Visualizing Multi-dimensional State Spaces Using Selective Abstraction

Christian Burghard
AVL List GmbH
Graz, Austria
christian.burghard@avl.com

Luca Berardinelli
Institute of Business Informatics - Software Engineering,
Johannes Kepler University
Linz, Austria
luca.berardinelli@jku.at

***Abstract*—Domain-specific languages (DSLs) are popular for many reasons, such as increasing productivity for developers and improving communication with domain experts. Both textual and graphical DSLs are viable solutions with complementary pros and cons: while graphical DSLs shorten the learning curve and facilitate documentation and communication, textual DSLs aim at higher productivity thanks to more efficient editor functionalities. This paper presents the industrial experience on the adoption of a *hybrid* approach combining an existing textual DSL with a read-only graphical state machine representation (*visualization*), equipped with a *selective abstraction* functionality that offers user-specific, highly configurable views on states and transitions. Our approach is the result of an evolutionary process to improve the modelling experience, relying on frequent user feedback. We argue that a well-tailored visualization is a suitable way to shorten the learning curve and ease the adoption of model-driven approaches in industrial settings.**

***Keywords*-Industrial Experience, Domain-Specific Language, Model-Driven Engineering, User Experience, Graphical Language**

I. Introduction

More than ever, industry faces the problem of the constantly growing complexity of technical systems. Model-driven Engineering (MDE) [1] has emerged as a practice to tackle this complexity increase. Models of real-world systems are created through the process of *abstraction* - i.e. the omission of irrelevant details. The *level of abstraction* - i.e. the decision which information is or is not relevant - ultimately determines the usefulness of a specific model. The correct level of abstraction is as much determined by the problem domain at hand as it is by the engineer's individual understanding thereof. It can therefore vary from application to application and from person to person. While the underlying data of a model remains on a fixed abstraction level, engineers can use different *representations* of the same model which can further rise and fine-tune the abstraction to highlight different model aspects and make the model accessible to people with different backgrounds.

This work presents a combination of textual and graphical modeling languages in the context of a Model-based Testing [2] use case in the automotive domain [3], [4]. It is a result of the HybriDLUX project on improving the ease-of-use of domain-specific tooling in industry.

The AVL List GmbH is the world's leading supplier of testbeds for the automotive industry. Such testbeds, whether for engines, batteries, powertrains or whole vehicles, consist of a variety of measurement devices capturing different quantities like fuel flow and temperature, torque, amount of exhaust particles, among various others. The measurement devices are connected to a testbed control software running on a desktop PC. Due to the high variety of devices, device firmware versions and control software versions, integration testing for testbeds becomes challenging. At AVL, we tackle this complexity through the application of Model-based Testing approaches [2]. In our previous work [4], [5], we presented a textual Domain-specific Language (DSL), the Measurement Device Modeling Language (MDML), which is used by AVL's test engineers to specify measurement device models from which integration tests can be generated. In the work at hand, we extend our MDE approach by incorporating a graphical DSL called State Machine Language (SML), which serves to improve the intuitive understandability of our models via *visualisations* whose content is determined by users via a *selective abstraction* mechanism. Our current MDE solution is the result of an evolutionary user-centric development approach [6], which, as a side effect, serves as a continuous evaluation of our work.

The contributions of this paper are i) the introduction of our MDE framework (hereafter called the *HybriDLUX framework*), in terms of its artefacts and relationships, ii) its advanced visualization capabilities (i.e. selective abstraction) for non-MDE experts, and iii) a retrospect on the successful application of the aforementioned evolutionary development process.

The rest of this paper is structured as follows. Section II introduces the HybriDLUX Model-driven Engineer-

Research herein was funded by the Austrian Research Promotion Agency (FFG), program "ICT of the Future", project number 867535 Hybrid Domain-specific Language User eXperience (HybriDLUX).

ing framework. Section III focuses on model visualisation and selective abstraction concepts and their implementation in the HybriDLUX framework. Section IV outlines the development and evaluation process of SML. Section V refers to relevant related work. Finally, section VI concludes the paper.

II. The HybriDLUX framework

The aim of the HybriDLUX project is to decrease the complexity of our testing processes and increase the degree of automation in our industrial setting through the introduction of MDE principles. The minimal wealth of knowledge for any MDE practitioners includes the three pillar concepts of *model*, *metamodel*, and *model transformation* [1].

Models are abstractions of reality for a given purpose. In MDE, they are prescriptive, machine-readable artefacts, obtained at the end of a modeling activity. Models are connected (i.e., model elements may be linked beyond the boundary of one model), and dynamic (i.e., models may be analyzed and executed in some form) [1], [7].

Metamodels define the modeling concepts and their relationships and provide the intentional description of all possible models, which, in turn, conform to the metamodel. Metamodels define modeling languages in a purely conceptual way and are independent of any form of concrete representation. The concrete syntax of a language assigns graphical or textual elements to metamodel elements that can be understood by users and, possibly, edited through model editors [1].

Model transformations are programs executed by a transformation engine, which take one or more models as input to produce one or more models as output, according to rules that map source and target modeling concepts as defined by the corresponding metamodels [1], [8]. In MDE, model transformations are used to solve different tasks [8], [9] such as code generation, model refactoring, reverse engineering to name just a few.

In this context, the HybriDLUX project is focusing on a Model-based Testing [2] scenario, thus combining modeling and testing activities. Test engineers are demanded to produce *models* representing the software behavior of embedded devices, from which integration test cases are generated, after the model has been *validated*. In particular, HybriDLUX aims at improving the user experience during the modeling phase by combining textual editable models with view-only graphical models or *visualisations*.

Fig. 1 shows the HybriDLUX framework in terms of MDE artefacts, i.e., metamodels, models, transformations, and supporting technologies. The framework comprises two DSLs, namely MDML and SML, to represent the software behavior of automotive measurement devices for the purpose of test case generation. MDML and SML models are integrated by means of an MDML2SML transformation, defined on language concepts from the corresponding metamodels. The modeling use case is detailed

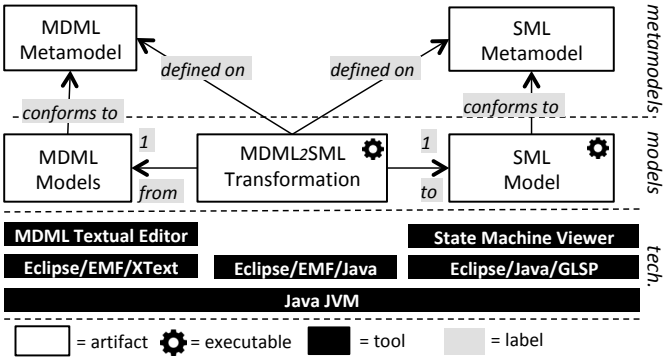

Fig. 1. Overview of the HybriDLUX framework.

by the workflow depicted in the activity diagram in Fig. 2, where models represent the engineering data exchanged among each single step. The rest of this section introduces the MDE artefacts and technologies in Fig. 1 and details the modeling workflow in Fig. 2 with the help of an introductory example, which is intentionally trivial due to space limitation. We model the control software of a measurement device that switches between standby and measurement mode and, at the same time, can be controlled remotely via the control software or manually supervised by an operator.

A. Measurement Device Modeling Language

The Measurement Device Modeling Language (MDML) is a textual DSL that was introduced in [5]. It was designed to model the software behavior of automotive measurement devices, which can be expressed in the form of a state machine. MDML was created in close cooperation with its user base to efficiently encode these measurement device state machines. MDML models represent one of the outcomes of the modeling activity. They are directly edited by users (see Fig. 2) via a dedicated textual editor.

Following the taxonomy by Utting and Legeard [2], MDML can be classified as a *pre/post* language as it expresses each state as a valuation of several *state variables* and each transition through its *pre-* and *post-conditions*. Therefore, neither states nor transitions are explicitly represented in the MDML metamodel, whose excerpt is depicted in Fig. 3, covering the necessary concepts for the purpose of this work. For a thorough description, we refer to [4].

The MDML model of our example use case is distributed over Listings 1,2 and 3. Each MDML model starts with the definition of one or more state variables. Each state variable definition consists of a `public` or `private` qualifier to indicate the variables' visibility from the outside, the keyword `statevar`, the variable name, its value domain (i.e., set of possible values) and its value in the initial state:

```
1  private statevar Level {Manual, Remote} = Remote;
2  public statevar Mode {Standby, Measure} = Standby;
```

Listing 1. MDML State Variables

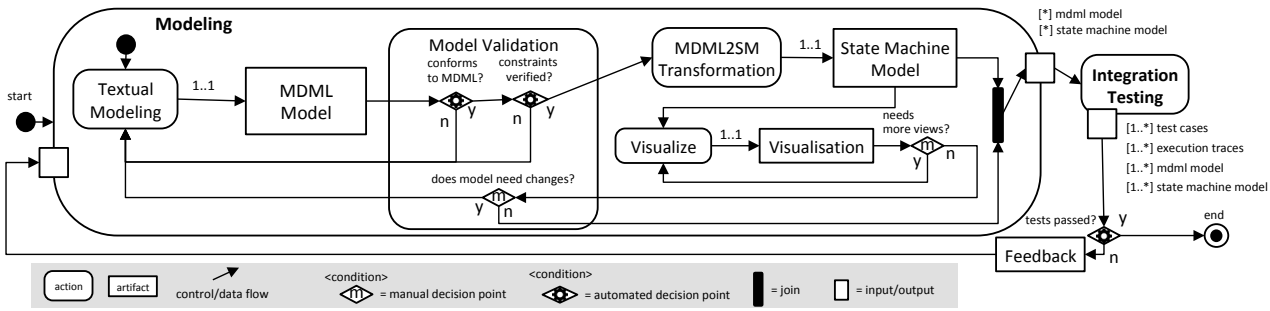

Fig. 2. Modeling in HybriDLUX.

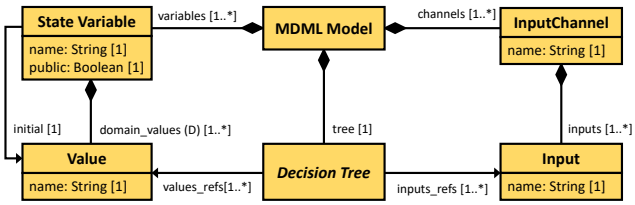

Fig. 3. Excerpt of the MDML metamodel.

In the following, we use n to denote the total number of state variables in a given MDML model and D_j to denote the domain of the j-th state variable. For example, the state variable definitions in Listing 1 induces $D_1 = \{\texttt{Manual}, \texttt{Remote}\}$ and $D_2 = \{\texttt{Standby}, \texttt{Measure}\}$. For the domains of different state variables, we define:

$$D_j \cap D_k = \emptyset \text{ for } 1 \leq j < k \leq n \tag{1}$$

The valuation of all state variables at a given time constitutes the current state of the encoded state machine. The expression of states as the valuation of state variables lends a certain aspect of *multi-dimensionality* to state machines defined in MDML. A similar effect can be achieved in UML state machine diagrams [10] by using multiple *regions*.

In addition to the state variables, the user specifies one or more *input channels*, each defining one or more *inputs*. An input channel definition consists of the keyword `input`, the channel name and its list of possible inputs:

```
input Action {SMAN, SMES, STBY};
```

Listing 2. MDML Inputs

These inputs are used to trigger the transitions of the state machine. Their possible grouping into different input channels is purely cosmetic. Henceforth, we will use I to denote the cumulative set of inputs defined in a given MDML model.

The rest of the model consists of a decision tree (Introduced as an abstract metaclass for explanatory purposes). This tree contains `given` statements, encoding conditions on the current values of state variables (pre-conditions), under which specific value changes can take place. The inputs triggering these value changes are incorporated via `when` statements. Finally, `then` statements specify the individual value changes (post-conditions). If no value change is specified for a specific state variable, its value remains unchanged. The above keywords are reminiscent of various Behaviour-driven Development languages like Gherkin[1], from which MDML was derived [4], [5]. In addition, MDML models may also contain statements in the form of Event-Condition-Action (ECA) notation, similar to UML's state entry actions, as well as time-triggered actions (Time-triggers are treated like inputs and are members of I.). Listing 3 shows a very simple example of a decision tree:

```
when Action = SMAN then Level -> Manual;

given Mode = Standby when Action = SMES
  then Mode -> Measure and Level -> Remote;

given Mode = Measure when Action = STBY
  then Mode -> Standby and Level -> Remote;
```

Listing 3. MDML Decision Tree

In more complex MDML models, multiple `given` statements can be combined and cascaded to form a tree structure, allowing for the concise modeling of multiple transitions at once. The effect is similar to that of UML *super-states*. However, MDML is even more flexible and offers a multitude of options to structure the decision tree and encode a specific state machine behavior. Each user is free to choose a modeling style that suits him or her best.

B. State Machine Language

The State Machine Language (SML) is a graphical and executable DSL. It was designed to give the users visual feedback about MDML-encoded, implicit state machine models. The SML model corresponding directly to the textual MDML model given in Listings 1, 2, and 3, is shown in Fig. 4. SML models cannot be directly edited. Instead, each SML model is obtained from an MDML model through a model transformation [1]. We refer to such a graphical read-only model representation as a *visualization*. Different SML visualizations can be generated and inspected by the users to assess the correctness of the source MDML model contents, and, together, they represent the outcome of the modeling workflow that feed the subsequent integration test activity (Fig. 2).

[1]https://cucumber.io/docs/gherkin/reference/

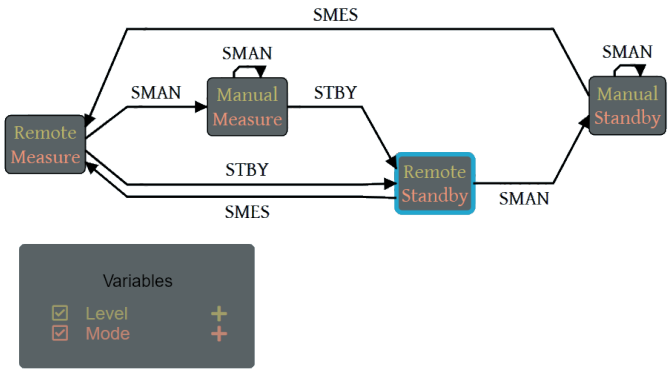

Fig. 4. Visualization of an example state machine.

In contrast to MDML, SML can be classified as a *transition-based* language, which expresses states and transitions explicitly [2]. The only direct overlap between the MDML and SML metamodels are the concepts of state variables and their respective values, as well as inputs. MDML can represent a specific state machine in many different ways, as previously stated. However, a specific state machine is uniquely represented by one particular SML model. Due to this lack of structural freedom, the metamodel of SML is much simpler than that of MDML and shown in its entirety in Fig. 5.

The runtime semantics of SML is well-defined and similar to that of many other well-known state machine formalisms. The set of states in a given *state machine model* (i.e., SML model) is denoted by $Q \subseteq D_1 \times \cdots \times D_n$. Each state $q \in Q$ is an n-tuple of values from each domain. Due to Equation 1, ordering and multiple occurrences become irrelevant and we can express each state as a set $q = \{x_1, \ldots, x_n\}$ of state variable values with $x_j \in D_j$. We write a transition from a pre-state q on an input $i \in I$ to a post-state q' as $\langle q, i, q' \rangle$. The set of transitions in the state machine model $\delta \subseteq Q \times I \times Q$ is also called the *transition relation*. The transition relation fully specifies the state transitions of an SML model. SML allows no additional behavioral specifications akin to ECA formalisms. While the underlying MDML model may contain such elements, they all have been factored into δ.

UML is the most widely-used modeling language in industry [11] and most practitioners are familiar with UML or UML-like notations for state machines. Therefore, the graphical concrete syntax chosen for SML renders states as rounded boxes and transitions as directed arrows between those boxes. Each state is labelled with its corresponding state variable valuation. The initial state is highlighted by a thick blue border. In addition to the model contents, each visualization contains a legend, listing all state variables in the model by name. The state variables are color-coded to match their respective value within the individual states. Transitions are labeled with their respective input.

The concrete syntax of SML is obtained by mapping

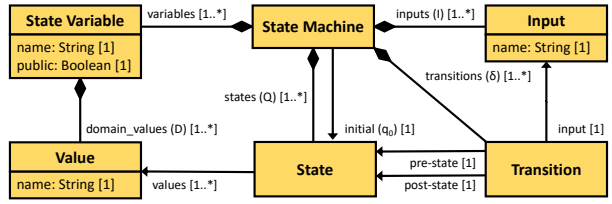

Fig. 5. The complete SML metamodel.

SML metamodel elements to graphical shapes and it is highly configurable through a mechanism called *selective abstraction*, which will be described in detail in Section III. As a result, different concrete syntax configurations (i.e., *abstractions*) can be assigned to SML models, producing different visualizations of the same state machine model.

C. Model Validation and Transformation

The HybriDLUX modeling workflow (Fig. 2) comprises model validation and transformation steps. Validation is performed on both MDML and SML models. Here, the term *validation* refers to the following actions: i) checking the conformance of any MDML artefacts to the MDML metamodel, ii) inspection of MDML and SML artefacts performed by users during the modeling activity, and iii) verification of constraints during the execution of the MDML2SML model transformation. The conformance of MDML models to their metamodel is automatically checked by the model editor. Inspections can be performed on MDML models and SML visualisations to determine if the corresponding source MDML model needs changes. However, thanks to the systematic mapping of MDML to SML, inspections of complex behaviors are better supported by SML visualisations.

Finally, additional validation constraints are encoded in the MDML2SML transformation. Indeed, the role of the MDML2SML transformation is twofold: On the one hand, as expected, it detects patterns of source MDML model elements to be transformed into patterns of target SML model elements. In addition, it verifies some static semantics constraints in order to obtain a valid and executable SML model as an outcome. As a result of both functions and the well-defined runtime semantics of SML, the MDML2SML transformation effectively defines the runtime semantics of MDML. The induced semantics is most closely related to Synchronous Languages [12]–[14], but also similar to that of Guarded Command Languages [15] or Abstract State Machine languages [16], with differences in the combination of individual rules and the semantics of unspecified transitions.

In the following, we give a brief overview of the transformation's general functionality and provide an example based on the MDML model given in Listings 1-3. The transformation algorithm follows an exploratory approach. It starts by establishing the initial state $q_0 \in Q$ of the SML model, as determined by the initial values of each state variable in the MDML model. The state variable definitions in Listing 1, for example, yields the initial

state $q_0 = \{\texttt{Remote}, \texttt{Standby}\}$. After the initial state has been established, the transformation algorithm starts to probe all available inputs. For the state q_0 and the input `SMES` defined in Listing 2, the decision tree in Listing 3 prescribes a value change of the variable `Mode` from `Standby` to `Measure`, creating a newly discovered post-state $q_1 = \{\texttt{Remote}, \texttt{Measure}\}$. The state variable `Level` does not change since it already had the value `Remote` in q_0. Therefore, the transformation logs a new state q_1 and a new transition $\langle q_0, \texttt{SMES}, q_1 \rangle$. If the decision tree does not define a value change for a specific combination of pre-state and input, the transition remains unspecified and will not be included in the transition relation. If the decision tree defines multiple contradictory (i.e., *non-deterministic*) value changes, this constitutes a semantic model validation error and no state machine model can be obtained. This process is repeated for all states in Q (which are discovered on the fly) and all inputs in I. In essence, we perform a closure on Q under the pre/post-relation defined by the MDML model for all inputs in I. The state machine model in Fig. 4 has been obtained by transforming the MDML model from Listings 1-3.

D. The MDML and SML Synergy

We use SML alongside MDML for two main reasons: understandability and executability.

Firstly, SML provides an executable semantics to MDML via the MDML2SML transformation [17]. An explicit state machine representation, as provided by SML, is easily executable and very flexible to use for test case generation, error tracing and other operations that may be implemented in future versions of the HybriDLUX toolchain. Test cases, for example, can easily be represented as lists of transitions. The more implicit MDML representation of the same state machine would be much more unwieldy to use, since the structural freedom of MDML decision trees complicates the computation of the individual transitions. Therefore, we decided to transform each MDML model into an executable state machine model early on in the workflow and use the latter as a basis for all subsequent activities. This allows all semantics-defining code for MDML to be concentrated in the implementation of the model transformation. Due to the aforementioned structural freedom, an optimal transformation from state machine models back to MDML would be highly non-trivial. Thus, we designed the transformation to be unidirectional and the concrete syntax of SML to be read-only.

Secondly, graphical languages have an advantage in communicability and understandability, compared to textual ones [18]. While MDML gives the user a great deal of control over the details of a model, it generally does not offer an intuitive overview, i.e., the user cannot easily judge the degree of completeness of an MDML model at a glance. Therefore, we designed the concrete graphical syntax of SML to mitigate the main drawback of MDML and provide the test engineers with such an intuitive model overview.

E. Model-based Test Engineering Process

The interplay between MDML and SML induces the test engineering workflow depicted in Fig. 2: The test engineer starts by creating an MDML model in a dedicated textual editor. If the MDML model passes the automated syntax and semantics checks, it is transformed into a state machine model, conforming to the SML metamodel, as described in Section II-C. Once the state machine model has been obtained, a visualization can be shown to the user. If this visualization does not suite the user's needs for visual clarity, he or she can obtain different visualizations of the same state machine model through selective abstraction. Once a suitable visualization has been found, the user can judge the overall completeness and correctness of the model. Afterwards, the user can either start the process over by correcting any modeling error in the MDML model, or proceed to use the finished state machine model for the model-based integration testing process, which is described in [4] and exceeds the scope of this work. In short, we perform an automated test case generation step, using fault-based, transition-based and random coverage techniques [2]. The generated test cases are then deployed to a test automation framework for repeated automated execution. After the test case execution, feedback is collected by comparing the workflow outcomes (MDML and SML models, generated test cases and their execution traces). Whenever the generated tests turn out to be inadequate due to a modeling error, the process can again be repeated by refining and/or correcting the given MDML model.

Generating tests according to state-machine-based coverage criteria, such as transition coverage, requires a full state space exploration, as it is accomplished by our model transformation. Currently, our system models at AVL are not large enough to cause a state space explosion. If this were ever to change, we would likely abandon state-machine-based coverage criteria and revert to structural or mutation-based coverage criteria on the decision tree, as we have used previously [4]. For model validation, we could use techniques like Bounded Model Checking [19], which checks the model against temporal logic formulas while avoiding a full state space exploration.

F. Technologies

Our framework is built on an Eclipse[2]-based Rich Client Platform, containing both the Xtext[3] language workbench for textual Domain-specific Languages, as well as the Graphical Language Server Platform (GLSP)[4] to create graphical model representations in SML. GLSP is currently being developed by our project partner

[2] https://www.eclipse.org
[3] https://www.eclipse.org/Xtext/
[4] https://www.eclipse.org/glsp/

EclipseSource[5], who can quickly respond to our project-related needs. We specified the MDML metamodel in Xtext, thereby obtaining a textual editor prototype out-of-the-box. Over time, this editor prototype evolved into a modeling tool, which is used by AVL's test engineers on a daily basis. Due to historic reasons, the metamodel for SML was never formally specified in EMF/Ecore[6]. Instead, it has been realized as a hierarchy of plain Java classes which is not based on any pre-existing framework. This Java-API has, up until now, been maintained AVL-internally and supports the execution of SML models. We consider a formal Ecore-based definition of the SML metamodel part of our future work. The MDML2SML transformation algorithm is currently written in Java.

III. Visualisation and Selective Abstractions

State machine models for industrial use may contain several tens to hundreds of states, resulting in overloaded visualizations. Therefore, we integrated our visualization concept with selective abstraction to help the user to reduce this visual complexity. The selective abstraction mechanism is realised via *state variable reduction* and *state hiding* functionalities that are introduced in the rest of this section. Both mechanisms can be thought of as functions, mapping a visualization of the SML model to a more abstract visualization. Both can be applied multiple times and are mutually commutative.

A. State Variable Reduction

State variable reduction is a refinement of Ladenbergers and Leuschels more general *projection diagram* approach [20]. As per the taxonomy of Liu et al. [21], it can be classified as an *attribute-based node grouping* approach, combining sets of states into superstates. Initially, the visualization contains no reduced state variables and shows the full extent of the SML model given by Q and δ (see Fig. 4). Therefore, all states $q \in Q$ contain one value from each domain D_j, respectively:

$$|q \cap D_j| = 1 \text{ for } j \in [1..n] \tag{2}$$

When the j-th state variable is reduced, a new visualization is created, in which each state q is replaced by a more abstract version $q_{\downarrow j}$ of itself. This abstract state lacks the value from D_j and therefore can no longer be distinguished by it:

$$q \xrightarrow{\text{reduce } j} q \setminus D_j = q_{\downarrow j} \tag{3}$$

All duplicated occurrences of $q_{\downarrow j}$ emerging as a result of the reduction are merged into one, creating a new set of abstract states $Q_{\downarrow j}$. The "reduce" operation constitutes a specific form of the more general *projection function* in [20]. As their pre- and post-states are abstracted, each transition is as well mapped to a more abstract version:

$$\langle q, i, q' \rangle \xrightarrow{\text{reduce } j} \langle q_{\downarrow j}, i, q'_{\downarrow j} \rangle \tag{4}$$

All emerging duplicate transitions are merged into one, resulting in the creation of a new abstract transition relation $\delta_{\downarrow j} \subseteq Q_{\downarrow j} \times I \times Q_{\downarrow j}$ which now may be non-deterministic[7]. However, the underlying model contents are still represented by Q and δ, which are deterministic[8]. All instances of non-determinism in $\delta_{\downarrow j}$ are merely artefacts of the visualization's heightened abstraction level.

The user can reduce a state variable in the legend by unchecking the box next to it (see Fig. 6). This triggers an animation in which all states that were only distinguishable by this variable collapse into their common $q_{\downarrow j}$. This animation is too fast to convey any detailed information to the user but it allows an intuitive understanding of the state variable reduction process. The abstract visualization $(Q_{\downarrow j}, \delta_{\downarrow j})$ can be further abstracted through the reduction of additional state variables, until only one state variable remains.

B. State Hiding

If the SML model visualization obtained by state variable reduction is still deemed too complex to be intuitively understandable, we offer an additional functionality called *state hiding* to further reduce visual complexity. This functionality can be classified as an *attribute-based node sparsification* approach [21], as it allows the user to hide specific states based on individual state variable values $x \in D_1 \cup \cdots \cup D_n$. If the user chooses to hide the value x, each state q is hidden, iff $x \in q$. A transition is hidden iff its pre- and/or post-state are hidden.

The legends in Fig. 4 and 6 display a "+" sign next to each state variable. A click on this sign opens a list of all values within the domain of this state variable. Each value again comes with a checkbox through which all containing states can be hidden and un-hidden. An important difference between state variable reduction and state hiding is that the former technically never removes

[5]https://eclipsesource.com/

[6]https://www.eclipse.org/modeling/emf/

[7]This may happen if the pre-states of two transitions with the same input are merged but their post-states are not.

[8]Being deterministic, δ could be written as a function from pre-state and input to post-state, i.e. $\delta : Q \times I \to Q$. This may no longer be possible with $\delta_{\downarrow j}$.

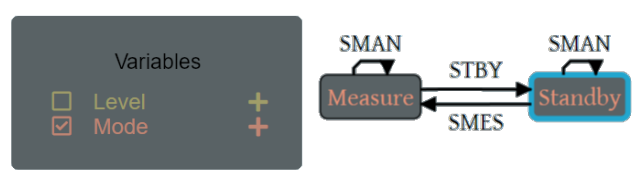

Fig. 6. An alternative visualization of the example state machine with the state variable `Level` reduced.

any states or transitions from the visualisation, but rather combines several of them into one. In contrast, state hiding completely removes states and transitions from the visualisation. Therefore, it could potentially suggest to the user that these states and transitions are not present in the model. To mitigate the risk of such a misunderstanding, we explicitly mark state variables in the legend with a specific icon if their domain contains hidden values.

IV. Continuous Tool Evaluation and Evolution

A. The Value of Good User Experience

The notion of *user experience* (UX) encompasses but at the same time goes far beyond mere tool usability. Essentially, it is an umbrella term for all positive or negative emotions that a user may associate with a given tool. Therefore a tool's user experience is strongly related to its acceptance by a specific user base.

In this regard, if the learning process for a newly developed MDE tool is very long, its introduction to a work environment with fast-paced development cycles becomes difficult. While users invest considerable time to adjust to the new model-driven approach, they endanger their own deadlines and, at the same time, end up with quickly outdated models. This, in turn, motivates the users to adhere to their familiar non-model-based methods.

It was this predicament that led to the rejection of an older and otherwise fully functional UML-based version of our Model-based Testing approach [3]. It is therefore essential to carefully consider how a new MDE approach is *introduced* to a given user base to maximize its UX and, consequently, its chance of acceptance.

B. MDE Micro-Injections

Stieglbauer and Rončević [6] have suggested an approach to maximize the UX of a planned MDE tool by taking immediate influence on its development process. This approach is called *MDE micro-injections*. Its central idea is to synchronize tool development cycles with the development cycles of its future users. In each cycle, a tool prototype is created and evaluated against the current non-model-based approach. This evaluation encompasses both tool functionality as well as regular user feedback, which is obtained through lightweight feedback rounds to not endanger the users' deadlines.

On the part of the tool developer, this process requires the ability of fast tool prototyping, as it is, for example, granted by the Xtext language workbench for textual DSLs. The tool is developed in small increments and in each iteration, the users are exposed to the new MDE principles in *tiny doses* (hence, the name "micro-injections"). Each new iteration must produce a small but measurable benefit. This user-centric development process gives the users the feeling that their opinions are heard and their needs are met. They develop positive feelings towards the MDE tool from the very beginning. They feel that they are provided with a tool that suits their specific needs and that they *desire* to use. The tool introduction process is experienced positively and with confidence rather than as a burden. In turn, the tool developer gains confidence that, through this positive feedback loop, the MDE tool quickly evolves to a form that adequately meets the user and process requirements.

MDE micro-injections have already proven successful in our industrial environment during the development and introduction process of MDML [4], [22].

C. Evolution of SML

Since the inception of MDML, we planned to extend our modeling approach by a graphical model representation [5]. The first viable visualization concept was created by Altenhuber [23]. He proposed to highlight different aspects of MDML models through four different types of visualizations and validated his approach through a series of user interviews. Notably, two of his visualizations directly correspond to SML models with 0 and $n-1$ reduced state variables, respectively. Upon evaluation, one user suggested a generalization of this approach for a selected subset of state variables. The other two visualization types by Altenhuber allowed the users to selectively show and hide parts of the state machine model. These two aspects served as the direct inspiration for our state variable reduction and state hiding functionalities. A new specification, obtained from the results of Altenhuber's work, served as the basis to initiate a micro-injections process which, at the time of writing, is still ongoing.

We regularly improve and evaluate our visualization approach in cooperation with our users, as well as UI experts. In each development cycle, the users receive an updated build of the modeling tool. While the tool is already in productive use at the AVL Test Center and in various stages of introduction in other departments of AVL, we consistently include beta-versions of the SML-based model visualization feature. After receiving the current build, our users are interviewed about how well the newly made improvements aid them in their modeling workflow, and which aspects of the tool could still be improved. Early on, our users confirmed that our selective abstraction approach benefits their ability to intuitively understand complex models and they expressed interest to use it in their day-to-day work. Exemplary scenarios benefiting by the visualization include i) manual completeness checking when writing an MDML model from scratch (see the manual model validation step in Fig. 2), or ii) verifying modifications made to a pre-existing model. The user's positive feelings towards the visualization feature encourage them to give constructive feedback, which is incorporated in subsequent tool releases and helps to further streamline their modeling workflow. It also enables the developers to quickly react to unforeseen user requirements.

The aforementioned completeness checking step provides an example of such an unforeseen requirement. Previously, the manual completeness check of MDML models

against the used reference material involved printing or hand-drawing a visual state machine model and checking off inspected transitions with a highlighter. In response, we have included the possibility to highlight transitions directly in the SML model through a double-click. Other features requested by the users include i) some additional edge sparsification options - e.g. based on different input channels, ii) different representations for input-triggered and time-triggered transitions, e.g. through dashed lines, iii) visual model comparison functionalities and iv) many minor changes like smoother zooming behavior, configurable legend positions, etc. The users requested the visualization to be updated on demand, rather than upon changing the MDML model. While they prefer to switch between MDML and SML tabs, the Eclipse-based editor also allows to display both models side-by-side (see Fig. 7), as well as in different windows. Currently, a UI expert is working on a re-design of SML's visual style, preserving the underlying visual language but updating it to a more compelling and modern look and feel. On our largest productively used model (58 states, 231 transitions), the visualization currently takes a few seconds to load, but the performance improves when the model is simplified through the reduction of state variables. We aim to improve performance on larger models in upcoming iterations of the micro-injections process. The successful application of MDE micro-injections to SML has further shown that a combination of textual and graphical DSLs can be developed in a sufficiently fast and flexible manner with the GLSP framework and that we can maintain short enough tool iterations.

V. Related Work

The most relevant related work is that of Ladenberger and Leuschel [20], who visualized the state spaces of Event-B models via projection diagrams in the ProB animator and model-checker. Since Event-B is a pre/post-language, they also obtain an explicit state machine representation of the model via the ProB model checker. The authors approach the problem of complexity reduction from the point of view of formal methods. Our work approaches it from a MDE-perspective. While the underlying formalisms of projection diagrams and state variable reduction are equivalent, they somewhat differ in their application. With projection diagrams, the user explicitly selects the information to be maintained by the visualization. With state variable reduction, the user rather selects the information to be abstracted away. Also, while the former approach creates visualizations based on queries to the underlying model, the latter is obtained in a step-wise manner by the user as he or she progressively reduces or restores state variables. While the approach in [20] is more general, we believe that a selection of a few good abstraction options helps to make our tool more accessible to users without a background in formal methods. Although ProB offers different options for state space visualization, the authors did not mention any explicit combination of projection diagrams with other complementary means to further reduce the visual complexity of visualizations, like state hiding. Instead, they incorporate customizable graphical syntax elements to highlight the values of specific state variables. The authors argued the usefulness of their approach through a quantitative evaluation against a large state machine model. We find it interesting that we arrived at very similar results through an iterative user-centric approach. We argue that our independently obtained results further substantiate the usefulness of the underlying approach.

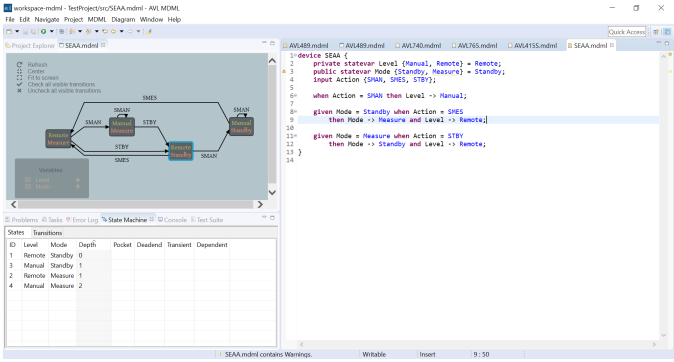

Fig. 7. Screenshot of the MDML editor, including the SML visualization at the top left.

Leuschel and Turner [24] describe two functionalities called *signature merge* and *DFA abstraction* in ProB. Signature merge merges all states q with the same *signature* of enabled inputs $I_S = \{i \in I \mid \exists\, q' \in Q : \langle q, i, q' \rangle \in \delta\}$. DFA abstraction defines a configurable abstraction function $\alpha : I \to I_\alpha$ on the inputs. The input abstraction is followed by a determinization and a minimization step, resulting in a reduced state machine. Van Ham et al. [25] have created an interactive visualization approach for state variable-based state machines, which abstracts information through the clustering of similar nodes by means of an equivalence relation. The clusters are then arranged in a 3D-space in a way that emphasizes symmetries between parts of the state space. Other attributes, like state variable values, are communicated through a configurable coloring functionality. Liu et al. have compiled a survey of existing graph summarization techniques [21]. They have created a taxonomy to categorize existing techniques (like the ones presented in this paper) and cite prototypical examples for each technique.

VI. Conclusion and Future Work

We presented our industrial experience on a Model-based Testing use case, involving the combination of graphical and textual DSLs. We described the graphical language SML, which we developed based on the work of Altenhuber [23] to use in combination with - and mitigate the drawbacks of our textual language MDML. We have presented SML's selective abstraction mechanism. We

have outlined the development of SML, which followed the MDE micro-injections process. The successful application of this process is indicated by our positive user feedback and suggests that our visualization approach benefits the intuitive understanding of our state machine models and is an adequate solution to the problem of increasing complexity in our application domain. The results show that the GLSP framework is adequate for the fast-paced development of graphical and/or hybrid MDE tools and they further substantiate the findings of Stieglbauer and Rončević [6], as well as Ladenberger and Leuschel [20].

We will likely continue the development of SML for some time, with a possible focus on model debugging techniques. Also, we plan to re-define the SML metamodel based on EMF/Ecore and make it compatible to the AutomataLib[9] Java library for state machine data structures to ensure interoperability with other applications. Together with the Ecore-based definition of the SML metamodel, we will internally evaluate the benefits in using transformation languages (e.g., ATL [26] or ETL [27]), instead of Java, for implementing model transformations in the HybriDLUX framework.

[9]https://learnlib.de/projects/automatalib/

References

[1] M. Wimmer, J. Cabot, and M. Brambilla, *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2017.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007.

[3] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korošec, W. Krenn, R. Schlick, and B. V. Schmidt, "Model-based mutation testing of an industrial measurement device," in *International Conference on Tests and Proofs*. Springer, 2014, pp. 1–19.

[4] C. Burghard, *Model-based Testing of Measurement Devices Using a Domain-specific Modelling Language*. Master's Thesis at Graz University of Technology, 2018.

[5] C. Burghard, G. Stieglbauer, and R. Korošec, "Introducing MDML - A domain-specific modelling language for automotive measurement devices," *CEUR Workshop Proceedings*, vol. 1711, pp. 28–31, 2016.

[6] G. Stieglbauer and I. Rončević, "Objecting to the revolution: Model-based engineering and the industry-root causes beyond classical research topics." in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS-Science and Technology Publications, Lda, 2017, pp. 629–639.

[7] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[8] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[9] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. Selim, E. Syriani, and M. Wimmer, "Model transformation intents and their properties," *Software and Systems Modeling*, vol. 15, no. 3, pp. 647–684, 2016.

[10] *OMG Unified Modelling Language. Version 2.5.1*, Object Management Group Std. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF

[11] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 869–891, 2013.

[12] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992. [Online]. Available: https://doi.org/10.1016/0167-6423(92)90005-V

[13] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems," in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987, pp. 178–188. [Online]. Available: https://doi.org/10.1145/41625.41641

[14] B. K. Aichernig, K. Hörmaier, F. Lorber, D. Nickovic, and S. Tiran, "Require, test, and trace IT," *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 4, pp. 409–426, 2017. [Online]. Available: https://doi.org/10.1007/s10009-016-0444-z

[15] E. W. Dijkstra, "Guarded commands, non-determinancy and a calculus for the derivation of programs," in *Language Hierarchies and Interfaces, International Summer School, Marktoberdorf, Germany, July 23 - August 2, 1975*, ser. Lecture Notes in Computer Science, F. L. Bauer and K. Samelson, Eds., vol. 46. Springer, 1975, pp. 111–124. [Online]. Available: https://doi.org/10.1007/3-540-07994-7_51

[16] Y. Gurevich, "Evolving algebras: an attempt to discover semantics," in *Current Trends in Theoretical Computer Science - Essays and Tutorials*, ser. World Scientific Series in Computer Science, G. Rozenberg and A. Salomaa, Eds. World Scientific, 1993, vol. 40, pp. 266–292. [Online]. Available: https://doi.org/10.1142/9789812794499_0021

[17] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson, "Semantic anchoring with model transformations," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2005, pp. 115–129.

[18] F. Ciccozzi, M. Tichy, H. Vangheluwe, and D. Weyns, "Blended modelling – what, why and how," in *MPM4CPS workshop*, September 2019. [Online]. Available: http://www.es.mdh.se/publications/5642-

[19] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, pp. 117–148, 2003. [Online]. Available: https://doi.org/10.1016/S0065-2458(03)58003-2

[20] L. Ladenberger and M. Leuschel, "Mastering the visualization of larger state spaces with projection diagrams," in *International Conference on Formal Engineering Methods*. Springer, 2015, pp. 153–169.

[21] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 62:1–62:34, 2018. [Online]. Available: https://doi.org/10.1145/3186727

[22] G. Stieglbauer, C. Burghard, S. Sobernig, and R. Korošec, "A daily dose of DSL," in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS-Science and Technology Publications, Lda, 2018, pp. 642–651.

[23] A. Altenhuber, *Improving the Comprehension of Domain-Specific Languages by Utilizing Visualizations*. Master's Thesis at Vienna University of Technology, 2016.

[24] M. Leuschel and E. Turner, "Visualising larger state spaces in ProB," in *International Conference of B and Z Users*. Springer, 2005, pp. 6–23.

[25] F. van Ham, H. van de Wetering, and J. J. van Wijk, "Interactive visualization of state transition systems," *IEEE Trans. Vis. Comput. Graph.*, vol. 8, no. 4, pp. 319–329, 2002. [Online]. Available: https://doi.org/10.1109/TVCG.2002.1044518

[26] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008. [Online]. Available: https://doi.org/10.1016/j.scico.2007.08.002

[27] D. S. Kolovos, R. F. Paige, and F. Polack, "The epsilon transformation language," in *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, ser. Lecture Notes in Computer Science, A. Vallecillo, J. Gray, and A. Pierantonio, Eds., vol. 5063. Springer, 2008, pp. 46–60. [Online]. Available: https://doi.org/10.1007/978-3-540-69927-9_4