc. can be identified as sets since they are designated by a *plural* noun. We use the convention that, within the definition of a rewrite rule, vectors and sets *of the same name* relate in the following way:

$$\forall i \in \{1, \ldots, \mathit{dim}(\vec{\mathbf{X}})\} : \vec{\mathbf{X}}_i \in \mathbf{X}$$
$$\forall x \in \mathbf{X}\; \exists i \in \{1, \ldots, \mathit{dim}(\vec{\mathbf{X}})\} : x = \vec{\mathbf{X}}_i$$

In this way, the set retains all elements of the vector but loses all information about multiplicity and order of elements. Sets are used whenever the transformation algorithm does not depend on the order of elements and/or when a given vector must be cleared of duplicate elements. Furthermore, we will often use indexed element sequences of the following form:

$$\vec{\mathbf{X}}_1 \dots \vec{\mathbf{X}}_m$$

While the indexing of vector elements is straightforward, the indexing of set elements requires the definition of a family $(\mathbf{X}_i)$ which is linked to $\mathbf{X}$ in the following way:

$$\forall x \in \mathbf{X}\ \exists i \in \{1, \dots, |\mathbf{X}|\} : x = \mathbf{X}_i$$

This measure effectively turns $\mathbf{X}$ into an *indexable set.* Moreover, whenever indexed sequences occur, we adhere to the convention that the sequence invariably iterates over *all* elements of the vector or set. We use this shorthand notation to avoid the explicit definition of index ranges on every occurrence.

$$\begin{aligned} \vec{\mathbf{X}}_1 \dots \vec{\mathbf{X}}_m &\Rightarrow m = dim(\vec{\mathbf{X}}) \\ \mathbf{X}_1 \dots \mathbf{X}_n &\Rightarrow n = |\mathbf{X}| \end{aligned}$$

The alteration of a set over the course of the model transformation process is signified by the transformation operator ($\mapsto$).

Boolean Predicates

The description of the model transformation logic employs several Boolean predicates in conditional statements or set definitions. These predicates are defined as follows:

The predicate $hasSA(s)$ is true iff the MDML model of the device D contains a secondary action trigger `when` v `->` s with $v \in StateVariables_D, s \in States_v$. Furthermore:

$$\begin{aligned} hasSA(v) &\Leftrightarrow \exists s \in States_v : hasSA(s) \\ hasSA(D) &\Leftrightarrow \exists v \in StateVariables_D : hasSA(v) \end{aligned}$$

The predicate $hasLT(v)$ is true iff the MDML model of the device D contains a last-transition v `-> last` with $v \in StateVariables_D$. The predicate $public(v)$ is true iff the state variable $v \in StateVariables_D$ has been declared public upon definition.

6.2.2 Base Structure Strategy

The first strategy adds the basic structural blocks of every OOAS to the empty model. This includes a type definition block **T** with a single class definition C which, in turn, receives a variable definition block **V**, a method definition block **M**, an action definition block **A** and a `do-od` block $DOOD$. Finally, the system assembly block SAB references the single class definition. The blocks **V**, **M**, **A** and $DOOD$ are implicitly assumed to be attributes of the class C. The formal description of the transformation step performed by this strategy is given below, followed by a code snippet to give an impression of the transformation result. We will use the same structure to describe all subsequent strategies.

$$\begin{aligned}
\mathbf{T} :=& \quad \{C(Name_D)\} \\
\mathbf{V} :=& \quad \{\} \\
\mathbf{M} :=& \quad \{\} \\
\mathbf{A} :=& \quad \{\} \\
DOOD & \quad \text{not yet defined} \\
SAB :=& \quad C
\end{aligned}$$

```
types

  AVL740 = autocons system
  |[
  var

  methods

  actions

  do

  od
  ]|
system AVL740
```

6.2.3 Type Enum Strategy

For each state variable present in the MDML model, an enum type encompassing all its possible values is added to the type definition. These values are prefixed with the name of the state variable to allow different state variables to contain a state of equal name while still preserving the ability to index the original *EObject* by name.

$$\mathbf{T} \mapsto \mathbf{T} \cup \{\langle v \rangle^{Type} \mid v \in StateVariables_D\}$$

$$v \xrightarrow{\text{Type}} \langle v \rangle^{TName} = \{\mathbf{S}_1^N, \ldots, \mathbf{S}_n^N\}$$

$$\mathbf{S}^N = \{\langle s \rangle^{Name} \mid s \in States_v\}$$

```
types
  t_DeviceState = {DeviceState_Pause , ... , DeviceState_Measurement} ;
  t_Status = {Status_Ready , Status_Busy} ;
  ...
```

6.2.4 Event Enum Strategy

Another enum type is created for all possible events which can trigger a `when` statement. This includes input events, secondary actions and time triggers. Note that the type name is prefixed with "*e_*" instead of "*t_*" to prevent name collisions with possible state variables called "Event".

$$\begin{aligned}\mathbf{T} &\mapsto \mathbf{T} \cup \{T_{Event}\ (\mathbf{E}^N)\}\\ \mathbf{E}^N &= \{\langle e\rangle^{Name} \mid e \in Events_c,\ c \in InputChannels_D\} \cup\\ &\quad \{\langle s\rangle^{EName} \mid s \in States_v,\ v \in StateVariables_D,\ hasSA(s)\} \cup\\ &\quad \{\langle t\rangle^{Name} \mid t \in TimeTriggers_D\}\end{aligned}$$

```
types
  ...
  e_Event = {i_UserAction_SPAU , ... , tt_TimeTriggerImpl585ce5c8} ;
  ...
```

6.2.5 State Variable Strategy

Each state variable of the MDML model receives a direct representation in the variable definition block which is of the previously created enum type. Each OOAS state variable is initialized with the initial state specified in the MDML model.

$$\mathbf{V} \mapsto \mathbf{V} \cup \{\langle v\rangle^{Var} \mid v \in StateVariables_D\}$$
$$v \xrightarrow{\text{Var}} \langle v\rangle^{Name}\ \texttt{:}\ \langle v\rangle^{TName}\ \texttt{=}\ \langle InitialState_v\rangle^{Name}$$

```
var
  v_DeviceState : t_DeviceState = DeviceState_Pause ;
  v_Status : t_Status = Status_Ready
```

6.2.6 Last-Variable Strategy

If a state variable experiences last-transitions at some point, it needs a backup variable of the same type to store its value previous to the last-transition. The last-variable is initialized with the same state as its corresponding state variable. Therefore, taking a last-transition from the initial state has no effect. While the model of AVL740 contains no such feature, an example from another model is given below:

$$\mathbf{V} \mapsto \mathbf{V} \cup \{\langle v\rangle^{LVar} \mid v \in StateVariables_D,\ hasLT(v)\}$$
$$v \xrightarrow{\text{LVar}} \langle v\rangle^{LName}\ \texttt{:}\ \langle v\rangle^{TName}\ \texttt{=}\ \langle InitialState_v\rangle^{Name}$$

```
var
  ...
  last_DeviceState : t_DeviceState = s_DeviceState_Pause
```

6.2.7 Secondary Action Queue Strategy

If the model contains secondary actions, a corresponding queue V_{SAQ} in the form of an array of type *e_Event* must be added to the variables. The secondary action queue is limited in length to the number of state variables present in the model which constitutes a simple upper limit of possible consecutive secondary actions. If the model does not contain secondary actions, this strategy produces no change in order to prevent the generation of unreachable code. Several of the following strategies employ similar conditional transformations for the same reason. The model of AVL740 does not contain secondary actions and therefore lacks this feature and all other features related to secondary actions.

$$\mathbf{V} \mapsto \begin{cases} \mathbf{V} \cup \{V_{SAQ}(|StateVariables_D|)\} & hasSA(D) \\ \mathbf{V} & \text{otherwise} \end{cases}$$

```
var
  ...
  secondary_action_queue : list [2] of e_Event = [nil]
```

6.2.8 Queue Secondary Action Strategy

If the model contains secondary actions, the secondary action queuing method M_{QSA} gets added. The content of this method is *static* in the sense that it is independent from the MDML model.

$$\mathbf{M} \mapsto \begin{cases} \mathbf{M} \cup \{M_{QSA}\} & hasSA(D) \\ \mathbf{M} & \text{otherwise} \end{cases}$$

```
methods
  ...
  queueSecondaryAction(event : e_Event) =
    secondary_action_queue := secondary_action_queue ^ [event]
  end
```

6.2.9 Dequeue Secondary Action Strategy

Similarly, if the model contains secondary actions, a static dequeuing method M_{DSA} must be added which removes and returns the first element in the secondary action queue.

$$\mathbf{M} \mapsto \begin{cases} \mathbf{M} \cup \{M_{DSA}\} & hasSA(D) \\ \mathbf{M} & \text{otherwise} \end{cases}$$

```
methods
  ...
  dequeueSecondaryAction : e_Event =
    result := hd secondary_action_queue ;
    secondary_action_queue := tl secondary_action_queue
  end
```

6.2.10 Secondary Action Trigger Strategy

Let v be a state variable which, at some point, triggers secondary actions. Whenever v experiences a transition which causes secondary actions, the transition must be added to the secondary action queue. Hence, a method must be created for each v which takes its newly assigned value as a parameter, queues the corresponding secondary action via the queuing method M_{QSA} if necessary and skips otherwise:

$$\mathbf{M} \mapsto \mathbf{M} \cup \{\langle v\rangle^{SAT} \mid v \in StateVariables_D,\ hasSA(v)\}$$

$$v \xrightarrow{\text{SAT}} \langle v\rangle^{SATName}\ \texttt{(value : }\langle v\rangle^{TName}\texttt{) = (}\mathbf{S}_1^Q \ \square \ldots \square\ \mathbf{S}_n^Q\texttt{) // skip}$$

$$\mathbf{S}^Q = \{\langle s\rangle^{QSA} \mid s \in States_v,\ hasSA(s)\}$$

$$s \xrightarrow{\text{QSA}} \texttt{requires value = }\langle s\rangle^{Name}\texttt{ : }M_{QSA}(\langle s\rangle^{EName})\texttt{ end}$$

```
methods
  ...
  trigger_DeviceState_SecondaryActions(value : t_DeviceState) =
    (
      requires value = s_DeviceState_Pause :
        queueSecondaryAction(se_DeviceState_Pause)
      end []
      requires value = s_DeviceState_Standby :
        queueSecondaryAction(se_DeviceState_Standby)
      end
    ) //
    skip
  end
```

6.2.11 Changed-Observable Strategy

Each public state variable receives a dedicated observable action which reports its changed value to an outside observer. This action performs no further operations and therefore only contains a `skip` action. In accordance with the remark in Section 4.4.4, the changed-observable is not emitted if the value of a state variable does not change during a transition. This behaviour is encoded in the setter method of each state variable.

$$\mathbf{A} \mapsto \mathbf{A} \cup \{\langle v\rangle^{OC} \mid v \in \mathit{StateVariables}_D,\ \mathit{public}(v)\}$$
$$v \xrightarrow{\text{OC}} \mathit{Obs}_C(\langle v\rangle^{OCName},\ \langle v\rangle^{Type})$$

```
actions
  obs obs_DeviceState_Changed(value : t_DeviceState) = requires true :
    skip
  end ;

  obs obs_Busy_Changed(value : t_Busy) = requires true :
    skip
  end
```

6.2.12 State Variable Setter Strategy

For each state variable, a setter method is created. This method handles all additional operations related to the assignment of this state variable. If a state variable is private and experiences no last-transitions or secondary actions, the setter method takes a very simple form. It takes the argument *actual_value* and sets the state variable to this value.

$$\mathbf{M} \mapsto \mathbf{M} \cup \{M_{Set}(v) \mid v \in \mathit{StateVariables}_D\}$$
$$M_{Set}(v) = \begin{cases} M_{SetSimple}(v) & \neg \mathit{public}(v) \wedge \neg \mathit{hasLT}(v) \wedge \neg \mathit{hasSA}(v) \\ M_{SetFull}(v) & \text{otherwise} \end{cases}$$
$$M_{SetSimple}(v) = \langle v\rangle^{SName}(\langle v\rangle^{AA}) \texttt{ = } X$$
$$X = \langle v\rangle^{Name} \texttt{ := actual\_value}$$
$$v \xrightarrow{\text{AA}} \texttt{actual\_value:}\langle v\rangle^{TName}$$

```
methods
  ...
  # Example. AVL740's Busy is actually public.
  setBusy(actual_value : t_Busy) =
    v_Busy := actual_value
  end
```

If at least one of the aforementioned conditions is violated, the content of the setter method is guarded with the condition that the assigned value is different from the current value of the state variable. If the state variable experiences last-transitions, its value is copied to the last-variable prior to being set. After a public state variable has been set, its changed-observable is triggered to report the newly assigned value to the environment. For public state variables, the setter method takes an additional argument *reported_value* which serves as an input for the observable. In the original OOAS, setter methods are always called with the same assignment of both *reported_value* and *actual_value*. The method is set up in this way so that both values can be mutated independently from each other. Lastly, if the state variable triggers secondary actions, the call to its secondary action trigger is inserted.

$$M_{SetFull}(v) = \begin{cases} \langle v \rangle^{SName}\ (\langle v \rangle^{AR},\ \langle v \rangle^{AA}) \texttt{ = } Y & public(v) \\ \langle v \rangle^{SName}\ (\langle v \rangle^{AA}) \texttt{ = } Y & \text{otherwise} \end{cases}$$

$$Y = \texttt{requires actual\_value <> } \langle v \rangle^{Name}\ : Z \texttt{ end } /\!/ \texttt{ skip}$$

$$Z = \begin{cases} \langle v \rangle^{LName} \texttt{ := } \langle v \rangle^{Name} \texttt{;}\ A & hasLT(v) \\ A & \text{otherwise} \end{cases}$$

$$A = \begin{cases} B\,;\langle v \rangle^{SATName}\ \texttt{(actual\_value)} & hasSA(v) \\ B & \text{otherwise} \end{cases}$$

$$B = \begin{cases} X\,;\langle v \rangle^{OCName}\ \texttt{(reported\_value)} & public(v) \\ X & \text{otherwise} \end{cases}$$

$$v \xrightarrow{\text{AR}} \texttt{reported\_value : } \langle v \rangle^{TName}$$

```
methods
  ...
  setDeviceState(reported_value : t_DeviceState , actual_value : t_DeviceState) =
    requires actual_value <> v_DeviceState :
      # last_DeviceState := v_DeviceState;
      v_DeviceState := actual_value ;
      obs_DeviceState_Changed(reported_value) #;
      # trigger_DeviceState_SecondaryActions(actual_value)
    end //
    skip
  end ;
```

6.2.13 Tree Action Strategy

Whenever the OOAS processes an event, it is evaluated via the decision tree, specified in MDML. However, under certain circumstances, certain additional operations have to be performed, either before or after decision tree traversal. These additional actions can be called in a static action which encapsulates the call of the decision tree.

$$\mathbf{A} \mapsto \mathbf{A} \cup \{Int_{TA}\}$$

```
actions
  ...
  treeAction(event : e_Event) = requires true :
    # Housekeeping actions go here
    tree(event)
    # Housekeeping actions go here
  end
```

While all of these "housekeeping" operations have been removed during the development of the MDML-to-OOAS transformation, we left *treeAction* in the OOAS as an intermediate calling stage. This did not impact test case generation in a negative way and proved quite helpful when we introduced last-minute changes of the model transformation (see Section 9.4) .

6.2.14 Tree Strategy

The decision tree within the MDML model directly carries over to the OOAS in the form of the action *tree* which takes the current event and processes it depending on the current state configuration. The transformation of model elements within the tree is described by the recursive rewrite rule $\langle X \rangle^T$. $Body_D$ is the root of the decision tree structure of device D. In every case, $Body_D$ maps to a behaviour list $\vec{\mathbf{B}}_1 \ldots \vec{\mathbf{B}}_m$. Such behaviour lists also occur in block form when preceded by `given` statements.

$$\begin{aligned}
\mathbf{A} &\mapsto \mathbf{A} \cup \{Int_T(D)\} \\
Int_T(D) &= \texttt{tree(event : e\_Event) requires true : } \langle D.Body \rangle^T \texttt{ end} \\
\vec{\mathbf{B}}_1 \ldots \vec{\mathbf{B}}_m &\xrightarrow{\mathrm{T}} \langle \vec{\mathbf{B}}_1 \rangle^T \square \ldots \square \langle \vec{\mathbf{B}}_m \rangle^T \\
\{\vec{\mathbf{B}}_1 \ldots \vec{\mathbf{B}}_m\} &\xrightarrow{\mathrm{T}} \langle \vec{\mathbf{B}}_1 \rangle^T \square \ldots \square \langle \vec{\mathbf{B}}_m \rangle^T
\end{aligned}$$

```
actions
  ...
  tree(event : e_Event) = requires true :
    ...
  end
```

Given Statements

As described in Section 4.1.4, `given` statements can encode different equality and set-based conditions on state variables. Although OOAS offers an equivalent of the $\in$ operator, element inclusion was translated as a disjunction of equality conditions. This offers the potential to produce more fine-grained mutants upon test case generation. Note that list-based conditions are cleared of duplicates to reduce the potential of equivalent mutants (see Section 7.3 for more information).

$$\texttt{given}\ \mathit{condition}\ \mathit{content} \xrightarrow{\mathrm{T}} \texttt{requires}\ \langle \mathit{condition}\rangle^{GS}\ \texttt{:}\ \langle \mathit{content}\rangle^{T}\ \texttt{end}$$

$$v\ \texttt{=}\ s \xrightarrow{\mathrm{GS}} \langle v\rangle^{Name}\ \texttt{=}\ \langle s\rangle^{Name}$$

$$v\ \texttt{!=}\ s \xrightarrow{\mathrm{GS}} \langle v\rangle^{Name}\ \texttt{<>}\ \langle s\rangle^{Name}$$

$$v\ \texttt{in}\ \{\vec{\mathbf{S}}_1 \ldots \vec{\mathbf{S}}_m\} \xrightarrow{\mathrm{GS}} \langle v\ \texttt{=}\ \mathbf{S}_1\rangle^{GS}\ \texttt{or} \ldots \texttt{or}\ \langle v\ \texttt{=}\ \mathbf{S}_n\rangle^{GS}$$

$$v\ \texttt{not in}\ \{\vec{\mathbf{S}}_1 \ldots \vec{\mathbf{S}}_m\} \xrightarrow{\mathrm{GS}} \texttt{not}(\langle v\ \texttt{=}\ \mathbf{S}_1\rangle^{GS}\ \texttt{or} \ldots \texttt{or}\ \langle v\ \texttt{=}\ \mathbf{S}_n\rangle^{GS})$$

```
# given  DeviceState = Pause
requires v_DeviceState = Pause : ... end ...
# given DeviceState != Pause
requires v_DeviceState <> Pause : ... end ...
# given DeviceState in {Pause, Standby}
requires v_DeviceState = Pause or v_DeviceState = Standby : ... end ...
# given DeviceState not in {Pause, Standby}
requires not (v_DeviceState = Pause or v_DeviceState = Standby) : ... end ...
```

When Statements

A `when` statement can be triggered by input events (see Section 4.1.5), state transitions (as a secondary action, see Section 4.4.1) or by timed behaviours (see Section 4.3). In the latter two cases, the rewrite rules are straightforward. In the case of input events, however, the situation is more complicated. Since the grouping of input events into input channels has never been explicitly defined in the OOAS, the conditions involving `!=`, `not in` and `= any` cannot simply be written as inequalities or set exclusions. Instead they are written as set inclusions on all possible input events of channel c, minus those which are excluded by the `when` statement. Again, the list-based expressions are cleared of duplicates to reduce the potential of equivalent mutants (see Section 7.3 for more information).

$$\texttt{when}\ \textit{trigger reaction} \xrightarrow{\text{T}} \texttt{requires}\ \langle \textit{trigger}\rangle^{WS}\ \texttt{:}\ \langle \textit{reaction}\rangle^{T}\ \texttt{end}$$

$$v\ \texttt{->}\ s \xrightarrow{\text{WS}} \texttt{event}\ \texttt{=}\ \langle s\rangle^{EName}$$

$$t\ \texttt{elapsed} \xrightarrow{\text{WS}} \texttt{event}\ \texttt{=}\ \langle t\rangle^{Name}$$

$$e \xrightarrow{\text{IE}} \texttt{event}\ \texttt{=}\ \langle e\rangle^{Name}$$

$$c\ \texttt{=}\ e \xrightarrow{\text{WS}} \langle e\rangle^{IE}$$

$$c\ \texttt{in}\ \{\vec{\mathbf{E}}_1, \dots, \vec{\mathbf{E}}_m\} \xrightarrow{\text{WS}} \langle \{\langle e\rangle^{IE} \mid e \in \mathbf{E}\}\rangle^{OR}$$

$$\mathbf{X} \xrightarrow{\text{OR}} \mathbf{X}_1\ \texttt{or} \dots \texttt{or}\ \mathbf{X}_n$$

$$\mathcal{E}(c, X) = \{\langle e\rangle^{IE} \mid e \in \textit{Events}_c \setminus X\}$$

$$c\ \texttt{!=}\ e \xrightarrow{\text{WS}} \langle \mathcal{E}(c,\ \{e\})\rangle^{OR}$$

$$c\ \texttt{=}\ \texttt{any} \xrightarrow{\text{WS}} \langle \mathcal{E}(c,\ \{\})\rangle^{OR}$$

$$c\ \texttt{not in}\ \{\vec{\mathbf{E}}_1, \dots, \vec{\mathbf{E}}_m\} \xrightarrow{\text{WS}} \langle \mathcal{E}(c,\ \{\mathbf{E}_1, \dots, \mathbf{E}_n\})\rangle^{OR}$$

```
# when DeviceState -> Pause
requires event = s_DeviceState_Pause : ... end ...
# when 3 sec elapsed
requires event = tt_TimeTriggerImpl585ce5c8 : ... end ...
# when UserAction = SPAU
requires event = i_UserAction_SPAU : ... end ...
# when UserAction != STBY
# when UserAction in {SPAU, SMES, ...}
# when UserAction not in {STBY, ...}
# when UserAction = any (list includes STBY)
requires event = i_UserAction_SPAU or event = i_UserAction_SMES or ... : ... end ...
```

Then Statements

The leaves of the decision tree are represented by `then` statements. Each `then` statement includes a vector of reactions $\vec{\mathbf{R}}$ which are executed consecutively. Reactions can either be state transitions (see Section 4.1.6), self-transitions (see Section 4.4.4) or last-transitions (see Section 4.4.2). Transitions are encoded as calls to the setter method of the respective state variable. In the case of simple state transitions, the method call receives the target state enum value as its argument. In the case of last-transitions, the last-variable itself serves as the setter argument. If the state variable is public, the method call receives a second argument which is identical to the first one.

$$\texttt{then } \vec{\mathbf{R}}_1 \texttt{ and} \ldots \texttt{and } \vec{\mathbf{R}}_M \,; \xrightarrow{\text{T}} \langle \vec{\mathbf{R}}_1 \rangle^{TS} \;;\ldots;\; \langle \vec{\mathbf{R}}_M \rangle^{TS}$$

$$\texttt{DoNothing} \xrightarrow{\text{TS}} \texttt{skip}$$

$$v \texttt{ -> } s \xrightarrow{\text{TS}} \begin{cases} \langle v \rangle^{SName} \; (\langle s \rangle^{Name},\; \langle s \rangle^{Name}) & public(v) \\ \langle v \rangle^{SName} \; (\langle s \rangle^{Name}) & \text{otherwise} \end{cases}$$

$$v \texttt{ -> last} \xrightarrow{\text{TS}} \begin{cases} \langle v \rangle^{SName} \; (\langle v \rangle^{LName},\; \langle v \rangle^{LName}) & public(v) \\ \langle v \rangle^{SName} \; (\langle v \rangle^{LName}) & \text{otherwise} \end{cases}$$

```
# then DoNothing
skip
# then DeviceState -> Pause
setDeviceState(s_DeviceState_Pause , s_DeviceState_Pause)
# then Busy -> False (private, example)
setBusy(s_Busy_False)
# then DeviceState -> last
setDeviceState(last_DeviceState , last_DeviceState)
# then Busy -> last (private, example)
setBusy(last_Busy)
```

6.2.15 Primary Action Strategy

The controllable *Primary* action is the sole means by which an outside observer can supply input to the OOAS. It is a static action which calls the *treeAction* without any additional operations.

$$\mathbf{A} \mapsto \mathbf{A} \cup \{Ctr_{PA}\}$$

```
actions
  ...
  ctr ctr_PrimaryAction(event : e_Event) = requires true :
    treeAction(event)
  end
```

6.2.16 Secondary Action Strategy

If the model contains secondary actions, they are processed via a dedicated static action. A secondary action is removed from the front of the queue. If the decision tree cannot process it further, the secondary action is simply "swallowed". This action is disabled if the secondary action queue is empty.

$$\mathbf{A} \mapsto \begin{cases} \mathbf{A} \cup \{Int_{SA}\} & hasSA(D) \\ \mathbf{A} & \text{otherwise} \end{cases}$$

```
actions
  ...
  int_SecondaryAction = requires len secondary_action_queue > 0 :
    treeAction(dequeueSecondaryAction()) //
    dequeueSecondaryAction()
  end ;
```

6.2.17 Do-Od Block Strategy

Finally, the `do-od` block is filled with a disjunction of all possible *PrimaryActions* - first and foremost, those triggered by input events. While a more intricate handling of time triggers has been planned, they are currently processed in the same way as input events. If the model contains secondary actions, the `do-od` block also needs to contain a call to *SecondaryAction* which processes all pending internal transitions and gets priority over all *PrimaryActions*.

$$\mathbf{DOOD} := \begin{cases} Int_{SA} \mathbin{/\!/} X & hasSA(D) \\ X & \text{otherwise} \end{cases}$$

$$X = \mathbf{E}_1^P \,\square \ldots \square\, \mathbf{E}_m^P \,\square\, \mathbf{T}_1^P \,\square \ldots \square\, \mathbf{T}_n^P$$

$$\mathbf{E}^P = \{\langle e\rangle^{PA} \mid e \in Events_c,\ c \in InputChannels_D\}$$

$$\mathbf{T}^P = \{\langle t\rangle^{PA} \mid t \in TimeTriggers_D\}$$

$$y \xrightarrow{\text{PA}} Ctr_{PA}(\langle y\rangle^{Name})$$

```
do
  # int_SecondaryAction() //
  ctr_PrimaryAction(ie_UserAction_SPAU) []
  ...                                   []
  ctr_PrimaryAction(ie_UserAction_SMES) []
  ctr_PrimaryAction(tt_TimeTriggerImpl2b097067) []
  ...                                           []
  ctr_PrimaryAction(tt_TimeTriggerImpl237abfe5)
od
```

7 Test Case Generation with MoMuT

7.1 Tool Overview

MoMuT::UML [64] (read: "MoMuT for UML") is an academic tool for model-based mutation testing of state machines, created by the Austrian Institute of Technology (AIT), as well as the Graz University of Technology. The front-end supports a defined subset of UML [63] which is translated to OOAS for further processing. Alternatively, OOAS is also accepted as a direct input language which makes the tool suitable for our DSL use case. Additionally, the MoMuT tool family also includes MoMuT::TA [12] for timed automata (a variant of LTS, extended by timing constraints [17]), as well as MoMuT::REQs for requirement-driven test case generation [9]. Since its inception during the MOGENTES project, MoMuT::UML went through several development stages which are described below:

7.1.1 Generation 1: Enumerative Back-End

The first back-end of MoMuT::UML was a conformance checker named *Ulysses* [7]. As its input, *Ulysses* takes the original OOAS, as well as a mutated version. It computes the LTSs of both systems in a breadth-first manner and combines them into a product graph while checking **ioco** on-the-fly. *Ulysses* enumerates all possible system traces which results in a high memory consumption and restricts its applicability to simple and slightly complex models [8, 56, 62]. Therefore, *Ulysses* requires data type ranges to be restricted within the OOAS definitions to mitigate a state space explosion [64].

7.1.2 Generation 2: Symbolic Back-End

During TRUFAL, a symbolic conformance checker was developed as an alternative to *Ulysses'* enumerative approach [56]. Like *Ulysses*, the system was written in Prolog, but it also interfaces to Microsoft's Z3[1] solver [69]. While the former enumerated all possible system traces, the latter approached test case generation as a constraint satisfaction problem. In addition to **ioco**, the symbolic back-end utilizes/offers another conformance relation called *refinement* [11]. Compared to **ioco**, refinement is far more strict since it imposes a condition on the internal state of a system rather than on observables:

$$I \text{ refines } S =_{df} \forall\, \mathbf{V}, \mathbf{V}' : I(\mathbf{V}, \mathbf{V}') \Rightarrow S(\mathbf{V}, \mathbf{V}')$$

[1] `https://github.com/Z3Prover`

Here, both I and S denote (object-oriented) action systems, $\mathbf{V}$ and $\mathbf{V}'$ denote sets of system-internal variables encoding the pre- and post-state of an arbitrary action. Furthermore, if refinement is fulfilled, **ioco** also holds [105]:

$$I \text{ refines } S \Rightarrow I \textbf{ ioco } S$$

The symbolic back-end starts by performing a refinement check on the mutant which is computationally significantly cheaper than an **ioco** check. Only if a refinement violation is detected, a targeted **ioco** check is performed. This approach resulted in a runtime reduction of up to 90% compared to *Ulysses*. In terms of resources, the algorithm is tied to CPU speed, rather than memory capacity. During experiments performed within the TRUFAL project, the symbolic back-end was able to cope with tasks which caused the enumerative back-end to run out of memory [6].

7.1.3 Generation 3: Search-Based Back-End

At the beginning of the TRUCONF project, MoMuT::UML entered an extensive modification process with the goal of increasing its performance to make it applicable to large-scale models as they are used in other industrial branches [62]. The working principle of the back-end was changed from formal conformance checking to a search-based approach which is also more suitable for workflow parallelization. The programming language was switched to C++ in order to have better control over resource consumption. Other than for the enumerative and symbolic back-ends, no mutated models are generated. Instead, information about each individual mutation is added to the original model which is then compiled to machine code to allow for an efficient execution. Thereafter, the model is executed under the control of an external scheduler that decides which model traces should be explored. If the execution trace reaches a point were a mutation becomes effective, the corresponding mutant is marked as *found*. If a found mutant crashes or can be made to show observable behaviour which constitutes an **ioco** violation, it is classified as killed. A found mutant which could not be shown to violate refinement of the original model is classified as equivalent[2]. If a mutant could not be killed despite violating refinement, it is classified as *weakly killed* [75]. The research on adequate search strategies is still ongoing. A recent publication [43] details the use of rapidly exploring random trees (RRTs) which were designed to be applicable to a broad class of path planning problems [66]. This strategy requires the definition of distance metrics between state configurations as well as different heuristics for start and successor states for the currently explored branch. Another important strategy is breadth-first search (BFS). While this strategy is still under development, it was used in our case study due to its ability to find all reachable mutants (see Chapter 9).

[2]The corresponding algorithm is still under development. Since it is currently unable to show that the internal state of the mutant is identical to the internal state of the original under *all* circumstances, MoMuT::UML currently does not exclude equivalent mutants from the mutation score.

7.2 Mutation Operators

In 1996, Offutt et al. [74] conducted a study to determine which mutation operators contribute the most to the mutation score of a test suite. The OOAS-related mutation operators of MoMuT [68] are largely consistent with those found by the study. Another study by Offutt et al. [76] has shown that mutation coverage on some of these operators subsumes various structural coverage criteria. A list of all mutation operators which are relevant to our use case, offered by MoMuT or otherwise relevant to the above mentioned studies are given below:

Absolute Value Insertion (ABS): For our use case, the relevant ABS operations are *enum value replacement* and *guard falsification*. At the time of our experiments, enum value replacement was only supported in assignment actions. Guard falsification replaces the guard expression of a guarded command with `false` which, depending on the individual circumstances, can change or eliminate many system traces at once. MoMuT also offers the replacement of integer and Boolean values which are not relevant to our use case.

Arithmetic Operator Replacement (AOR): Arithmetic operators ($+$, -, $\cdot$, $\div$, mod) do not occur in our OOAS architecture and therefore do not contribute to the generated set of mutants.

Logical Connector Replacement (LCR): The replacement of logical connectors ($\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) greatly contributes to our mutant sets, as it is later described in Section 7.3. Coverage of LCR mutants ensures *decision coverage* and, in combination with ROR, *decision condition coverage*, as well as *multiple condition coverage*.

Relational Operator Replacement (ROR): In relational expressions on enum values, MoMuT supports the operators $=$ and $\neq$. In special cases, the OOAS includes relational expressions on integers which elicit the full range of ROR mutations ($=$, $\neq$, $>$, $<$, $\geq$, $\leq$). Coverage of ROR mutants ensures *condition coverage* and, in combination with LCR, *decision condition coverage*, as well as *multiple condition coverage*.

Statement Analysis (SAN): This mutation operator ensures *statement coverage* by replacing individual statements with a `fail` statement which results in a kill as soon as it is reached. Currently, MoMuT does not support such an operator.

Unary Operator Insertion (UOI): Of this category, only the mutation $\neg e \mapsto e$ is relevant to our use case. MoMuT further supports mutations on unary integer expressions (abs, $+$, -).

Set-Based Operators: Although not examined by Offutt et al. [76, 74], nor relevant to our use case, MoMuT provides mutations on set-based operators and quantifiers ($\in$,$\notin$,$\forall$,$\exists$).

7.3 Common Mutation Patterns

This Section contains a description of the different mutation patterns which MoMuT produces on OOASs generated by our test case generation toolchain. For each mutation pattern, an examination of its potential to produce equivalent or non-equivalent mutants is given. Generally, all `given` and `when` statements are mapped to guarded commands. A guard falsification ($gd(A) \mapsto \bot$) of a guarded command will always result in the loss of all system traces which include the action A. This may directly or indirectly lead to an **ioco** violation although this is entirely dependent on the individual model contents.

7.3.1 Mutations on Equality Expressions

Both `given` and `when` statements include a term $T \Leftrightarrow variable = value$ when written in their simplest form:

```
given DeviceState = Pause ...
when UserAction = STBY ...
```

Through the mutation $= \mapsto \neq$, all traces in which the term was previously fulfilled are lost. The reverse case is true for MDML statements which are represented as $T \Leftrightarrow variable \neq value$. This is only the case for `given` statements of the following form:

```
given DeviceState != Pause ...
```

Here, the mutation $\neq \mapsto =$ also eliminates all original traces.

7.3.2 Mutations on List-Based Expressions

To illustrate the effect of mutations on list-based expressions, we define a set $\mathbf{E}$ of n equality conditions e_i which are mutually exclusive:

$$\forall e_i, e_j \in \mathbf{E} : (e_i \wedge e_j) \Rightarrow (i = j)$$

This mutual exclusivity had to be enforced during model transformation by eliminating duplicate cases from list-based expressions (see Section 6.2.14). In the OOAS, list inclusion expressions have been translated as disjunctions of equality conditions with each list element:

$$T \Leftrightarrow e_1 \vee \cdots \vee e_i \vee e_{i+1} \vee \cdots \vee e_n \text{ with } e_j \Leftrightarrow variable = value_j$$

The following types of MDML statements are represented in this way:

```
given DeviceState in {Pause, Standby, ... } ...
when UserAction != STBY ...
when UserAction in {SPAU, SMES, ... } ...
when UserAction not in {SPAU, SMES, ... } ...
when UserAction = any ...
```

We will now examine all relevant mutations on the $\vee$ operator between e_i and e_{i+1}, starting wit $\vee \mapsto \wedge$. The modified logical implication operator $\overset{\mathrm{E}}{\Rightarrow}$ implicitly takes the mutual exclusivity of all $e \in \mathbf{E}$ into account:

$$e_1 \vee \cdots \vee e_i \wedge e_{i+1} \vee \cdots \vee e_n \overset{\mathrm{E}}{\Rightarrow} e_1 \vee \cdots \vee e_{i-1} \vee e_{i+2} \vee \cdots \vee e_n$$

The $\wedge$ operator binds stronger than $\vee$. Due to their mutual exclusivity, the expressions e_i and e_{i+1} are effectively eliminated from the term, resulting in the loss of all traces associated with them.

$$e_1 \vee \cdots \vee e_i \rightarrow e_{i+1} \vee \cdots \vee e_n \overset{\mathrm{E}}{\Rightarrow} \neg e_1 \wedge \cdots \wedge \neg e_i$$

The $\rightarrow$ operator binds weaker than $\vee$. This results in the loss of all traces associated with the expressions e_1 to e_i.

$$e_1 \vee \cdots \vee e_i \leftrightarrow e_{i+1} \vee \cdots \vee e_n \overset{\mathrm{E}}{\Rightarrow} \neg e_1 \wedge \cdots \wedge \neg e_n$$

The $\leftrightarrow$ operator binds weaker than $\vee$. In this case, the term is only fulfillable if none of the expressions are true. All previously associated traces are eliminated. Finally, we examine the mutation $= \mapsto \neq$ by representing it as $e_i \mapsto \neg e_i$:

$$e_1 \vee \cdots \vee \neg e_i \vee \cdots \vee e_n \overset{\mathrm{E}}{\Rightarrow} \neg e_i$$

While this mutation adds traces for all cases which were not included in the list, it also removes all traces associated with e_i.

Analogously to the list inclusion ($\in$) case, list exclusion ($\notin$) expressions are represented as negated inclusions:

$$T \Leftrightarrow \neg(e_1 \vee \cdots \vee e_i \vee e_{i+1} \vee \cdots \vee e_n)$$

Of all MDML statements, only list exclusions on state variables are represented in this way:

```
given DeviceState not in {Pause, Standby, ... } ...
```

We will again examine all relevant mutations, starting with $\vee \mapsto \wedge$:

$$\neg(e_1 \vee \cdots \vee e_i \wedge e_{i+1} \vee \cdots \vee e_n) \overset{\mathrm{E}}{\Rightarrow} \neg e_1 \wedge \cdots \wedge \neg e_{i-1} \wedge \neg e_{i+2} \wedge \cdots \wedge \neg e_n$$

Analogous to the list inclusion case, the mutation $\vee \mapsto \wedge$ results in the removal of e_i and e_{i+1} from the term, thereby weakening it. This mutation is equivalent since it preserves all original model traces.

$$\neg(e_1 \vee \cdots \vee e_i \rightarrow e_{i+1} \vee \cdots \vee e_n) \overset{\mathrm{E}}{\Rightarrow} e_1 \vee \cdots \vee e_i$$

The mutation $\vee \mapsto \rightarrow$ forces one of the cases e_1 to e_i. If the list covered all possibilities ($\exists e \in \mathbf{E} : e = \top$) the term would have been unfulfillable in the first place. Generally, this is not the case and all previously allowed traces disappear.

$$\neg(e_1 \vee \cdots \vee e_i \leftrightarrow e_{i+1} \vee \cdots \vee e_n) \overset{\mathrm{E}}{\Rightarrow} e_1 \vee \cdots \vee e_n$$

In combination with $\neg$, the $\leftrightarrow$ operator produces an exclusive disjunction which is equivalent to a negation of the original term. All original traces disappear.

$$\neg(e_1 \vee \cdots \vee \neg e_i \vee \cdots \vee e_n) \overset{\text{E}}{\Rightarrow} e_i$$

The mutation $= \mapsto \neq$ removes all traces associated with values which were not included in the list while forcing the value associated with the mutated expression.

$$\neg\neg(e_1 \vee \cdots \vee e_n) \overset{\text{E}}{\Rightarrow} e_1 \vee \cdots \vee e_n$$

Finally, the mutation$\neg \mapsto \neg\neg$ removes all original traces while adding all traces associated with each value within the list.

7.3.3 Mutations on other OOAS Elements

We conclude our considerations by presenting some less frequently occurring mutation patterns:

- A guard falsification on an observable action prevents the observable from ever occurring. Unless the observable is never shown in the first place, this always results in a non-equivalent mutant.
- A guard falsification within a setter method (see Section 6.2.12) effectively reduces it to a `skip` action. This will prevent the associated state variable from ever being set to a new value and disable the associated changed-observable, resulting in a non-equivalent mutant. The mutation $\neq \mapsto =$ on the guard has a similar effect while also showing a disallowed observable on self-transitions.
- Setter arguments can be mutated[3] in two different ways: changing the enum argument for the reported value results in the emission of a disallowed observable and a non-equivalent mutant. Mutating the enum argument for the actual value merely changes the internal system state and may or may not indirectly cause an **ioco** violation, depending on the model. If the mutated target state allows all transitions which are allowed by the original target state, the mutant is equivalent.
- The guard of a secondary action (see Section 6.2.16) asserts the length of the secondary action queue to be greater than zero. It can be subjected to guard falsification, as well as relational operator replacements. In the first case, as well as with $> \mapsto <$, all traces containing secondary actions disappear. The mutation $> \mapsto \neq$ is always equivalent because the length of the secondary action queue cannot be negative. The mutations $> \mapsto (\leq, =, \geq)$ cause the OOAS to crash because the dequeuing method (see Section 6.2.9) is executed on an empty list.
- Guard falsifications and $= \mapsto \neq$ within the secondary action trigger methods (see Section 6.2.10) prevent the queuing of certain secondary actions at the appropriate time, resulting in the alteration of all traces associated with them. Additionally, $= \mapsto \neq$ causes the queuing of inappropriate secondary actions which may also cause the emission of disallowed observables.

[3]At the time of our experiments, MoMuT only supported the mutation of enum values within variable assignments. Therefore, we added two temporary variables for each state variable. Before referencing each setter method, the arguments were assigned to these variables which were then used as setter arguments. We did not mention this makeshift mechanism in Chapter 6 for brevity reasons.

7.4 Abstract Test Cases

As its output, MoMuT produces a set of *abstract test cases*. They can be thought of as small labelled transition systems, each depicting an individual execution trace of the OOAS. Abstract test cases are encoded as directed graphs in the Aldebaran[4] file format. An example test case, generated from the AVL740 model is given below:

```
des (0, 5, 5)
(0,"ctr _AVL740_root.ctr_PrimaryAction(event=ie_UserAction_STBY);
    obs _AVL740_root.obs_DeviceState_Changed(value=s_DeviceState_Standby);
    obs _AVL740_root.obs_Status_Changed(value=s_Status_Busy) ",1)
(1,"ctr _AVL740_root.ctr_PrimaryAction(event=tt_TimeTriggerImpl15d81201);
    obs _AVL740_root.obs_Status_Changed(value=s_Status_Ready) ",2)
(2,"ctr _AVL740_root.ctr_PrimaryAction(event=ie_UserAction_SMES);
    obs _AVL740_root.obs_DeviceState_Changed(value=s_DeviceState_Measurement) ",3)
(3,"ctr _AVL740_root.ctr_PrimaryAction(event=ie_UserAction_SPAU);
    obs _AVL740_root.obs_DeviceState_Changed(value=s_DeviceState_Pause);
    obs _AVL740_root.obs_Status_Changed(value=s_Status_Busy) ",4)
(4,"pass",4)
```

The first line gives an overview over the size of the encoded graph, specifically, the index of the initial node (0), the overall number of transitions (5) and the overall number of nodes (5). All test cases generated by our toolchain are deterministic and are therefore represented by a linear graph. For non-deterministic OOASs, MoMuT would generate branching test cases. The rest of the file encodes the transitions, each containing the index of their source node, one or more actions within quotation marks and the index of the target node. Actions can be either controllable or observable which are executed in the order of appearance. The last of the transition denotes an explicit “pass” verdict which is reached at the end of the test sequence. A failure of the SUT to produce any of the required observables at the appropriate time is considered a test failure.

[4]http://www.inrialpes.fr/vasy/cadp/man/aut.html

8 Transformation from Abstract to Concrete Test Cases

The previously described abstract test cases lie at the same high abstraction level as the MDML model. Therefore, they must be transformed into *concrete* test cases which can be executed against actual measurement devices. During the TRUFAL project, this task was performed by a small command line tool called the *test case transformator* [sic!] (TCT) [18]. Each abstract test case was parsed by a regular expression grammar and each controllable or observable was replaced by a C# statement from an XML-based template. This XML file was device-specific and had to be created manually in addition to the device model. During the early stages of the TRUCONF project, MoMuT was changed to accommodate cascaded calls of methods and/or actions within object-oriented action systems. As a side-effect, the format of abstract test cases was changed so that several controllables and observables could be concatenated within a single node. This change was incompatible with the original implementation of the test case transformator. Due to these functional and usability-related problems, the algorithm was rewritten from scratch as part of the MDML IDE. Instead of regular expressions, a dedicated Xtext grammar was created to parse the Aldebaran files. Since the internal logic of the TCT was much simpler than that of the code generator, we forwent a rigorous test-driven development practice.

8.1 The Test Automation Framework

The *test automation framework* (TAF) [18] is used by the AVL test center to manage all automated tests for the PUMA Open system. It was created during the TRUFAL project to bundle all test automation code into a single C# source tree. This source tree compiles into a .dll file which can be run within the NUnit[1] unit testing framework. The TAF makes extensive use of *PageObjects*, a concept adopted from the Selenium[2] software testing framework for web applications. In the context of our particular use case, a PageObject encapsulates the device driver used by PUMA Open to connect to a specific measurement device via an Ethernet connection. Figure 8.1 shows a simplified[3] class diagram of the measurement device test fixture as it exists within the TAF. Its object model is separated into two distinct regions, one related to PageObjects, the other related to test suites. Each measurement device PageObject is derived from a common base class called

[1]`http://nunit.org/`

[2]`http://www.seleniumhq.org/`

[3]The class diagram only shows classes, members and methods which are relevant to our immediate use case.

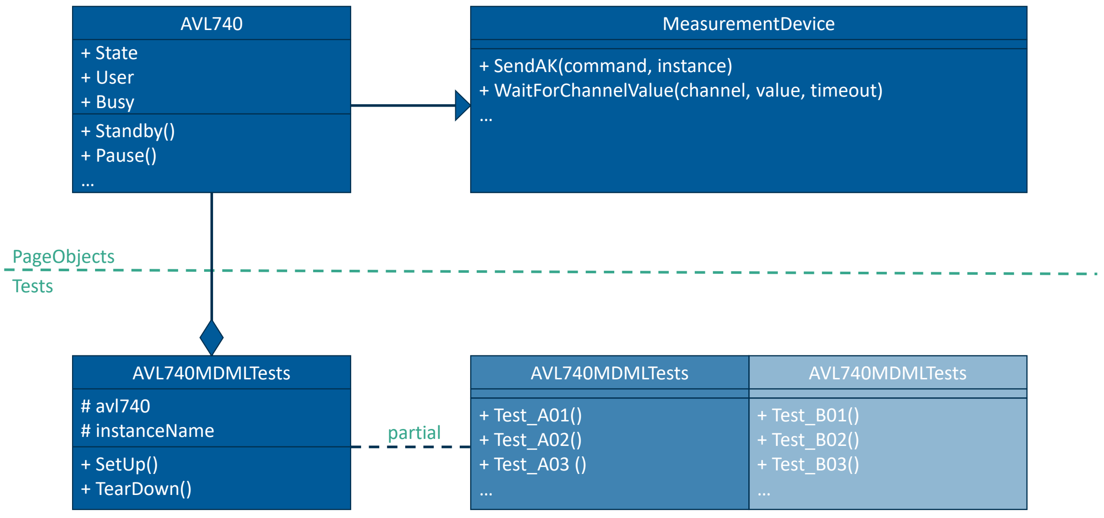


Figure 8.1: A simplified class diagram of the test fixture.

`MeasurementDevice`, which provides rather low-level and device-unspecific functionalities like the transmission of AK commands [57] via the `SendAK` method or the assertion of the value of a system channel within a certain time frame via the `WaitForChannelValue` method. The PageObjects for individual devices provide a series of system channels corresponding to the Operation, Transition and Control state variables, as well as a series of more high-level methods which manage the transitions to different system states, like `Pause` or `Standby`. These methods send a control sequence to the *configurable device handler* (CDH) which in turn sends a series of related AK commands to the device. Note that the naming patterns of PageObject members and methods may differ substantially from the naming conventions used in the specification material. As it was already done during TRUFAL [18], the TRUCONF test suites employ a programming concept called *partial classes* which allows to distribute the definition of a class over different source files. For each measurement device, a test suite base file (Figure 8.1, lower left) must be created. This file contains an instantiation of the measurement device PageObject, as well as the definition of test setup and teardown methods which perform any tasks necessary before and after the execution of a test case (like resetting the SUT to its initial state configuration). While the test suite base file has to be created manually, its declaration as a partial class allows for the incorporation of additional source files, each of which adds a set of automatically generated test methods to the class definition (Figure 8.1, lower right). This architecture allows individual test suites to be seamlessly added or removed without causing compilation problems. However, since arbitrary auto-generated test suites must be compatible with each other in a common TAF build, the collision of file and test method names must be prohibited. Therefore, each file or method name receives a unique number, based on the date and time at which the test suite was generated.

8.2 Transformation Schemes

A test suite can be transformed according to several different *test case transformation schemes* (TCT schemes). Each TCT scheme is a collection of different code templates and transformation rules to synthesize test cases for different purposes. In contrast to the architecture used within TRUFAL, the TCT schemes are not specific to any measurement device, but instead utilize all available regularities within the TAF object model and naming patterns and draw all other necessary information from names and annotations within the MDML model. The MDML IDE currently supports three different transformation schemes, described in detail below.

8.2.1 Concrete Device (AK-Commands)

This transformation scheme makes exclusive use of the methods `WaitForChannelValue` and `SendAK` within the class `MeasurementDevice` and completely bypasses the high-level methods of the dedicated device class[4]. It also has the benefit of allowing the user to leverage the largest amount of information from the DeviceKB. However, of all currently implemented TCT schemes, it requires the most thorough naming and annotation conventions to be compatible with the PageObject. We provide an example of this transformation scheme by applying it to the abstract test case shown in Section 7.4:

```
[Test]
[Category(TestCategory.Priority.Medium)]
public void BFS_test_1712291106_00()
{
  avl740.SendAK("STBY", instanceName);
  avl740.WaitForChannelValue(avl740.DeviceState, AVL740.States.Standby, 2);
  avl740.WaitForChannelValue(avl740.Status, AVL740.TransitionStates.Busy, 2);
  Waiting.ForSeconds(15);
  avl740.WaitForChannelValue(avl740.Status, AVL740.TransitionStates.Ready, 2);
  avl740.SendAK("SMES", instanceName);
  avl740.WaitForChannelValue(avl740.DeviceState, AVL740.States.Measurement, 2);
  avl740.SendAK("SPAU", instanceName);
  avl740.WaitForChannelValue(avl740.DeviceState, AVL740.States.Pause, 2);
  avl740.WaitForChannelValue(avl740.Status, AVL740.TransitionStates.Busy, 2);
}
```

The above example also illustrates the naming conventions required by the transformation scheme: The MDML device name determines the name of the PageObject variable (transformed into lower case). State variables must be named like their corresponding system channels (e.g. `DeviceState`). States must be annotated with their corresponding enum values (e.g. `AVL740.States.Standby`). Input events must be written as 4-letter AK commands (e.g. `STBY`), annotated with additional arguments for the method `SendAK`, if necessary.

[4]This TCT scheme is called "concrete" in contrast to the DAL transformation scheme (see Section 8.2.3).

8.2.2 Concrete Device (Simple)

In addition to the AK command-based transformation scheme described above, we also implemented a *"simple"* transformation scheme which utilizes the high-level controllable methods provided by the concrete device PageObjects. This TCT scheme was added on request from our test engineers who used high-level methods in their hand-crafted test cases. It also requires a far less rigorous naming convention within the MDML model. Input events must be named after their associated method names and annotated with arguments, if necessary.

```
1 [Test]
2 [Category(TestCategory.Priority.Medium)]
3 public void BFS_test_1712291106_00()
4 {
5   avl740.Standby();
6   Waiting.ForSeconds(15);
7   avl740.Measurement();
8   avl740.Pause();
9 }
```

In addition to initiating a state transition, the high-level methods assert both the pre- and post-state of the transition. Also, they block during transient states like *Busy* to ensure that the device is in a stable state upon method termination. This has several detrimental implications for model-based testing: The transformation scheme is unable to test model traces which include the abortion of transient states like *Busy*. Therefore, such traces must be absent from the model if this TCT scheme is to be utilized. Furthermore, the high-level methods do not explicitly assert that transient states ever occur, which makes a full **ioco** check of transient states impossible and should substantially limit our ability to find faults via the coupling effect.

High-level PageObject methods have previously been used in test cases created by the TRUFAL tool chain [18]. At this time, they offered the possibility to disable the pre- and post-state assertions through a boolean argument. However, this functionality has since been removed. Since their function is either fulfilled or impaired by the automatic pre- and post-state assertions within the controllables, all observables present in the abstract test case are hidden by the transformation scheme. Also, note that the `Waiting` method at line 6 of the above example only executes after the transient state has already elapsed. Thus, the waiting time is completely pointless and only unnecessarily increases the execution time of the test case. However, we chose not to hide timed transitions in the TCT scheme since we cannot rule out the possibility that the high-level methods allow for the asynchronous execution of timed behaviours on some occasions. Overall, the *simple* TCT scheme will require a substantially different and more coarse MDML modelling style in order to produce functioning test suites.

8.2.3 Device Abstraction Layer

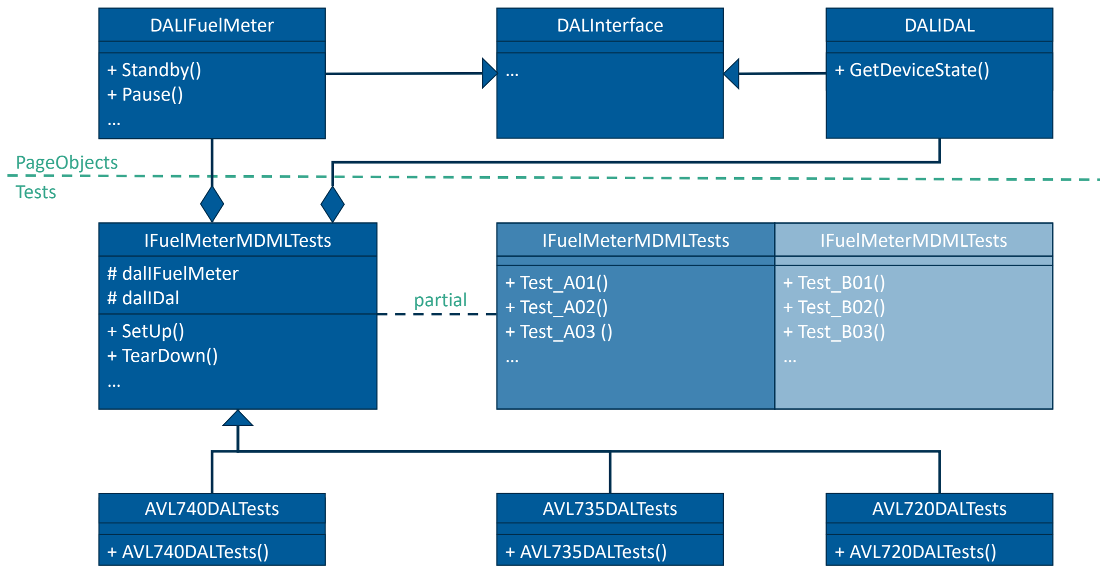


Figure 8.2: A simplified class diagram of the DAL test fixture.

The *device abstraction layer* (DAL) groups similar measurement devices into common device classes, e.g. *FuelMeters*, *SmokeMeters*, *Opacimeters*, *ParticleCounters* or *Conditioning* devices. The devices within a class usually share large parts of their state models. Due to the properties of the **ioco** implementation relation described in Section 2.2.2, the overlapping part of the state spaces can be covered by a common MDML model. The object model of the test fixture, as seen in Figure 8.2, differs noticeably from the test fixture object model for concrete devices which was described in Section 8.1. PageObjects of DAL interfaces are entirely detached from the `MeasurementDevice` class hierarchy and are instead derived from the base class `DALInterface`. Moreover, the state retrieval method `GetDeviceState` is located in a separate interface called `DALIDAL` which is implemented by all measurement devices. While DAL test suites are valid for all devices within a class, a concrete system under test must be specified in order to execute them. In the object model, this SUT specification must be kept separate from the test suite definition to keep the test suites to keep the tests device-independent. Therefore, the test fixture class serves as a base for device-specific derived classes (Figure 8.2, bottom) which instantiate the SUT reference with a concrete measurement device within their constructors.

```
[Test]
[Category(TestCategory.Priority.Medium)]
public void BFS_test_1712291106_00()
{
  dalIFuelMeter.Standby().AssertReturnCodeIs(1);
  dalIDal.GetDeviceState().AssertDeviceStateIs(DeviceState.StandBy);
  Waiting.ForSeconds(15);
  dalIFuelMeter.ContinuousMeasurement().AssertReturnCodeIs(1);
  dalIDal.GetDeviceState().AssertDeviceStateIs(DeviceState.Measurement);
  dalIFuelMeter.Pause().AssertReturnCodeIs(1);
  dalIDal.GetDeviceState().AssertDeviceStateIs(DeviceState.Pause);
}
```

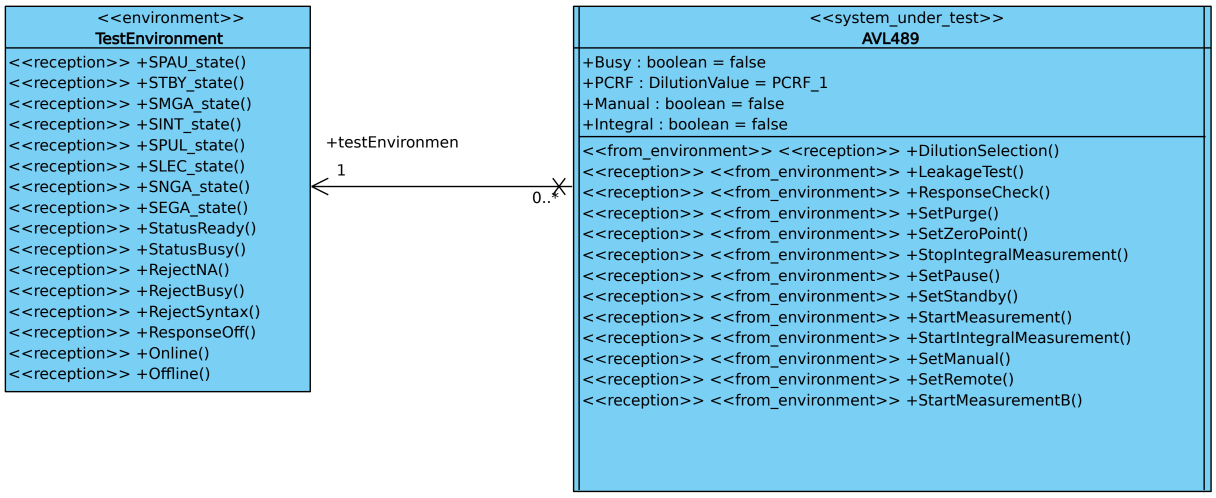
On first glance, the controllable methods like `Standby` or `Pause` strongly resemble the high-level methods of concrete device PageObjects. In fact, they also abstract *Busy* phases of the device which requires a similarly adapted modelling style as the *simple* TCT scheme for concrete devices. Other than the high-level methods, the DAL controllables do not make any assertions about pre- and post-states. They are merely followed up by a method which asserts the absence of device errors during the execution of a command. The DAL transformation scheme requires a moderately extensive naming convention. The Operation state variable must be named *DeviceState* to match the name of the state retrieval method. This naming convention has not been abstracted in case that other state variables of the device should eventually become observable through state retrieval methods. The names of input events must match the controllable method names and they must be annotated with arguments, if necessary. Since the controllable methods hide the Transition state variable, it can be omitted in MDML models. The Control state variable can be influenced through the methods `RequestControl` and `ReleaseControl`, but cannot be observed. Therefore it should be declared `private` within the MDML model.

9 Case Study: AVL489

To validate the TRUCONF test case generation toolchain, we conducted a series of experiments based on an MDML model of the *AVL Particle Counter* (AVL489) [20]. We used this model to generate a series of test suites and executed them against a testbed simulation model of AVL489 which was modified to exhibit a predefined set of faults (subsequently called *SUT mutants*). The experiments were designed to mirror a similar set of trials which were performed at the conclusion of the TRUFAL project and documented in a corresponding paper [6].

9.1 Experiment Setup

9.1.1 AVL489 UML Model



<<environment>>
TestEnvironment
<<reception>> +SPAU_state()
<<reception>> +STBY_state()
<<reception>> +SMGA_state()
<<reception>> +SINT_state()
<<reception>> +SPUL_state()
<<reception>> +SLEC_state()
<<reception>> +SNGA_state()
<<reception>> +SEGA_state()
<<reception>> +StatusReady()
<<reception>> +StatusBusy()
<<reception>> +RejectNA()
<<reception>> +RejectBusy()
<<reception>> +RejectSyntax()
<<reception>> +ResponseOff()
<<reception>> +Online()
<<reception>> +Offline()
+testEnvironmen
1
0..*
<<system_under_test>>
AVL489
+Busy : boolean = false
+PCRF : DilutionValue = PCRF_1
+Manual : boolean = false
+Integral : boolean = false
<<from_environment>> <<reception>> +DilutionSelection()
<<reception>> <<from_environment>> +LeakageTest()
<<reception>> <<from_environment>> +ResponseCheck()
<<reception>> <<from_environment>> +SetPurge()
<<reception>> <<from_environment>> +SetZeroPoint()
<<reception>> <<from_environment>> +StopIntegralMeasurement()
<<reception>> <<from_environment>> +SetPause()
<<reception>> <<from_environment>> +SetStandby()
<<reception>> <<from_environment>> +StartMeasurement()
<<reception>> <<from_environment>> +StartIntegralMeasurement()
<<reception>> <<from_environment>> +SetManual()
<<reception>> <<from_environment>> +SetRemote()
<<reception>> <<from_environment>> +StartMeasurementB()

Figure 9.1: Final UML class diagram representing the test interface of AVL489, created during TRUFAL.

To maximize the comparability of our results to those of the TRUFAL case study, we used the final UML model of AVL489 as a reference to develop the MDML model. Figure 9.1 depicts the test interface of the UML model. It shows that, in addition to observables reflecting the system state (e.g. *SPAU_state*, *StatusReady* or *Offline*), the test environment can also receive additional signals like *RejectBusy*. While the TRUFAL toolchain was able to handle incomplete models due to its usage of the **ioco** relation [6], this property was not harnessed during evaluation. Instead, an input-complete model was used which rejected disallowed inputs with an appropriate error observable. These rejections have

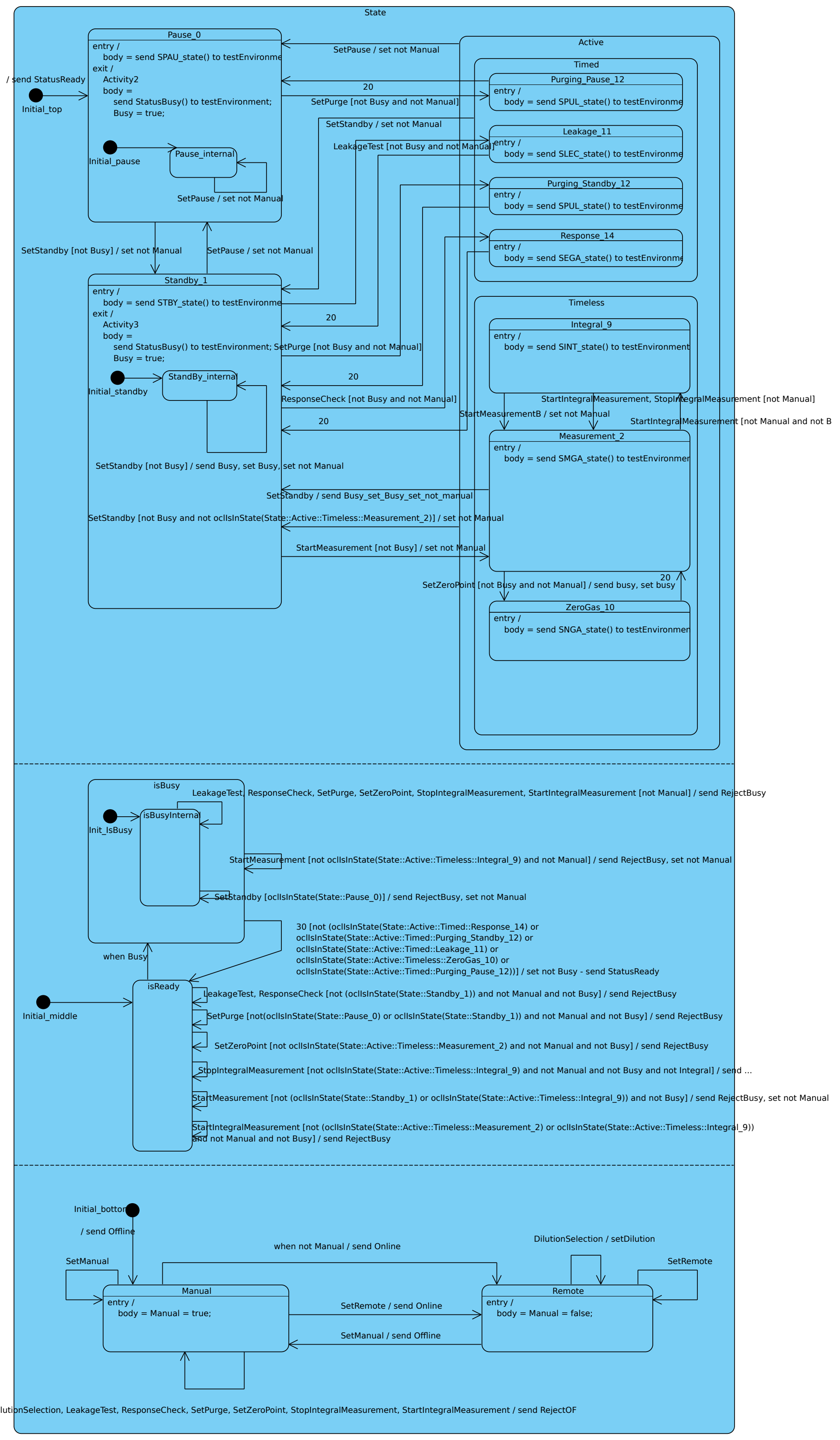

State
Pause_0
entry /
body = send SPAU_state() to testEnvironme
exit /
Activity2
body =
send StatusBusy() to testEnvironment;
Busy = true;
/ send StatusReady
Initial_top
Initial_pause
Pause_internal
SetPause / set not Manual
SetPause / set not Manual
20
SetPurge [not Busy and not Manual]
SetStandby / set not Manual
LeakageTest [not Busy and not Manual]
Active
Timed
Purging_Pause_12
entry /
body = send SPUL_state() to testEnvironme
Leakage_11
entry /
body = send SLEC_state() to testEnvironme
Purging_Standby_12
entry /
body = send SPUL_state() to testEnvironme
Response_14
entry /
body = send SEGA_state() to testEnvironme
SetStandby [not Busy] / set not Manual
SetPause / set not Manual
Standby_1
entry /
body = send STBY_state() to testEnvironme
exit /
Activity3
body =
send StatusBusy() to testEnvironment; SetPurge [not Busy and not Manual]
Busy = true;
20
20
Initial_standby
StandBy_internal
ResponseCheck [not Busy and not Manual]
20
Timeless
Integral_9
entry /
body = send SINT_state() to testEnvironment
StartIntegralMeasurement, StopIntegralMeasurement [not Manual]
StartMeasurementB / set not Manual
StartIntegralMeasurement [not Manual and not B
Measurement_2
entry /
body = send SMGA_state() to testEnvironmer
SetStandby [not Busy] / send Busy, set Busy, set not Manual
SetStandby / send Busy_set_Busy_set_not_manual
SetStandby [not Busy and not oclIsInState(State::Active::Timeless::Measurement_2)] / set not Manual
StartMeasurement [not Busy] / set not Manual
20
SetZeroPoint [not Busy and not Manual] / send busy, set busy
ZeroGas_10
entry /
body = send SNGA_state() to testEnvironmer
isBusy
LeakageTest, ResponseCheck, SetPurge, SetZeroPoint, StopIntegralMeasurement, StartIntegralMeasurement [not Manual] / send RejectBusy
Init_IsBusy
isBusyInternal
StartMeasurement [not oclIsInState(State::Active::Timeless::Integral_9) and not Manual] / send RejectBusy, set not Manual
SetStandby [oclIsInState(State::Pause_0)] / send RejectBusy, set not Manual
30 [not (oclIsInState(State::Active::Timed::Response_14) or
oclIsInState(State::Active::Timed::Purging_Standby_12) or
oclIsInState(State::Active::Timed::Leakage_11) or
oclIsInState(State::Active::Timeless::ZeroGas_10) or
oclIsInState(State::Active::Timed::Purging_Pause_12))] / set not Busy - send StatusReady
when Busy
isReady
Initial_middle
LeakageTest, ResponseCheck [not (oclIsInState(State::Standby_1)) and not Manual and not Busy] / send RejectBusy
SetPurge [not(oclIsInState(State::Pause_0) or oclIsInState(State::Standby_1)) and not Manual and not Busy] / send RejectBusy
SetZeroPoint [not oclIsInState(State::Active::Timeless::Measurement_2) and not Manual and not Busy] / send RejectBusy
StopIntegralMeasurement [not oclIsInState(State::Active::Timeless::Integral_9) and not Manual and not Busy and not Integral] / send ...
StartMeasurement [not (oclIsInState(State::Standby_1) or oclIsInState(State::Active::Timeless::Integral_9)) and not Busy] / send RejectBusy, set not Manual
StartIntegralMeasurement [not (oclIsInState(State::Active::Timeless::Measurement_2) or oclIsInState(State::Active::Timeless::Integral_9))
and not Manual and not Busy] / send RejectBusy
Initial_bottom
/ send Offline
SetManual
when not Manual / send Online
DilutionSelection / setDilution
SetRemote
Manual
entry /
body = Manual = true;
SetRemote / send Online
SetManual / send Offline
Remote
entry /
body = Manual = false;
DilutionSelection, LeakageTest, ResponseCheck, SetPurge, SetZeroPoint, StopIntegralMeasurement, StartIntegralMeasurement / send RejectOF

Figure 9.2: Final UML state machine diagram of AVL489, created during TRUFAL.

been modelled as self-transitions in the Transition and Control region of the state machine diagram (see central and bottom region in Figure 9.2).

9.1.2 AVL489 MDML Model

The MDML model of AVL489 was created based on the information present in the UML model. A major difference to the UML model is that all functionality related to error handling has been omitted. During the development of MDML, the refusal of erroneous commands had a low priority as a test target, compared to other functionalities. The ineffectiveness of input actions could be modelled by the `DoNothing` command. This, however, would only pose an indirect solution since the model transformation does not support the explicit observability of unchanged state variables[1]. Furthermore, since the ability to employ partial device models for test case generation is a key requirement to MDML (see Section 3.3, req. 7), we took the chance of examining a model with partially unspecified behaviour[2]. The MDML model of AVL489 used in this case study is given below in its full extent:

[1]This shortcoming has been resolved over the course of the experiments. See test suite **F** (Section 9.4) for more information.

[2]While the model size was somewhat decreased by this measure, the change is insufficient to explain the significant reduction in test case generation time observed throughout the case study.

```
device AVL489 {
  public statevar DeviceState {
    Pause_0, Standby_1, Measurement_2, Integral_9, ZeroGas_10, Leakage_11,
    Purging_12, Response_14
  } = Pause_0;
  public statevar Status {Ready, Busy} = Ready;
  public statevar UserLevel {Manual, Remote} = Remote;
  input UserAction {
    SetDilution(1), LeakageTest, ResponseCheck, Purge, ZeroPoint,
        StopIntegralMeasurement, Pause, Standby, StartMeasurement,
        StartIntegralMeasurement, Manual, Remote
  };

  // OPERATING STATE *************************************************************
  given Status = Ready {
    given UserLevel = Remote {
      given DeviceState = Pause_0 {
        when UserAction = Purge then DeviceState -> Purging_12 and Status -> Busy;
      }
      given DeviceState = Standby_1 {
        when UserAction = LeakageTest then DeviceState -> Leakage_11
                  and Status -> Busy;
        when UserAction = Purge then DeviceState -> Purging_12 and Status -> Busy;
        when UserAction = ResponseCheck then DeviceState -> Response_14
          and Status -> Busy;
      }
      given DeviceState = Measurement_2 {
        when UserAction = StartIntegralMeasurement then DeviceState -> Integral_9;
        when UserAction = ZeroPoint then DeviceState -> ZeroGas_10
                  and Status -> Busy;
      }
    }

    // Only Ready
    given DeviceState = Pause_0 {
      when UserAction = Standby then DeviceState -> Standby_1 and Status -> Busy
        and UserLevel -> Remote;
    }
    given DeviceState = Standby_1 {
      when UserAction = Standby then Status -> Busy and UserLevel -> Remote;
      when UserAction = StartMeasurement then DeviceState -> Measurement_2
        and Status -> Busy and UserLevel -> Remote;
    }
    // Compound returns
    given DeviceState in {Integral_9, ZeroGas_10}{
      when UserAction = Standby then DeviceState -> Standby_1
            and UserLevel -> Remote;
    }
  }

  // Always possible
  given DeviceState = Pause_0 {
    when UserAction = Pause then UserLevel -> Remote;
  }
  given DeviceState = Standby_1 {
    when UserAction = Pause then DeviceState -> Pause_0 and Status -> Busy
      and UserLevel -> Remote;
  }
  given DeviceState = Measurement_2 {
    // Compound returns
    when UserAction = Standby then DeviceState -> Standby_1 and Status -> Busy
      and UserLevel -> Remote;
  }
  given DeviceState = Integral_9 {
    given UserLevel = Remote when UserAction in {
      StartIntegralMeasurement, StopIntegralMeasurement
    } then DeviceState -> Measurement_2;
  }

  // Compound returns
  given DeviceState in {
    Measurement_2, Integral_9, ZeroGas_10, Leakage_11, Purging_12, Response_14
  } when UserAction = Pause then DeviceState -> Pause_0 and UserLevel -> Remote;
  given DeviceState in {Leakage_11, Purging_12, Response_14}
    when UserAction = Standby then DeviceState -> Standby_1 and UserLevel -> Remote;

  given DeviceState = ZeroGas_10 when 20 sec elapsed
    then DeviceState -> Measurement_2;
  given DeviceState = Leakage_11 when 20 sec elapsed then DeviceState -> Standby_1;
  given DeviceState = Purging_12 when 20 sec elapsed then DeviceState -> last;
  given DeviceState = Response_14 when 20 sec elapsed then DeviceState -> Standby_1;

  // TRANSITION STATE ************************************************************
  given Status = Busy {
    given DeviceState not in {ZeroGas_10, Leakage_11, Purging_12, Response_14} {
      when 30 sec elapsed then Status -> Ready;
    }
  }

  // CONTROL STATE ***************************************************************
  when UserAction = Manual then UserLevel -> Manual;
  when UserAction = Remote then UserLevel -> Remote;
  given UserLevel = Remote when UserAction = SetDilution then DoNothing;

}
```

While the above model was built to support the *simple* TCT scheme for concrete devices, several test suites within the case study required the AK-based TCT scheme. Therefore, the names of several state variables, states and input events had to be altered and/or annotated to match naming conventions present in the TAF source code. The header section of the AK-based model variant is given below. Apart from the changes mentioned above, it is identical to the one supporting the *simple* transformation scheme.

```
device AVL489 {
  public statevar State {
    Pause(AVL489.States.Pause),
    Standby(AVL489.States.Standby),
    Measurement(AVL489.States.Measurement),
    Integral(AVL489.States.IntegralMeasurement),
    ZeroGas(AVL489.States.ZeroPoint),
    Leakage(AVL489.States.Leakage),
    Purging(AVL489.States.Purging),
    Response(AVL489.States.Response)
  } = Pause;
  public statevar Busy {False, True} = False;
  public statevar User {
    Manual(MeasurementDevice.UserLevel.Manual),
    Remote(MeasurementDevice.UserLevel.Remote)
  } = Remote;
  input UserAction {
    EKGA(1), SLEC(1), SEGA, SPUL, SNGA, SINA,
    SPAU, STBY, SMGA, SINT, SMAN, SREM
  };
  ...
```

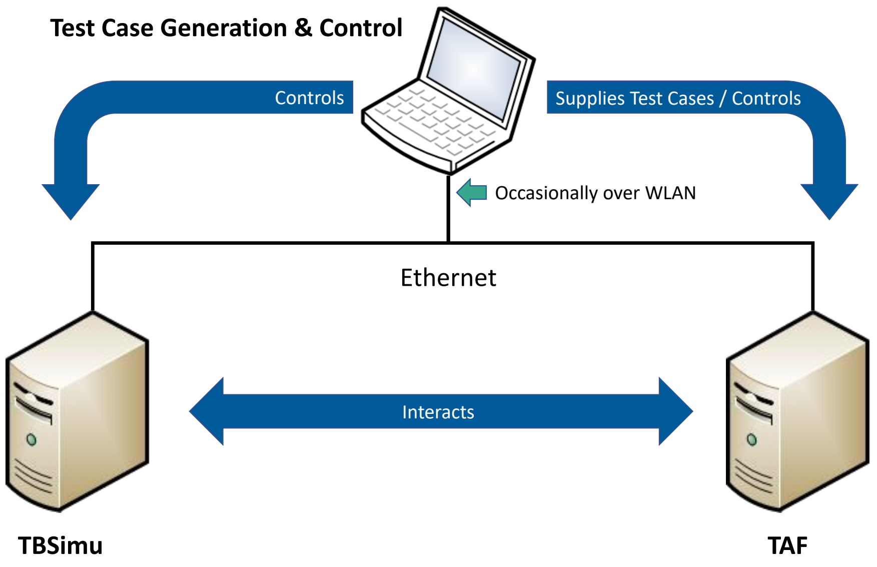

Test Case Generation & Control
Controls
Supplies Test Cases / Controls
Occasionally over WLAN
Ethernet
Interacts
TBSimu
TAF

Figure 9.3: System architecture of the experiment set-up.

9.1.3 Test Case Generator

In the time before and during the execution of this case study, the MoMuT test case generator was undergoing substantial modifications. To give us access to its newest features like the breadth-first search strategy, the developers provided us with a development snapshot of the tool which had not yet undergone the appropriate quality assurance practices. Furthermore, we were informed that the mutation-based and combined mutation/random search strategies employed during the TRUFAL evaluation phase [6] would be discontinued. Despite it being currently under development, we chose the breadth-first search (BFS) strategy for all our experiments because it was able to find all reachable mutants within our model.

9.1.4 System Architecture

The experiment setup was made up of three individual computers:

1. One computer was used to compile and run the test suites within the test automation framework. The compiled TAF projects are executed within NUnit which causes an instance of PUMA Open 2.0 to start and to connect to all configured measurement devices which are reachable over Ethernet.
2. One computer was used to run a testbed simulation (TBSimu) model of the AVL489 measurement device. Over an Ethernet connection, this simulator is indistinguishable from a real *AVL Particle Counter*. Such simulation models are used by the AVL test center as an inexpensive way to easily recreate different system configurations without the need to have the actual devices present [6, 18].

3. One computer was used to run the test case generation toolchain and to control the other two machines by means of virtual network computing (VNC).

The physical experiment setup can be seen in its entirety in Figure 9.3. While the system specifications of the TBSimu and TAF machines are comparable (or identical) to those of the TRUFAL evaluation, we used a standard-issue laptop for test case generation, as opposed to a high-end computer, as it was used within TRUFAL. A comparison of their hardware properties is given in Table 9.1.

	TRUFAL	**TRUCONF**
Processor Model	2×6-core Intel Xeon	2-core Intel Core i5-6300U
Processor Frequency	3.47 GHz	2.40 GHz
Logical Cores	24	4
Memory	190 GB	8 GB
Operating System	64 bit Debian 7.1	64 bit Windows 10

Table 9.1: Hardware comparison of the computers used for test case generation within the TRUFAL [6, 56] and TRUCONF projects.

9.1.5 System Under Test

ID	**Description**
1	Operation *SetManual* disabled in state *Measurement*
2	Operation *SetManual* disabled in state *IntegralMeasurement*
3	Operation *SetManual* disabled in state *Purging*
4	Device will not become *Busy* when changing to state *Pause*
5	Device will not become *Busy* when changing to state *Standby*
6	Device will not become *Busy* when changing to state *Leakage*
7	Operation *SetRemote* disabled in state *ZeroGas*
8	Operation *SetRemote* disabled in state *Purging*
9	Operation *SetRemote* disabled in state *Leakage*
10	Duration the device stays *Busy* divided in half
11	Duration the device stays *Busy* doubled
12	Operation *StartMeasurement* disabled
13	Operation *StartIntegralMeasurement* disabled
14	Operation *SetPurge* disabled
15	Operation *ZeroGas* disabled
16	Device becomes *Busy* after *SetPause* in state *Pause*

Table 9.2: The faults (SUT mutants) which can be simulated by the modified testbed simulation model of AVL489 [6, p.13].

As our system under test, we used a testbed simulation model of AVL489 which had been modified to simulate one of 16 different faults, called SUT mutants. A description of all SUT mutants is given in Table 9.2. As it has been indicated in the table, the mutants can be roughly separated into 6 different groups, each of which share a similar failure mode.

During the rest of this case study, we will refer to the SUT mutants as M_1 to M_{16}, as well as M_0 for the unchanged SUT. The modified simulation model was created to mirror the simulation model which was used during the evaluation phase of the TRUFAL project. However, our SUT had to be recreated from scratch because the original simulation model was not available any more. Therefore we cannot guarantee that details which are not specified by Table 9.2 are consistent with the original (most prominently, the duration of the additional *Busy* phase of M_{16}).

9.2 Test Suite S (Shallow)

9.2.1 Generation

Setup

For an initial test run, we generated a test suite **S** utilizing the *simple* TCT scheme for concrete devices. As described in Section 8.2.2, this transformation scheme utilizes high-level PageObject methods. Since these methods do not explicitly assert that transient states like *Busy* ever occurred, **S** is effectively blind to faults associated with them. Therefore, we expected this test suite to exhibit only a limited coverage of SUT mutants.

Generation Run

Due to the recent improvements made to MoMuT, the test case generation was completed in a mere 3 seconds on standard-issue hardware. This is an immense improvement over the performance of the TRUFAL toolchain which took a minimum of 44 hours to generate a mutation-based test suite on a high-end system. The speed-up of the test case generation is likely due to the combination of the rebuilt MoMuT back-end with an OOAS architecture that relies less on parametrized actions and therefore possesses a smaller parameter space than that which was used during the TRUFAL project. Since test case generation times of this magnitude are negligible for our industrial application, we will not consider them any further in this case study.

Model-Based Coverage Analysis

We chose MoMuT's breadth-first exploration strategy for this and all subsequent test case generation runs because it left no reachable model mutant undiscovered. Moreover, MoMuT discovered one case of unreachable model content: the only mutant classified as "not found" only took effect in the state configuration *ZeroGas/Ready*, which could not be reached by the model. This highlights a model-internal inconsistency. **S** reached an overall mutation score of 60.37%. However, while reaching almost all branching points, MoMuT was unable to kill a noticeable amount of weakly non-equivalent mutants. This is most likely due to the functionality of the BFS search strategy which stops after all reachable state configurations

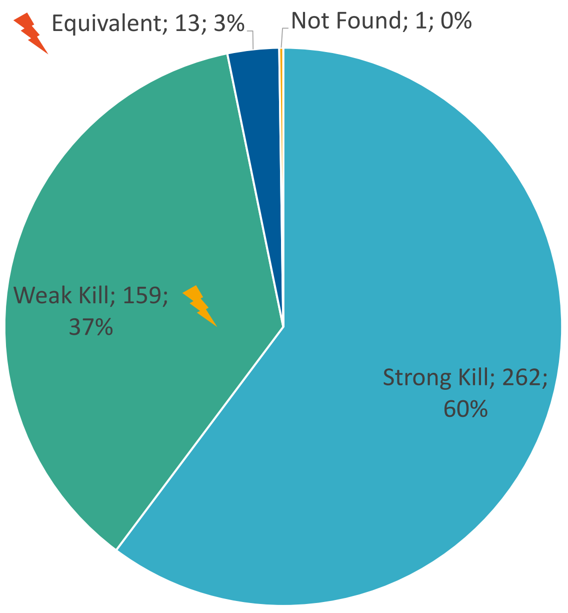


Figure 9.4: Mutation coverage results for Test Suite **S**. Results for Test Suite **D** are identical.

are visited. According to the MoMuT developers, the algorithm currently does not explore any further if a mutant gets activated shortly before this point. This is a known limitation and will be corrected in an upcoming release.

The live mutants include all mutations of the `actual_value` argument of the *DeviceState* setter, which cause the model to internally perform a different state transition than the one reported to the environment. They make up 75% of the live mutants while all corresponding mutants of *DeviceState*'s `reported_value` have been killed. 13 mutants were classified to be weakly equivalent. However, while three of them match our equivalence considerations (see Section 7.3), closer investigation has shown the remaining 10 to be wrongly classified. The problem affected the MDML code segment from line 71 to 75 in the MDML model (see Section 9.1.2), labelled "Compound returns". Here, MoMuT was unable to identify missing traces from *Measurement*, *Integral*, *ZeroGas* or *Leakage* to *Pause* and from *Leakage* to *Standby*. As a direct consequence of the search depth being limited by state configuration coverage, all generated tests had a depth of 6 or lower. Our test engineers found this to be beneficial since shorter test sequences are easier to debug. The results of the coverage analysis are depicted in Figure 9.4. The lightning symbols denote the unusually high number of weak kills produced by a yet imperfect algorithm as well as a definite bug regarding several mutants falsely labelled as equivalent.

9.2.2 Execution

SUT-Based Coverage Analysis

The fault-free system M_0 passes[3] all test cases of **S** but one. This test case tries to verify that the SUT can change from *IntegralMeasurement* back to *Measurement* through the repeated calling of the method `StartIntegralMeasurement`. This is bound to fail since the method asserts the state *Measurement* as its precondition. This example highlights how the *simple* TCT scheme is unable to represent certain model features. As expected, **S** uncovered only about a third of the SUT mutants. Out of M_1-M_3, **S** was only able to kill M_2. While not impossible in principle, **S** does not explore the other two paths. Similarly, **S** also misses M_7-M_9[4] and M_{16}. The mutants M_4-M_6 and M_{10}, on the other hand, could not be killed because the high-level methods do not recognize the absence or shortening of transient states. **S** was unable to distinguish M_0 from M_{11} unless the built-in time-out of the PageObject methods was reduced from 45 sec to 3 sec by way of trial. M_{12}-M_{15} are very coarse mutants and could be relatively easily killed, despite the aforementioned limitations of the test suite.

Causes fore Limited Effectiveness

The cause for the limited effectiveness of **S** may be a combination of several factors: First, as stated above, the *simple* transformation scheme for concrete devices is relatively insensitive to transient states. Secondly, the decision tree within the OOAS is structurally identical to the one in MDML. An efficiently structured decision tree contributes to the readability of the MDML model but has a comparatively low potential for mutations. This should lead to fewer, more coarse mutants and, subsequently, more granular tests. This phenomenon, described by Schlick et al. [88] as "*factoring*", constitutes a violation of the *separation of concerns* paradigm by introducing a conflict between the efficient representation of model features and the quality of the generated test suite. Since the limitation caused by the high number of weak kills also applies to the other test suites in this case study, it cannot be stated as a cause for the relative ineffectiveness of **S**.

Other Observations

Furthermore, we observed the occasional and irreproducible failure of high-level page object methods. These high-level methods execute CDH sequences which, in turn, cause the device driver to send a series of consecutive AK commands to the SUT. Our test engineers hypothesized that these AK commands are sent in such rapid succession that the SUT occasionally fails to process them. If correct, this would be unavoidable without altering the device driver. This hypothesis would also entail that these irreproducible failures should not appear in test suites generated with the AK transformation scheme. For the sake of brevity, we will henceforth refer to this failure mode as *CDH hiccup*.

[3]All tests but one have been observed to pass at least once. See "Other Observations".

[4]The Mutants M_7 and M_9 could not be killed by any TRUFAL or TRUCONF test suite.

9.3 Test Suite D (Deep)

9.3.1 Generation

As a next step, we created a second test suite **D**, which utilized the AK-based TCT scheme. Although we specified the same MoMuT input parameters as for test suite **S**, the generation run yielded one more test case than the previous run. Since MoMuT's search algorithm is supposed to be deterministic, we relayed this observation to the development team for further investigation. The model coverage of **D** is identical to the one of test suite **S**.

9.3.2 Execution

Tuning of Time-Out Values

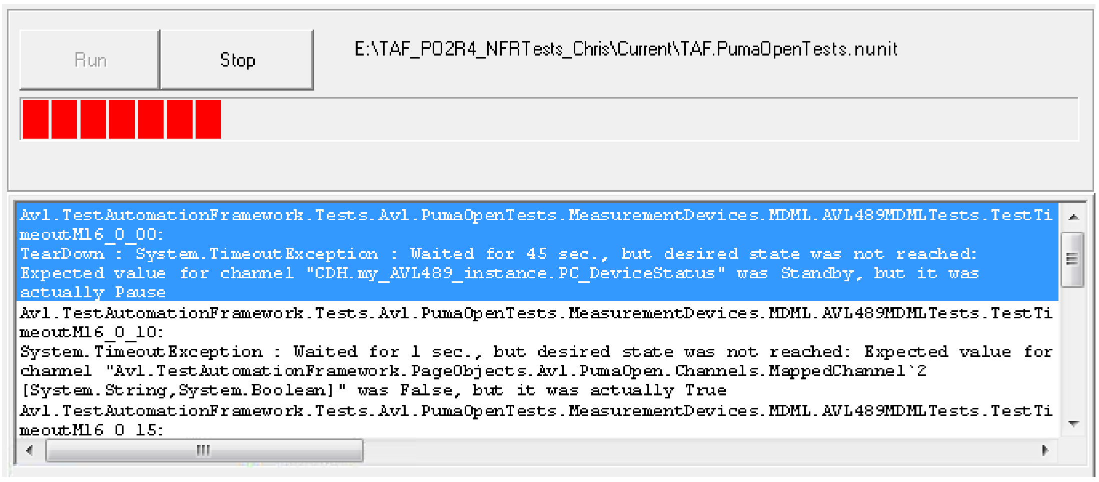
While wrong time-outs in the model were masked by the insensitivity of test suite **S**, they significantly affected the execution result of test suite **D**. Therefore, we iteratively reduced the time-outs within the model and the `WaitForChannelValue` methods to the minimal value which allowed **D** to pass on M_0. This measure was dictated by our lack of information about representative time-outs. Ideally, the correct time-out values should be known beforehand, e.g. from a specification document. However, our experiments show that **D** can uncover time-out violations if the correct values are used. We also retroactively tuned the time-outs of test suite **S** to obtain a representative value of its execution time.

SUT-Based Coverage Analysis

Of M_1-M_3, **D** missed M_3. In contrast to **S**, **D** killed M_5 and M_6. While M_5 was killed by 17 test cases, the test suite missed M_4. Mutants M_7-M_9 were missed because they required the SUT to change from state configuration $X/Y/Remote$ to $X/Y/Manual$ and would then refuse the transition back to $X/Y/Remote$. These traces are beyond the exploration scope of the breadth-first search strategy which stops after every reachable state configuration has been visited once. M_{10} and M_{11} have been killed after the time-outs of **D** had been tuned to appropriate values. The test suite killed all of the coarse mutants M_{12}-M_{15}. One test performed the state transition needed to reach M_{16}. However, due to the limited state surveillance, it did not assert the state of the *Busy* state variable, which was assumed to be unchanged.

Instances of Non-Conformance

As opposed to **S**, **D** uncovered 3 cases of non-conformance between the model and the SUT. As it turned out, the states *ZeroGas*, *Leakage* and *ResponseCheck* in fact do not revert back to their predecessors within 20 seconds.



```
E:\TAF_PO2R4_NFRTests_Chris\Current\TAF.PumaOpenTests.nunit
```

```
Avl.TestAutomationFramework.Tests.Avl.PumaOpenTests.MeasurementDevices.MDML.AVL489MDMLTests.TestTi
meoutM16_0_00:
TearDown : System.TimeoutException : Waited for 45 sec., but desired state was not reached:
Expected value for channel "CDH.my_AVL489_instance.PC_DeviceStatus" was Standby, but it was
actually Pause
Avl.TestAutomationFramework.Tests.Avl.PumaOpenTests.MeasurementDevices.MDML.AVL489MDMLTests.TestTi
meoutM16_0_10:
System.TimeoutException : Waited for 1 sec., but desired state was not reached: Expected value for
channel "Avl.TestAutomationFramework.PageObjects.Avl.PumaOpen.Channels.MappedChannel`2
[System.String,System.Boolean]" was False, but it was actually True
Avl.TestAutomationFramework.Tests.Avl.PumaOpenTests.MeasurementDevices.MDML.AVL489MDMLTests.TestTi
meoutM16_0_15:
```

Figure 9.5: A *teardown hiccup*, as shown on the user interface of NUnit.

Superfluous Test Cases

As a side effect of the limited state surveillance, **D** contained 3 test cases which only consisted of a call to `SendAK` and contained no assertions due to executing a command which did not change the state configuration. These test cases add no value to the test suite. This observation calls the design decision of limited state surveillance into question.

Other Observations

As expected, we did not observe any CDH hiccups within the test methods. However, we still observed their occurrence in the test teardown method which still utilizes high-level page object methods. This observation strengthens our belief that CDH hiccups are indeed caused by AK commands being sent in too rapid succession. We also occasionally observed reproducible assertion failures during test teardown. However, since a test case has already been passed at the time of teardown, we do not consider these assertion failures to be test failures. While they currently have to be manually identified when reviewing the execution results, they are also relatively easy to spot on the user interface of NUnit (see Figure 9.5). We will henceforth refer to all failures occurring during test teardown as *teardown hiccups*.

9.4 Test Suite F (Full)

Within the TRUFAL project, the idea to poll all state variables of the SUT had been abandoned due to performance reasons [6]. This measure prevented a full **ioco** check and limited the ability of the test suite to uncover faults via the coupling effect. However, based on our experiments with test suite **D**, we decided to create a test suite which polls all public state variables of the SUT after every input event and timed behaviour, thereby increasing the impact of the coupling effect. Another aim of this measure was to prevent tests without asserts, like they have occurred in test suite **D**. To accomplish this, we had to modify the code generator in the least invasive way possible, in order to save valuable laboratory time.

9.4.1 Changing the Code Generator

As a first step, all changed-observables in the OOAS needed to be disabled. The least invasive way to do this was to change them to internal actions, thereby rendering them inert.

$$\forall v \in \mathit{StateVariables}_D,\ \mathit{public}(v)\ :\ \mathit{observable}(\mathit{Obs}_C(v)) := \bot$$

```
actions
  ...
  obs_DeviceState_Changed(value : t_DeviceState) = requires true :
    skip
  end ;
  ...
```

Secondly, an observable action had to be created, which communicates the values of all public state variables to the environment.

$$\mathbf{M} \mapsto \mathbf{M} \cup \{\langle D\rangle^{OF}\}$$
$$D \xrightarrow{\text{OF}} \texttt{obs\_FullState}\ (\mathbf{P}^F_1\ ,\ldots,\ \mathbf{P}^F_n)\ \texttt{requires true : skip end}$$
$$\mathbf{P}^F = \{\langle v\rangle^{OFA} \mid v \in \mathit{StateVariables}_D,\ \mathit{public}(v)\}$$
$$v \xrightarrow{\text{OFA}} \langle v\rangle^{ArgName}\ :\ \langle v\rangle^{TName}$$

```
actions
  ...
  obs obs_FullState(v_DeviceState_arg : t_DeviceState , v_Busy_arg : t_Busy)
  = requires true :
    skip
  end
```

This observable action needs to be triggered after every traversal of the decision tree. The body of the action *treeAction* is modified in order to trigger the observable.

$$\mathit{Body}_{\mathit{Int}_{TA}} := \texttt{tree(event) ; obs\_FullState}\ (\mathbf{V}^P_1\ ,\ldots,\ \mathbf{V}^P_n)$$
$$\mathbf{V}^P = \{\langle v\rangle^{Name} \mid v \in \mathit{StateVariables}_D,\ \mathit{public}(v)\}$$

```
actions
  ...
  treeAction(event : e_Event) = requires true :
    tree(event) ;
    obs_FullState(v_DeviceState , v_Busy)
  end
...
```

All of the above changes were implemented as a single strategy which was scheduled to execute at the very end of the model transformation workflow. This measure leaves all previously defined strategies intact and makes the modification easily reversible. Note that this modification would report the full system state in-between secondary actions. Since secondary actions are supposed to represent internal transitions, this behaviour has to be corrected. However, due to our limited laboratory time and the fact that our model of AVL489 does not include secondary actions, we limit the modifications to the model transformation to the above mentioned actions for the time being. It is also worth noting that this measure in its current form negates the improvement in mutant granularity of distinguishing reported and actual state changes. This shortcoming must be addressed in the final implementation.

9.4.2 Generation

Upon generation, test suite **F** yielded 1 additional mutant and 19 additional kills. This resulted in a slightly higher mutation score of 64.6%. The modification produced no equivalent or unreachable mutants. However, according to the MoMuT developers, the increased number of kills is most likely due to a bug present in the development snapshot of MoMuT, which caused the non-appearance of observables to be counted as an empty set of output labels rather than the δ label. Therefore, MoMuT did not register the mutation as an **ioco** violation. This behaviour was masked by the changed OOAS architecture which causes an observable to occur invariably after every interaction with the environment. Overall, the test case generation process yielded 29 test cases. The results of the coverage analysis are depicted in Figure 9.6.

9.4.3 Execution

Execution Time

Despite the additional polling commands, the 29 test cases of **F** took 16 minutes to execute on the original SUT. The mean execution time per test case does not significantly differ from that of test suite **D**. We believe that improvements on the PUMA Open system made after the TRUFAL project have solved the performance issues that caused the need for an incomplete state surveillance.

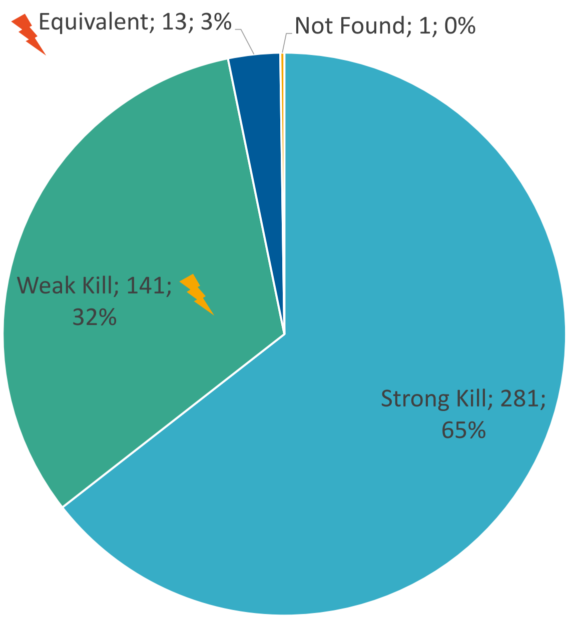


Figure 9.6: Mutation coverage results for test suite **F**.

SUT-Based Coverage Analysis

With 12 out of 16 killed SUT mutants, test suite **F** improves on the coverage of **D** as expected. M_7-M_9 were missed due to reasons detailed in Section 9.3.2. M_{16} was visited but the test case failed to recognize the short additional *Busy* phase when performing a self-transition on the state *Pause*. Based on further experimentation, we believe that this is due to the method `WaitForChannelValue` being too insensitive to detect transient states which are shorter than 2 seconds. Notably, this SUT mutant has been uncovered by TRUFAL test suites. Since we are not aware of any changes in the PageObject or the underlying device driver, it is possible that the former simulator model implemented a longer *Busy* phase. A more thorough observation of this issue on device driver level was impossible due to limited laboratory time.

Instances of Non-Conformance

Test suite **F** uncovered five instances of non-conformance between the model and the SUT, including the three instances discovered by **D**. Additionally, it was revealed that the transition from *IntegralMeasurement* to *Standby* causes the SUT to change to *Busy*, while the transition from *Standby* to *Pause* does not.

9.5 Results

Figures	**Test Suites**					
	S	**D**	**F**	**M**	**R**	**C**
TCT Scheme	Simple	AK	AK	-	-	-
Mutants	435	435	436	3103	3103	3103
Killed	262	262	281	2551	2173	2420
Weakly Killed	159	159	141	-	-	-
Weakly Equivalent	13	13	13	-	-	-
Not Found	1	1	1	-	-	-
Mutation Score	60.37%	60.37%	64.6%	82.21%	70.03%	77.99%
Tests	25	24	29	67	238	57
Maximum Depth	6	6	6	19	25	20
Generation Time	3 sec	5 sec	5 sec	44:09	01:44	67:35
Execution Time	(*00:24*) 00:15	(*00:22*) 00:12	00:15	00:29	01:36	00:29
Mean Ex. Time / TC	36 sec	30 sec	32 sec	26 sec	24 sec	30 sec
SUT Mutant Kills	5	10	12	12	10	13

Table 9.3: Generation and execution details of the generated test suites (S, D and F) in comparison with the test suites from TRUFAL (M, R and C) [6]. Times are given in hh:mm unless specified otherwise. Italic values have been measured before the time-outs have been tuned to appropriate values.

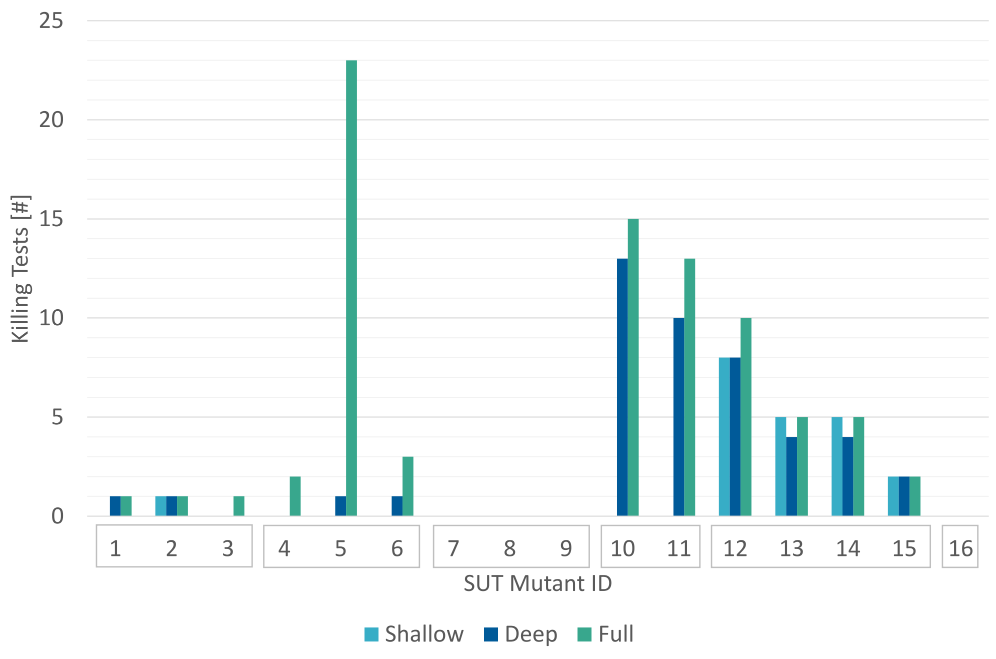


Figure 9.7: Number of kills per SUT mutant and test suite.

9.5.1 Gains

At the time of the TRUFAL project, mutation-based test case generation with MoMuT::UML required several tens of hours on high-end hardware while the concurrent approach implemented in the newer versions of the tool [62] terminates in mere seconds on standard-issue hardware. This significant improvement gives us the flexibility to experiment with more intricate and expensive forms of mutation in our future work. While the generated test suites are shorter and based on fewer killed mutants, they achieve a comparable level of SUT mutant coverage while needing less time to execute. All figures associated with these conclusions are given in Table 9.3. The kill rates of each test suite for individual mutants are depicted in Figure 9.7. Furthermore, we no longer weaken our testing approach by limiting the state surveillance to changing state variables which improves our ability to find faults via the coupling effect and brings us closer to an actual **ioco** check.

9.5.2 Possibilities for Improvement

The aforementioned performance gain could be invested in a depth increase of the breadth-first search algorithm. This might help to reduce the number of weakly killed mutants and increase the number of strong kills. According to the MoMuT developers, reducing weak kills is one of the main priorities of their future work. Another avenue for improvement would be the pre-processing of the MDML decision tree to encode the transitions for all possible state configurations separately into the OOAS. This approach, called "*de-factoring*" [88], could potentially produce very fine-grained mutants but would require an extensive reachability analysis in order to prevent the generation of unreachable OOAS code. As mentioned above, further modification must be made to the model transformation to account for secondary actions in combination with full state observability. The implementation of these modifications must adhere to the practice of test-driven development, which was not observed during our on-the-fly modifications during the evaluation phase. While teardown hiccups might pose an annoyance to test engineers, general CDH hiccups are only identifiable by (repeatedly) re-running all failed tests of a test suite. However, they are limited to test suites generated by the *simple* transformation scheme. The concrete fields of application of each TCT scheme must be evaluated by trial and hands-on experience. While the test engineers complimented the shortness of the generated test cases, they also expressed the need for meaningful test method names as opposed to the currently used ID numbers.

9.5.3 Unsolved Problems

When several state variables of a measurement device change at once, this change is atomic and therefore occurs in no particular order. With the recent modification of the model transformation, this atomic change has been introduced into the OOAS representation. However, due to the test case transformation algorithm, assertions on different state variables are executed in order of state variable declaration[5] in the MDML model. While the first state variable experiences the shortest time-out, the last state variable implicitly experiences the longest. This constitutes a violation of the paradigm of *separation of concerns*, as well as the semantics of the MDML language which states that the order of elements in the *set* of state variables has no effect. Furthermore, the test suites allow a SUT to perform arbitrary state variable changes as long as it exhibits the expected state configuration before the time-out expires. Therefore, intermediate state configurations which are too short to be recognized by the `WaitForChannelValue` methods must be disregarded when creating the MDML model.This problem has been known since the TRUFAL project but we found that it, with few exceptions, had little practical impact on the applicability of our testing method.

[5]This is a side effect of the code generator internally representing sets as ordered lists.

10 Concluding Remarks

10.1 Related Work

The main contributions of this work are the development of a textual DSL as a means to specify the behaviour of measurement devices in the form of state machines and its integration with a pre-existing model-based mutation testing methodology into an industrializable toolchain. In this section, we examine related work which mirrors one or both of these aspects.

10.1.1 DSL-Based Test Case Generation Toolchains

Törsel [100] has created a model-based testing tool for web applications which resembles our test case generation toolchain in several aspects: the front-end DSL structures the modelled information primarily by *views*, bundling their outgoing transitions. Therefore it bears resemblance to MDML when written in the state normal form (see Section 4.2.1). The toolchain is largely composed of the same operations, with the exception of an additional model transformation step which enhances the abstract test cases with additional information while still keeping them independent from the testing tool.

Efkemann and Peleska [40, 39] perform model-based test case generation for safety-critical avionic systems using a graphical DSL called *ITML*. A dedicated IDE provides a modelling front-end as well as the automated generation of executable test procedures. They ensure the *right level of abstraction* by hiding unnecessary details of the test environment. Efkemann also highlights the importance of tool-workflow integration - e.g. by using the Eclipse framework - and the assurance of tool usability [39, p.156].

Similarly, Haxthausen and Peleska [52] have developed a toolchain for the model-based verification and testing of railway systems which uses a graphical DSL in conjunction with a dedicated editor and well-formedness checker. Due to the large number of resulting test cases, a test selection heuristic is used. They use the semantics of *input/output state transition systems* which differ from IOTS by representing inputs and outputs as variables rather than labels.

Stefanescu et al. [95] have developed a graphical DSL called *MCM* for the model-based testing of service choreographies (the communication protocols within a service-based architecture). In an interesting antiparallel to the TRUFAL toolchain, they convert their MCM models into UML models which are then executed by a dedicated engine for the purpose of test case generation and debugging. The resulting abstract test suite can be made to satisfy different coverage criteria, each of which are based on input coverage, possibly in relation with test steps. The MCM editor has been realized as an Eclipse plugin with the integrated transformation from MCM to UML.

Olajubu et al. [79, 78] have automated the generation of test cases based on software requirements. These requirements are modelled in a textual DSL which, like MDML, tries to bridge the gap between natural language and formal specification. An Xtext-based IDE is used to specify the requirements which are then directly translated to test cases by model-to-text transformation scripts. The test selection criteria are dependent on the respective kind of requirement - e.g. representative values for equivalence class-based requirements or modified condition/decision coverage for logical requirements.

Proetzsch et al. [85] have created a testing toolchain for mobile robots, involving multiple DSLs. While one is used to directly specify generic test cases, the other is used for the graphical specification of test models. These models contain transitions between different locations which the robot can visit, possibly weighted by probability, thereby gaining the semantics of a Markov chain. Test cases can be derived either randomly or according to coverage-based strategies.

10.1.2 CNL-Based Test Case Generation Toolchains

Controlled natural languages (CNLs) are subsets of natural languages and are used to make formal specifications accessible to a very broad range of users regardless of their previous knowledge or experience [91]. While these languages might not look domain-specific on the surface, they are clearly defined in terms of scope and semantics. Both aspects are to a certain extent tailored to a specific domain in order to make the languages machine-readable. The intuitive understandability of CNLs was part of the reason why Gherkin was chosen as a baseline for the development of MDML.

The work of Colombo et al. [31] was previously mentioned as prototypical for our use case. They specified a web service through a state machine formalism in the form of Gherkin scenarios. These scenarios are then combined into a common model which serves as an input for a *property-based testing* (PBT) tool called QuickCheck[1] [30]. QuickCheck generates a series of test sequences, attempting to falsify the postconditions specified in the Gherkin scenarios.

Nogueira et al. [71] have developed a model-based testing tool which takes use cases in CNL form as input and converts them into a common labelled transition system. This method is akin to the manual process of compiling individual usage scenarios into a common MDML model. Finally a test suite covering all transitions of the LTS is generated. Moreover, they observed that the use of a CNL made their tool *desirable* to a large user base.

[1]`http://www.cse.chalmers.se/~rjmh/QuickCheck/`

10.1.3 State Machine Representation in Textual Languages

Lastly, we present a few ways in which state machines have been represented in textual languages. Since this category blurs with those of graph specification languages and general purpose languages which have been used for state machine specification, we will restrict this Section to a few non-exhaustive examples. The works of Törsel [100] and Colombo et al. [31] deserve repeated mentioning in this category as they specify their web-services in terms of views and scenarios which resemble the state normal form and a fully de-factored modelling style in MDML, respectively.

Conway and Edwards [32] have developed a DSL for the specification and automated generation of device drivers (e.g. Ethernet adapters). Their language allows for the independent declaration of each state. The notion of different state variables is only implied through the existence of sets of mutually exclusive states. Like in UML state machine diagrams, states may contain entry actions which can specify further state transitions, similar to MDML's secondary actions.

The *FSMLanguage* developed by Agron [4] is used to specify state machines in the context of hardware/software co-design. Here, states and transitions are defined separately. Transitions are guarded and grouped by their source state. Their evaluation priority is determined by their order of declaration. The language only supports one-dimensional state spaces but allows interaction with other state machines via message passing mechanisms, thereby resembling an intermediate development stage of MDML.

Ouimet et al. have created the *Timed Abstract State Machine Language* [80, 81] to specify timed abstract state machines, including non-functional properties like resource consumption. The models contain type and variable definitions, followed by guarded transition rules, resembling the basic header/body structure of MDML models. The language also allows for the parallel composition of multiple state machines.

Ratiu et al. [86] have proposed several extensions to the C programming language, one of which was designed to encode state machines. A state machine definition starts with the declaration of input and output symbols. Analogous to MDML's state normal form, each state is specified as a block statement which contains the definitions of outgoing transitions, including their associated inputs, guards and outputs.

10.2 Future Work

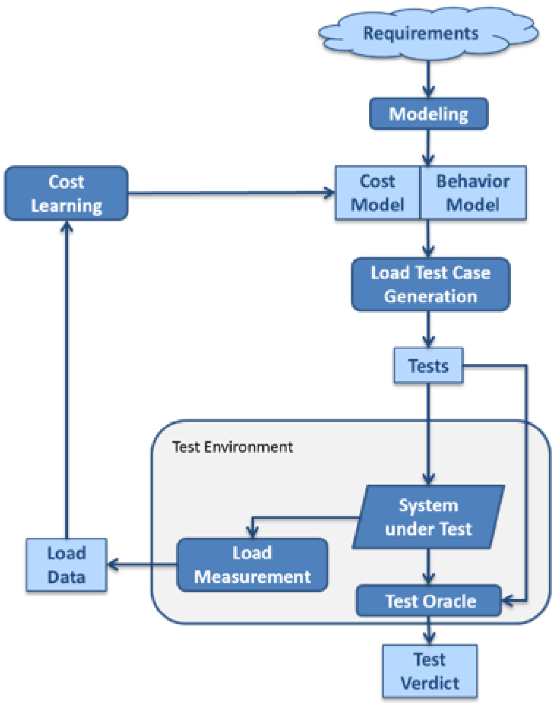


Figure 10.1: The TRUCONF toolchain as envisioned during project inception.

10.2.1 Graphical Model Representation

The opportunities of future work can be adequately illustrated by our initial plans for the TRUCONF toolchain, depicted in Figure 10.1. Starting with the activity of model creation, the first issue to address is that of the graphical representation of measurement device state machines. Though initially planned as part of the TRUCONF project, it has grown out of scope due to our prevalent difficulties with presenting a multidimensional state space in a form which is intuitively understandable to our end-users. Nevertheless, we understand that an adequate graphical representation would significantly enhance the intuitive understandability of device state machines at first glance. Altenhuber [16] has already laid important groundwork for the development of graphical views on MDML models which includes concepts for several different state space representations as well as feedback from our test engineers. With this information as a baseline, we endeavour to extend the MDML IDE with adequate graphical views.

10.2.2 Test Execution and Evaluation

Currently, the TRUCONF toolchain ends with the generation of a test suite in the form of a partial PageObject class file. This file has to be manually transferred to the TAF computer and incorporated into the TAF source tree which then takes about 10 minutes to compile. Afterwards, the generated tests can be run against a connected measurement device via the NUnit framework. Future versions of the TRUCONF toolchain could separate the test case generation from the modelling front-end and move it to a server which also runs a TAF instance. A test engineer would submit an MDML model to the server, together with instructions for test case generation and transformation. The test suite would then be automatically generated, integrated and executed. By incorporating information from already existing MoMuT and NUnit reports (both of which are machine-readable XML files), the test engineer could be given an immediate feedback on the mutants which are probably implemented by the SUT. The model transformation could also be improved by an additional *de-factoring* step which decouples the structure of the MDML model from the granularity of the generated mutants.

10.2.3 Non-Functional Requirements

While timed transitions and communication time-outs can arguably be considered as non-functional requirements, the development of a dedicated NFR measurement and verification method never reached an adequate level of maturity. This was mainly due to the focus of the measurement device use case of the TRUCONF project shifting away from NFR testing and towards usability and industrialization of the functional testing method. While an NFR testing method for measurement devices was developed, the concrete test targets (e.g. for CPU usage or memory consumption), as well as the accuracy of the method (e.g. for network latency) remain unclear. Further work on NFR testing methods for measurement devices will require a more targeted approach which takes the behaviour of the system on the device driver level into account. A closer collaboration with the measurement device engineering department for the purpose of requirement transparency will be necessary.

It may also be possible to incorporate research done within the second use case of the TRUCONF project. This use case concerned itself with the testing of functional and timing-related characteristics of an AVL web service called *testfactory management suite* (TFMS) [13, 15, 14, 90]. The business logic of the web service is represented in the form of a so-called rule engine model from which an extended finite state machine model is derived as input for the test case generation toolchain. Test cases are then either directly generated by a property-based testing tool called FsCheck[2] or indirectly by the incorporation of an external test case generator like MoMuT. The latter was used to generate test cases which are short yet still meet predefined coverage criteria. The test cases are then executed on the SUT to test its functional conformance while simultaneously obtaining probability distributions for certain costs (e.g. individual transition times). These cost measurements are then combined into a cost model, analogous to the feedback loop depicted in Figure 10.1. Via statistical model checking, certain properties of the cost model can be evaluated,

[2] `https://fscheck.github.io/FsCheck`

e.g. the probability that a transition meets a given time constraint. Afterwards, a minimal number of sample measurements are taken on the SUT in order to verify that it meets the predictions made from the cost model. The cost model learning, model checking and verification steps could be integrated into the extended toolchain as it was described in Section 10.2.2. Moreover, the approach would also allow for the testing of several measurement devices which are running in parallel and are operated from the same PUMA system.

10.3 Summary

A domain-specific language has been created as a front-end modelling formalism to facilitate the industrialization of a previously existing model-based mutation testing methodology for automotive measurement devices. This DSL was designed in close and frequent cooperation with the test engineers who would constitute its eventual user base. After observing their habitual workflows, the controlled natural language Gherkin was chosen as a baseline for the development of the measurement device modelling language (MDML).

The scenario-based structure of Gherkin was altered to represent state machines as decision trees which determine the next state transition based on the current state configuration and the current input. These decision trees need not be strictly structured according to the different state variables but offer a large degree of freedom for different modelling styles. This freedom in model structure is bought with an increased margin of modelling errors which demands enhanced user guidance by a dedicated modelling tool.

Along with the language, a plugin for the Eclipse IDE was developed, using the Xtext domain-specific language workbench. This plugin was developed to a full-fledged MDML tool which was integrated with AVL-internal data sources like the device knowledge base. Testing interfaces for device models can be imported directly from the DKB, thereby reducing the need for tedious manual data-gathering. On the other hand, the functional MDML model content has been purposefully kept redundant to uphold the four-eyes principle.

The MDML tool encapsulates the whole test case generation workflow, starting with the transformation of the MDML models to object-oriented action systems. The transformation has been realized as a series of strategies as per the strategy pattern. Therefore, it is easy to maintain the transformation logic by removing, changing or adding individual strategies. The transformation logic, in conjunction with the formal semantics of OOAS, constitutes a formal semantics of MDML.

The OOAS are processed by the MoMuT test case generator which was developed by the Austrian Institute of Technology and the Graz University of Technology. MoMuT uses a fault-based test selection criterion, combined with a search-based test case generation approach. It enhances the OOAS with a set of individual mutations and attempts to find model traces which uncover these mutations through an observable difference in behaviour. The combination of the different mutation operators with the model transformation logic results in the emergence of recurring mutation patterns. These mutation patterns have been examined in terms of their functional non-equivalence.

After an abstract test suite has been generated, it must be transformed to a concrete test suite which resides on the abstraction level of the system under test and can be incorporated into the test automation framework. The sequence of controllable and observable actions which comprises a test case is replaced by method calls to a measurement device PageObject. Depending on the model, this transformation can be guided by different transformation schemes. The *simple* transformation scheme uses high-level PageObject functionality while the AK-command-based transformation scheme operates on communication protocol level. A third transformation scheme directly leverages the properties of the **ioco** conformance relation by covering equal state machine sections of similar measurement devices by common MDML models.

The overall MBT methodology has been evaluated in a case study on the *AVL Particle Counter*, also known as AVL489. The measurement device was emulated by a testbed simulator which was programmed to exhibit a set of different implementation faults. This case study mirrored a similar series of experiments which were conducted at the conclusion of the TRUFAL project, resulting in a direct comparison between the former and current MBT approaches.

10.4 Conclusion

The necessity from which the MDML modelling formalism arose dictates two main requirements: first, it must be significantly more *desirable* to the test engineers than the previously existing approach developed in the TRUFAL project. And secondly, it must be functionally on par with the previous approach. As part of the work presented in this thesis, both of these requirements have been verified.

From an early development stage onward, the MDML language has been subjected to repeated user trials by test users with varying degrees of domain knowledge. Our experience shows that MDML is easy to learn and efficient for creating measurement device models. Invariably, all test subjects were able to use the modelling language after 1-2 hours of training. At later development stages, the focus of the evaluations shifted more and more away from the DSL itself and towards the user experience of the dedicated modelling tool. In the conducted walkthroughs, it proved itself to be well-tailored to its purpose, providing the test engineers with the right balance of flexibility and user guidance. The necessary redundancy of implementation and test data still dictates a life-cycle for MDML models. An initial model is built and iteratively reconciled with the SUT. Any non-conformances are classified as either actual implementation faults or modelling errors. While this model life cycle is unlikely to be disestablished in the foreseeable future, we are confident that the necessary effort for model maintenance has been significantly reduced.

To assess its functional quality, the MDML test case generation toolchain was subjected to the same trial as the previously existing TRUFAL toolchain. In some cases, it was hard to distinguish between the contributions of the revised MoMuT test case generator and those of the encapsulating modelling and model transformation method as many different factors came into play at once. Nevertheless, it has been shown that the MDML toolchain is on par with the TRUFAL toolchain with respect to test case quality, as evident by the number of SUT mutant kills. This conclusion is strengthened by the fact that this result was achieved using a still imperfect version of MoMuT which produced an excessive amount of weak kills and false cases of mutant equivalence. The mutant coverage statistics, along with closer examination of the individual model mutants have shown that the model transformation yields a low number of equivalent mutants. Furthermore, the testing methodology was improved to perform full state surveillance, bringing it closer to an actual **ioco** check. The case study has also uncovered notable differences in the functional quality of different test case transformation schemes. While the AK-command-based TCT scheme yields the best results, the question remains whether or not the simple transformation scheme is of long-term practical relevance.

Bibliography

[1] S. Abrahão, F. Bourdeleau, B. H. C. Cheng, S. Kokaly, R. F. Paige, H. Störrle, and J. Whittle, "User experience for model-driven engineering: Challenges and future directions," in *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*, 2017, pp. 229–236 (cit. on pp. 45, 53).

[2] J. Abrial, *Modeling in Event-B - System and software engineering*. Cambridge University Press, 2010 (cit. on p. 9).

[3] J. Abrial and S. Hallerstede, "Refinement, decomposition, and instantiation of discrete models: Application to Event-B," *Fundamenta Informaticae*, vol. 77, no. 1-2, pp. 1–28, 2007 (cit. on p. 9).

[4] J. Agron, "Domain-specific language for HW/SW co-design for FPGAs," in *DSL '09, Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages, Oxford, UK, July 15-17, 2009*, 2009, pp. 262–284 (cit. on p. 105).

[5] B. K. Aichernig, "Model-based mutation testing of reactive systems - from semantics to automated test-case generation," in *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, 2013, pp. 23–36 (cit. on pp. 12, 14).

[6] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korosec, W. Krenn, R. Schlick, and B. V. Schmidt, "Model-based mutation testing of an industrial measurement device," in *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings*, 2014, pp. 1–19 (cit. on pp. 4, 15, 18, 55, 72, 85, 90, 91, 97, 100).

[7] B. K. Aichernig, H. Brandl, E. Jöbstl, and W. Krenn, "Efficient mutation killers in action," in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, 2011, pp. 120–129 (cit. on pp. 14, 71).

[8] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran, "Killing strategies for model-based mutation testing," *Software Testing, Verification and Reliability*, vol. 25, no. 8, pp. 716–748, 2015 (cit. on p. 71).

[9] B. K. Aichernig, K. Hörmaier, F. Lorber, D. Nickovic, and S. Tiran, "Require, test, and trace IT," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 4, pp. 409–426, 2017 (cit. on p. 71).

[10] B. K. Aichernig, E. Jöbstl, and M. Kegele, "Incremental refinement checking for test case generation," in *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, 2013, pp. 1–19 (cit. on p. 14).

[11] B. K. Aichernig, E. Jöbstl, and M. Tappler, “Does this fault lead to failure? combining refinement and input-output conformance checking in fault-oriented test-case generation,” *Journal of Logical and Algebraic Methods in Programming*, vol. 85, no. 5, pp. 806–823, 2016 (cit. on p. 71).

[12] B. K. Aichernig, F. Lorber, and D. Nickovic, “Time for mutants - model-based mutation testing with timed automata,” in *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, 2013, pp. 20–38 (cit. on pp. 14, 71).

[13] B. K. Aichernig, S. Marcovic, and R. Schumi, “Property-based testing with external test-case generators,” in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, 2017, pp. 337–346 (cit. on p. 107).

[14] B. K. Aichernig and R. Schumi, “Property-based testing with FsCheck by deriving properties from business rule models,” in *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*, 2016, pp. 219–228 (cit. on p. 107).

[15] B. K. Aichernig and R. Schumi, “Property-based testing of web services by deriving properties from business-rule models,” *Software & Systems Modeling*, pp. 1–23, 2017 (cit. on p. 107).

[16] A. Altenhuber, “Improving the comprehension of domain-specific languages by utilizing visualizations,” Master’s Thesis, Vienna University of Technology, 2016 (cit. on pp. 22, 106).

[17] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994 (cit. on p. 71).

[18] J. Auer, “Automated integration testing of measurement devices,” Bachelor’s thesis, Graz University of Technology, Aug. 2013 (cit. on pp. 4, 7, 79, 80, 82, 90).

[19] *AVL FuelExact^{TM}PLU - product guide*, AVL List GmbH, Nov. 2015 (cit. on p. 25).

[20] *AVL Particle Counter - product guide*, AVL List GmbH, Nov. 2013 (cit. on pp. 8, 15, 85).

[21] R. Back and R. Kurki-Suonio, “Distributed cooperation with action systems,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 4, pp. 513–554, 1988 (cit. on p. 9).

[22] ——, “Decentralization of process nets with centralized control,” *Distributed Computing*, vol. 3, no. 2, pp. 73–87, 1989 (cit. on p. 9).

[23] R. Back and K. Sere, “Stepwise refinement of action systems,” *Structured Programming*, vol. 12, no. 1, pp. 17–30, 1991 (cit. on p. 9).

[24] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003 (cit. on p. 55).

[25] M. Bernardino, A. F. Zorzo, and E. de M. Rodrigues, “Canopus: A domain-specific language for modeling performance testing,” in *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*, 2016, pp. 157–167 (cit. on p. 3).

[26] M. M. Bonsangue, J. N. Kok, and K. Sere, “An approach to object-orientation in action systems,” in *Mathematics of Program Construction, MPC’98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, 1998, pp. 68–95 (cit. on pp. 7, 9, 23).

[27] T. A. Budd and A. S. Gopal, “Program testing by specification mutation,” *Computer languages*, vol. 10, no. 1, pp. 63–73, 1985 (cit. on p. 13).

[28] C. Burghard, G. Stieglbauer, and R. Korošec, “Introducing MDML - a domain-specific modelling language for automotive measurement devices,” in *Joint Proceedings of the International Workshop on Quality Assurance in Computer Vision and the International Workshop on Digital Eco-Systems co-located with the 28th International Conference on Testing Software and Systems (ICTSS) 2016*, Graz: CEUR Workshop Proceedings, pp. 28–31 (cit. on pp. 6, 15, 16, 22, 29, 45).

[29] N. Chomsky and M. P. Schützenberger, “The algebraic theory of context-free languages,” in *Studies in Logic and the Foundations of Mathematics*, vol. 35, Elsevier, 1963, pp. 118–161 (cit. on p. 43).

[30] K. Claessen and J. Hughes, “QuickCheck: A lightweight tool for random testing of Haskell programs,” in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP ’00), Montreal, Canada, September 18-21, 2000.*, 2000, pp. 268–279 (cit. on p. 104).

[31] C. Colombo, M. Micallef, and M. Scerri, “Verifying web applications: From business level specifications to automated model-based testing,” in *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014.*, 2014, pp. 14–28 (cit. on pp. 21, 104, 105).

[32] C. L. Conway and S. A. Edwards, “NDL: a domain-specific language for device drivers,” in *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’04), Washington, DC, USA, June 11-13, 2004*, 2004, pp. 30–36 (cit. on p. 105).

[33] N. Cunniff and R. P. Taylor, “Graphical vs. textual representation: An empirical study of novices’ program comprehension.,” in *Empirical Studies of Programmers: Second Workshop*, Ablex Publishing, Norwood, NJ, 1987, pp. 114–131 (cit. on p. 20).

[34] I. Dejanovic, M. Tumbas, G. Milosavljevic, and B. Perisic, “Comparison of textual and visual notations of DOMMLite domain-specific language,” in *Local Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information Systems, Novi Sad, Serbia, September 20-24, 2010*, 2010, pp. 131–136 (cit. on pp. 3, 20).

[35] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978 (cit. on p. 12).

[36] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans, “Featured model-based mutation analysis,” in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 655–666 (cit. on pp. 13, 14).

[37] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, “A survey on model-based testing approaches: A systematic review,” in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, ACM, 2007, pp. 31–36 (cit. on pp. 2, 3).

[38] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs,” *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975 (cit. on pp. 9, 10).

[39] C. Efkemann, “A framework for model-based testing of integrated modular avionics,” PhD thesis, University of Bremen, 2014. [Online]. Available: `http://elib.suub.uni-bremen.de/edocs/00104131-1.pdf` (visited on 04/09/2018) (cit. on p. 103).

[40] C. Efkemann and J. Peleska, “Model-based testing for the second generation of integrated modular avionics,” in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, 2011, pp. 55–62 (cit. on p. 103).

[41] J. Eriksson, “Tool-supported invariant-based programming,” PhD thesis, Turku Centre for Computer Science, Aug. 2010 (cit. on p. 58).

[42] A. Fall and J. Fall, “A domain-specific language for models of landscape dynamics,” *Ecological modelling*, vol. 141, no. 1-3, pp. 1–18, 2001 (cit. on p. 4).

[43] A. Fellner, W. Krenn, R. Schlick, T. Tarrach, and G. Weissenbacher, “Model-based, mutation-driven test case generation via heuristic-guided branching search,” in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*, 2017, pp. 56–66 (cit. on p. 72).

[44] R. B. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, 2007, pp. 37–54 (cit. on pp. 1, 3, 45).

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on pp. 55–57).

[46] T. R. Green and M. Petre, “When visual programs are harder to read than textual programs,” in *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). GC van der Veer, MJ Tauber, S. Bagnarola and M. Antavolits. Rome, CUD*, 1992, pp. 167–180 (cit. on p. 20).

[47] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel, “MontiCore: A framework for the development of textual domain specific languages,” in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, 2008, pp. 925–926 (cit. on p. 45).

[48] *OMG Unified Modelling Language. Version 2.5*, Mar. 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5 (visited on 04/09/2018) (cit. on pp. 3, 7, 16, 23, 31, 32).

[49] *OMG System Modelling Language. Version 1.5*, May 2017. [Online]. Available: http://www.omg.org/spec/SysML/1.5 (visited on 04/09/2018) (cit. on p. 24).

[50] *ISTQB® GTB Standardglossar der Testbegriffe*, 2015. [Online]. Available: http://www.german-testing-board.info/wp-content/uploads/2016/08/CT_Glossar_DE_EN_V30.pdf (visited on 04/09/2018) (cit. on p. 2).

[51] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, 1977 (cit. on p. 12).

[52] A. E. Haxthausen and J. Peleska, "Model checking and model-based testing in the railway domain," in *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*, 2015, pp. 82–121 (cit. on p. 103).

[53] W. Heijstek, T. Kühne, and M. R. V. Chaudron, "Experimental analysis of textual and graphical representations for software architecture design," in *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, Banff, AB, Canada, September 22-23, 2011*, 2011, pp. 167–176 (cit. on p. 20).

[54] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011 (cit. on p. 13).

[55] M. Jimenez, F. Rosique, P. Sanchez, B. Alvarez, and A. Iborra, "Habitation: A domain-specific language for home automation," *IEEE software*, vol. 26, no. 4, 2009 (cit. on p. 4).

[56] E. Jöbstl, "Model-based mutation testing with constraint and SMT solvers," PhD thesis, Graz University of Technology, Institute for Software Technology, Apr. 2014 (cit. on pp. 4, 14, 71, 91).

[57] K. Jogun, "A universal interface for the integration of emissions testing equipment into engine testing automation systems: The VDA-AK SAMT-interface," in *SAE Technical Paper*, SAE International, Mar. 1994 (cit. on pp. 25, 50, 80).

[58] A. N. Johanson and W. Hasselbring, "Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: A controlled experiment," *Empirical Software Engineering*, vol. 22, no. 4, pp. 2206–2236, 2017 (cit. on p. 4).

[59] K. Kapoor, "Formal analysis of coupling hypothesis for logical faults," *Innovations in Systems and Software Engineering*, vol. 2, no. 2, pp. 80–87, 2006 (cit. on p. 13).

[60] S. Kelly and J. Tolvanen, *Domain-specific modeling - Enabling full code generation.* Wiley, 2008 (cit. on pp. 3, 20).

[61] R. Korošec, "Automotive development test process design," AVL List GmbH, Tech. rep. TRUFAL Deliverable D4-3, Jun. 2014 (cit. on pp. 15, 54).

[62] W. Krenn and R. Schlick, "Mutation-driven test case generation using short-lived concurrent mutants–first results," *arXiv preprint*, 2016. [Online]. Available: `http://arxiv.org/abs/1601.06974` (cit. on pp. 14, 71, 72, 101).

[63] 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," in *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, 2009, pp. 186–207 (cit. on pp. 9, 10, 71).

[64] W. Krenn, R. Schlick, S. Tiran, B. K. Aichernig, E. Jöbstl, and H. Brandl, "MoMuT::UML model-based mutation testing for UML," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–8 (cit. on pp. 7, 9, 71).

[65] D. Kroening, "Modelling languages," ETH Zurich, Tech. rep. MOGENTES Deliverable D3.2b, Jun. 2010. [Online]. Available: `http://www.mogentes.eu/public/deliverables/` (visited on 04/09/2018) (cit. on pp. 16, 19).

[66] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Department of Computer Science, Iowa State University, Ames, IA 50011 USA, Tech. Rep., 1998 (cit. on p. 72).

[67] T. Mahatody, M. Sagar, and C. Kolski, "State of the art on the cognitive walkthrough method, its variants and evolutions," *International Journal of Human–Computer Interaction*, vol. 26, no. 8, pp. 741–785, 2010 (cit. on p. 53).

[68] *MoMuT::UML user manual*, Austrian Institute of Technology, Jun. 2017 (cit. on pp. 52, 73).

[69] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 337–340 (cit. on p. 71).

[70] G. Mussbacher, D. Amyot, R. Breu, J. Bruel, B. H. C. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. H. Hill, J. Kienzle, M. Schöttle, F. Steimann, D. R. Stikkolorum, and J. Whittle, "The relevance of model-driven engineering thirty years from now," in *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, 2014, pp. 183–200 (cit. on pp. 1, 3).

[71] S. Nogueira, E. Cartaxo, D. Torres, E. Aranha, and R. Marques, "Model based test generation: An industrial experience," in *1st Brazilian Workshop on Systematic and Automated Software Testing*, 2007 (cit. on p. 104).

[72] A. Nugroho and M. R. V. Chaudron, "A survey into the rigor of UML use and its perceived impact on quality and productivity," in *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9-10, 2008, Kaiserslautern, Germany*, 2008, pp. 90–99 (cit. on p. 3).

[73] A. J. Offutt, “Investigations of the software testing coupling effect,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5–20, 1992 (cit. on p. 13).

[74] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, “An experimental determination of sufficient mutant operators,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996 (cit. on p. 73).

[75] A. J. Offutt and S. D. Lee, “An empirical evaluation of weak mutation,” *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 337–344, 1994 (cit. on p. 72).

[76] A. J. Offutt and J. M. Voas, “Subsumption of condition coverage techniques by mutation testing,” Tech. Rep., 1996 (cit. on pp. 12, 73).

[77] V. Okun, “Specification mutation for test generation and analysis,” PhD thesis, University of Maryland, Baltimore County, 2004 (cit. on p. 13).

[78] O. Olajubu, “A textual domain specific language for requirement modelling,” in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 1060–1062 (cit. on p. 104).

[79] O. Olajubu, S. Ajit, M. Johnson, S. J. Turner, S. Thomson, and M. Edwards, “Automated test case generation from domain specific models of high-level requirements,” in *Proceedings of the 2015 Conference on research in adaptive and convergent systems, RACS 2015, Prague, Czech Republic, October 9-12, 2015*, 2015, pp. 505–508 (cit. on p. 104).

[80] M. Ouimet and K. Lundqvist, “A mapping between the timed abstract state machine language and UPPAAL’s timed automata,” Mälardalen University, Department of Computer Science and Electronics, Tech. Rep., 2007 (cit. on p. 105).

[81] M. Ouimet, K. Lundqvist, and M. Nolin, “The timed abstract state machine language: An executable specification language for reactive real-time systems,” *Proceedings of the 15th International Conference on Real-Time and Network Systems RTNS’07, March 23-29, 2007, Nancy, France*, p. 15, 2007 (cit. on p. 105).

[82] M. Paczona, “Model-based code generation for a battery test and battery emulation system,” Master’s Thesis, CAMPUS 02 - University Of Applied Sciences, Graz, Austria, 2016 (cit. on p. 3).

[83] M. Petre, “Why looking isn’t always seeing: Readership skills and graphical programming,” *Communications of the ACM*, vol. 38, no. 6, pp. 33–44, 1995 (cit. on p. 20).

[84] A. Pretschner and J. Philipps, “10 methodological issues in model-based testing,” *Model-based testing of reactive systems*, pp. 11–18, 2005 (cit. on pp. 2, 16, 47).

[85] M. Proetzsch, F. Zimmermann, R. Eschbach, J. Kloos, and K. Berns, "A systematic testing approach for autonomous mobile robots using domain-specific languages," in *KI 2010: Advances in Artificial Intelligence, 33rd Annual German Conference on AI, Karlsruhe, Germany, September 21-24, 2010. Proceedings*, 2010, pp. 317–324 (cit. on p. 104).

[86] D. Ratiu, B. Schätz, M. Völter, and B. Kolb, "Language engineering as an enabler for incrementally defined formal analyses," in *Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012*, 2012, pp. 9–15 (cit. on p. 105).

[87] D. Santiago, A. Cando, C. Mack, G. Nunez, T. Thomas, and T. M. King, "Towards domain-specific testing languages for software-as-a-service," in *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and CLoud computing co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, Florida, USA, September 29, 2013.*, 2013, pp. 43–52 (cit. on p. 3).

[88] R. Schlick, W. Herzner, and E. Jöbstl, "Fault-based generation of test cases from UML-models - Approach and some experiences," in *Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings*, 2011, pp. 270–283 (cit. on pp. 94, 101).

[89] B. V. Schmidt, *Automatische Testfallgenerierung aus UML-Modellen*, Presentation at CON.ECT Requirements Engineering Trends and Best Practices, Vienna, Austria, Nov. 2013. [Online]. Available: `https://trufal.files.wordpress.com/2013/11/20131120_con-ect_v2_split2.pdf` (visited on 04/09/2018) (cit. on p. 17).

[90] R. Schumi, P. Lang, B. K. Aichernig, W. Krenn, and R. Schlick, "Checking response-time properties of web-service applications under stochastic user profiles," in *Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings*, 2017, pp. 293–310 (cit. on p. 107).

[91] R. Schwitter, "Controlled natural languages for knowledge representation," in *COLING 2010, 23rd International Conference on Computational Linguistics, Posters Volume, 23-27 August 2010, Beijing, China*, 2010, pp. 1113–1121 (cit. on p. 104).

[92] Z. Sharafi, A. Marchetto, A. Susi, G. Antoniol, and Y. Guéhéneuc, "An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension," in *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, 2013, pp. 33–42 (cit. on p. 20).

[93] C. Solís and X. Wang, "A study of the characteristics of behaviour driven development," in *37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, Oulu, Finland, August 30 - September 2, 2011*, 2011, pp. 383–387 (cit. on p. 21).

[94] H. Stachowiak, *Allgemeine Modelltheorie.* Springer-Verlag, 1973 (cit. on p. 2).

[95] A. Stefanescu, S. Wieczorek, and A. Kirshin, “MBT4Chor: A model-based testing approach for service choreographies,” in *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*, 2009, pp. 313–324 (cit. on p. 103).

[96] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework.* Pearson Education, 2008 (cit. on p. 57).

[97] G. Stieglbauer, C. Burghard, S. Sobernig, and R. Korošec, “A daily dose of DSL - MDE micro injections in practice,” in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Portugal, January 22-24, 2018.*, Jan. 2018, pp. 642–651 (cit. on pp. 3, 7, 15, 16, 19, 20, 23, 45, 47, 52–54).

[98] G. Stieglbauer and I. Roncevic, “Objecting to the revolution: Model-based engineering and the industry - root causes beyond classical research topics,” in *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017.*, 2017, pp. 629–639 (cit. on pp. 19, 23, 53).

[99] S. Tiran, “The Argos manual,” Institute for Software Technology, Graz University of Technology, Graz, Austria, Tech. rep. 2012 (cit. on pp. 7, 9, 11, 23).

[100] A. Törsel, “A testing tool for web applications using a domain-specific modelling language and the NuSMV model checker,” in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, 2013, pp. 383–390 (cit. on pp. 103, 105).

[101] J. Tretmans, “Test generation with inputs, outputs and repetitive quiescence,” *Software - Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996 (cit. on pp. 13, 14).

[102] ——, “Model based testing with labelled transition systems,” in *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, 2008, pp. 1–38 (cit. on p. 14).

[103] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012 (cit. on pp. 1, 2).

[104] A. Wasowski, “Flattening statecharts without explosions,” in *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), Washington, DC, USA, June 11-13, 2004*, 2004, pp. 257–266 (cit. on p. 30).

[105] M. Weiglhofer and B. K. Aichernig, “Unifying input output conformance,” in *Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers*, 2008, pp. 181–201 (cit. on pp. 14, 72).

[106] N. Wirth, “Extended Backus-Naur form (EBNF),” ISO/IEC Standard 14977:1996(E), 1996 (cit. on p. 43).

[107] M. Wynne, A. Hellesoy, and S. Tooke, *The Cucumber book: behaviour-driven development for testers and developers*, ser. Pragmatic Programmers. Pragmatic Bookshelf, 2017 (cit. on p. 21).

[108] *Xtext documentation*, Eclipse Foundation / itemis AG, Sep. 2014. [Online]. Available: `http://www.eclipse.org/Xtext/documentation/2.7.0/Xtext%20Documentation.pdf` (visited on 04/09/2018) (cit. on pp. 22, 47, 48).