Continuous Development and Management of Credible Modelica Models

Leo Gall[1] Martin Otter[2] Matthias Reiner[2] Matthias Schäfer[1] Jakub Tobolář[2]

[1]LTX Simulation GmbH, Munich, Germany, leo.gall@ltx.de
[2]German Aerospace Center (DLR), Institute of System Dynamics and Control, Wessling, Germany

Abstract

Modeling and simulation is increasingly used in the design process for a wide span of applications. Rising demands and the complexity of modern products also increases the need for models and tools capable to cover areas such as virtual testing, design-space exploration or digital twins, and to provide measures of the quality of the models and the achieved results. In this article, we try to summarize the state-of-the-art and best-practice from the viewpoint of a Modelica language user, based on the experience gained in projects in which Modelica models have been utilized in the design process. Furthermore, missing features and gaps in the used processes are identified.
Keywords: credible model, model requirement, data management, validation, verification, Modelica model

1 Introduction

The modeling of physical systems is a complex process with many decisions, simplifications and assumptions on its way, usually done by many different decision-makers. In the real world, this often leads to simulation models and simulation results that are not well documented. Due to the increasing use of system simulation in nowadays product development, there is an increasing need in traceable model development with guarantees on model validity.

Models are often shared across organizational borders (for example, from supplier to OEM). On this way the direct access to model sources (for example to internal repositories) and the model history is lost. To avoid this, models have to "carry" documentation "with them". If only Black-Box Functional Mock-Up Units (FMU)[1] or result data with simulation reports are shared, the requirements on credibility and traceability are even higher.

The ITEA 3 project UPSIM[2] aims for system simulation credibility via introducing a formal simulation quality management approach, encompassing collaboration and continuous integration for complex systems. It shall be based on the recently proposed "Credible Simulation Process" (Heinkel and Steinkirchner 2021).

In this article, it is tried to summarize the state-of-the art in the development and management of credible Modelica models, as a basis for future improvements in the UPSIM project. The goal is to achieve a well-documented, traceable development process for Modelica based "*credible* digital twins".

The paper is structured as follows. First, a general overview of the modeling and simulation process is given in Section 2. Then, in Section 3, the former is particularly discussed, focusing on requirements, level of details, validation, etc. The management of Modelica libraries and simulation data is finally elaborated in Section 4.

Even if several of the addressed points are valid for any kind of model-based simulation, the focus of this paper is on the Modelica point of view.

2 Modeling process

A modeling process can be roughly divided into the following steps, examined in the corresponding sections:

1. Definition of the requirements, Section 3.1.

2. Definition of the modeling task, Section 3.2
(based on the requirements - for which purpose is the model needed?).

3. Implementation of the model, Section 3.3
(which detail is necessary for the modeling task, based on which data?).

4. Calibration and validation of the model, Section 3.4
(determination of the parameters of the model based on available data sources or measurement data).

5. Usage of the model.

The different steps in the modeling process are often performed by different persons, either in the same company or also across different companies (for example supplier and OEM). Thus, in order to have a traceable model, all relevant information has to be managed during the whole modeling process. Therefore, a source-code management (Section 4.1) and a version management (Section 4.2) is necessary and the utilized resources need to be documented (Section 4.3)

One simple approach is to use a template with the above-mentioned items (for example as Word or as Markdown document) and to store the filled-out template in the model, for example as file `model_history.md` in the

[1]https://fmi-standard.org/
[2]https://www.upsim-project.eu/

root directory of the relevant Modelica package. For every item a short summary must be included in the template together with a description where the full details can be found. This might be e.g., a report in the *Resources* directory of the package (cf. Section 4.3), a publication, an internal report of the organization, or repositories or file servers where simulation results or measurement data are stored.

In the future, it would be also very helpful if the log-comments of a version management system are automatically extracted and attached to the model documentation, in order to get easy access to this, usually, very valuable information. In open-source projects the release information of a software contains often a one-line comment and a link to every issue and/or pull/merge request for this release. The same process could be also done for models (Section 4.2).

Another approach to track changes of artefacts during the modeling process is presented in (König et al. 2020). The changes can be tracked more or less automatically by collecting standardized information, which is sent by the different tools involved in the modeling process, in a database using server communication. Unfortunately, the number of tools currently supporting this traceability approach is currently small, but is planed to be extended.

The focus of this paper is the development of a model until a state "ready to use" is reached. The usage of the model itself, including the maintenance of the model throughout its whole life-cycle, is, in contrast, not considered in this paper.

3 Model development

3.1 Requirements

As described in Section 2, the modeling process starts with the definition of the requirements to be fulfilled by the final model. Requirements are typically developed and defined textually in a document-based development process using natural-language that might be following some rules, such as the Easy Approach to Requirements Syntax (EARS), see (Alistair and Wilkinson 2019). As a typical example, see the requirements MIL-STD-704F for electrical systems in US military aircraft (Department of Defense 2016). Definitions of requirements with natural language might be supported with appropriate tools, such as DOORS [3] from IBM or Reqtify[4] from Dassault Systèmes, to get support for *collaboration, traceability* and *coverage analysis*.

There are several proposals and attempts to define and check requirements more formally, such as (Schamai 2013) where it is proposed to generate Modelica code for this purpose. At Electricité de France (EDF), the special language FORM-L – FOrmal Requirements Modelling Language, (Thuy 2014), was developed that *formally* defines requirements in a language close to the textual notation used by system designers. The FORM-L language is centered around the four basic questions: *What*, *Where*, *When*, *How Well*. The example from (Bouskela and Jardin 2018) maps the natural language requirement

> R1: While the system is in operation, the pump must not be started more than twice.

into the following FORM-L definition:

> **requirement** R1 **is**
> **for all** pump **in** system.pumps
> **during** system.inOperation
> **check**
> **count** (pump.isStarted **becomes** true) <= 2

FORM-L uses two and three-valued logic to define the logical parts of requirements. The reason to use three-valued logic is that in certain situations it is not possible to state whether a property is *satisfied* (= true) or *violated* (= false) and, therefore, a third value type *undefined* is introduced.

In order to make the FORM-L language directly accessible for the Modelica community, the open source Modelica library *Modelica_Requirements*[5] was developed (Otter et al. 2015). The library has about 200 model and block components and about 50 functions. It allows to define requirements with drag & drop and to "bind" these definitions to Modelica models, so the requirements are always checked when the models are simulated. It is then reported, whether requirements are satisfied, violated, or not tested. Due to the needs of the *Modelica_Requirements* library, some additional functions have been introduced in the Modelica Standard Library (MSL)[6], in particular the functions of sub-package *Modelica.Math.FastFourierTransform*.

In (Bouskela and Jardin 2018), the new Extended Temporal Language (ETL) is described for the simulation of the temporal aspects of FORM-L. ETL introduces a four-valued logic by the additional value *undecided* (meaning that the "decision making" is in progress), in contrast to *undefined* (meaning that the "condition" is not applicable). There is also the Modelica package *ReqSysPro* under development at EDF to practically use ETL within a Modelica model. For more details about the usage of FORM-L in the Modelica community, see also (Bouskela, Falcone, et al. 2021).

The abovementioned approaches define requirements formally and perform simulations on Modelica models to automatically retrieve answers whether requirements are satisfied, violated, or whether no answer can be given. In order to make that possible, corresponding scenarios must be defined (see Section 3.2). Thus, particular simulation runs must be selected to perform these tests. It might not be obvious to determine suitable scenarios. Instead, the

[3] https://www.ibm.com/products/requirements-management
[4] https://www.3ds.com/products-services/catia/products/reqtify/
[5] https://github.com/modelica-3rdparty/Modelica_Requirements
[6] https://github.com/modelica/ModelicaStandardLibrary

Proceedings of the 14th International Modelica Conference
September 20-24, 2021, Linköping, Sweden
DOI
10.3384/ecp21181359

central question is to figure out the scenarios and environmental conditions for which the requirements are not fulfilled, for example to figure out a (valid) load condition or an operating point, where a controlled system is unstable.

A standard approach is to use *Monte Carlo Simulations*, generating parameter and initial values randomly according to given distributions and perform simulations for every randomly selected value set. Most Modelica tools support Monte Carlo Simulation. There are also dedicated tools, such as Persalys[7]. The drawback for typical industrial usage scenarios is, that the number of parameters and initial values is so large, that it is easily possible that operating points are not found where requirements are violated. An alternative is to utilize more intelligent search processes:

- Since 1997, *worst-case optimization* is routinely used at DLR's Institute of System Dynamics and Control to tackle such problems. Hereby, multi-criteria optimization problems are defined so that systems behave as worse as possible. For more details, see e.g. (Bals, Fichter, and Surauer 1997; Joos 2015; Labusch et al. 2014).

- (Corso et al. 2020) provides a survey of optimization algorithms for black-box safety validation: "[..] finding disturbances to the system that cause it to fail (falsification), finding the most-likely failure, and estimating the probability that the system fails".

- TestWeaver from Synopsis[8] (Tatar and Mauss 2014) automatically generates stimuli for co-simulations of virtual ECUs, plant models and requirement watchers. The main generation strategy is the Coverage-Driven Generation which combines random, combinatorial and optimization strategies with the goal to increase (discrete) coverage measures and to find worst-cases for (continuous) quality measures.

3.2 Simulation scenarios

The next step in the modeling process is the definition of the modeling task. This implies a definition of the simulation scenarios, the model should be used for. Human-readable Modelica code is very well suited for compact storage of defined scenarios. But, there are many ways to handle static and dynamic boundary conditions of a Modelica model and to define test cases.

3.2.1 Parameters

When preparing a model for a specific simulation task, the source of parameter values should be documented: Who changed the default value and why? At the instance level, changing parameters means introducing *modifiers*. Modelica modifiers do not provide the ability to comment modifications. A good practice in larger Modelica projects is to define at which hierarchical level the parametrization should happen. Therefore, if there is a defined level, the source of parameters can be documented on Modelica info layer: component data sheets, measurements, educated guess, optimization results, etc. Another approach is to store parameters with this documentation in replaceable hierarchical records in a separate Modelica library or sub-library on a different file as the model, in order that changes to parameters are directly/easily visible in the version control system.

The parameter handling of Modelica should be improved to better support the formal definition of credible models. In principle, extensions could be introduced via *Custom Annotations* (Zimmer, Otter, and Elmqvist 2014), but large scale usage is currently not user-friendly. Furthermore, a standardized solution is needed, supported by all the Modelica tools. In particular, the following features would be useful:

- Defining the domain of validity of the model, preferably with a language element and not just in the documentation.

- Define parameter tolerances and/or uncertain distributions (such as normal or uniform distribution).

- Introduce an orthogonal concept to parameter propagation by mapping parameter values, their tolerances and distributions into a model, so that model structure/equations and model data can be much easier separated. This could be done by the *merge* concept proposed in (H. Elmqvist et al. 2021).

3.2.2 Boundary conditions

A specific system model can be used in different scenarios: a) simulation of stationary load points, b) simulation with dynamic boundary conditions, or c) with external boundary conditions (e.g. co-simulation or hardware-in-the-loop (HIL)).

The dynamic boundary conditions can be important for the correct initialization of a model. One example would be to initialize a system based on the correct environment temperature and pressure. From our past experience, it is hard to initialize based on external dynamic inputs, e.g. coming from table data or via co-simulation. This is, because in Modelica, the start values of input variables can not be directly used as initialization parameters (parameters having lower variability as inputs). To overcome this problem, users have to write additional initial equations in order to assign input values to initialization parameters, see example code in Listing 1.

If the equations allow initalization of dynamic states via connectors, graphical solutions like `Modelica.Mechanics.Rotational.Components.InitializeFlange`[9] are possible.

[7] https://persalys.fr

[8] https://www.synopsys.com/verification/virtual-prototyping/virtual-ecu/testweaver.html

[9] https://doc.modelica.org/Modelica%204.0.0/Resources/-helpDymola/Modelica_Mechanics_Rotational_Components.html

Listing 1. Initalization of states from output of CombiTimeTable

```
...
  parameter SI.Temperature T_initial(fixed=false) "Init, to be set from table output";

  Modelica.Thermal.HeatTransfer.Components.HeatCapacitor heatCapacitor(
    C=100, T(start=T_initial)) "Component to be initalized";

  Modelica.Blocks.Sources.CombiTimeTable combiTimeTable_Tamb
    "Boundary condition on table file";

initial equation
  T_initial = combiTimeTable_Tamb.y[1] "Get first time point of table data";
...
```

The open source Modelica library `ExternData`[10] from Thomas Beutlich and further contributors is very helpful in this respect, because it allows to read data from CSV, INI, JSON, MATLAB MAT (v4,v6,v7,v7.3), SSV (System Structure Parameter Values v1.0)[11], TIR (Tire properties), Excel XLS and XLSX, and XML files at the start of a simulation *without newly compiling the model*. A Claytex TechBlog[12] gives a good description how to use this library. Data from INI files can be read in a similar way with the open source Modelica library `DeviceDrivers`[13].

3.2.3 Test case definition

For the definition of *test cases*, usually a scripting language is needed to describe the wide variety of occurring situations. There is neither a standardized nor an accepted scripting language available in the Modelica community. Depending on user's preferences and the support of scripting languages in the used Modelica tool, one of the following scripting languages is typically used: Dymola functions or script files (mos-files), Microsoft Excel (called Excel in the following text) via Excel-plugins such as XRG Score[14] or TLK Simulator for Excel[15], Maple, Mathematica, Matlab, Python. Industrial users have often a strong preference for Excel or Python.

A new, interesting possibility is proposed in (Buse and Bellmann 2021) where one or more instances of a Lua interpreter[16] can be attached to a Modelica model via the Modelica external object interface. Hereby complex scenarios can be defined in a combination of Lua- and Modelica-code. The Lua interpreter itself is a small DLL (Dynamic Link Library) that is included in a Modelica library and therefore does not require any download or installation, contrary to other scripting environments that are used with Modelica tools.

The following examples demonstrate how Dymola's scripting can be utilized for the definition of test cases.

Example 1: Automotive driving maneuvers There exists a range of well defined standard scenarios to identify and verify the particular behavior of a vehicle. They are typically referred to as open loop or closed loop driving maneuvers. The former comprises for example ISO 4138 steady-state cornering, the latter ISO 3888 double lane change. The example provided here uses Dymola built-in functions.

Considering a vehicle's architecture as defined in the *VehicleInterfaces* library[17](Dempsey et al. 2006), see Figure 1, the particular driving maneuver conditions can be often controlled within the driver model only. Making this element replaceable enables to adapt the desired driving conditions from outside, e.g. when simulating the model. Utilizing this feature, there can even be implemented a function to simulate various driving maneuvers in one turn.

In the predefined vehicle's architecture, called `ConventionalVehicle` in Listing 2, a template of a vehicle model is put together with its environment. Therefore, a particular user's vehicle model shall be redeclared here and the architecture shall be checked against possible compiling errors in a pre-processing step. Then, the function in Listing 2 can be executed with particular settings for each of the desired maneuvers. The two exemplary maneuvers, mentioned above, are given in Listing 2.

Listing 2. Run driving maneuvers

```
function runManouvers
  input String modelSim =
    "VehicleLibrary.ConventionalVehicle"
    "Model to be simulated"
    annotation (
      Dialog(
        __Dymola_translatedModel(
          translate=false)));
  input String priorRedeclarations = ""
    "User defined prior modifications, e.g.
    'redeclare class block(par1=...),'";
  input String resultFile = "myVehicle"
```

[10] https://github.com/modelica-3rdparty/ExternData
[11] https://ssp-standard.org/
[12] https://www.claytex.com/tech-blog/using-external-data-in-your-dymola-model/
[13] https://github.com/modelica-3rdparty/Modelica_DeviceDrivers
[14] https://www.xrg-simulation.de/de/produkte/applications/score
[15] https://www.tlk-thermo.com/index.php/en/simulator-suite
[16] https://www.lua.org/
[17] https://github.com/modelica/VehicleInterfaces

DOI 10.3384/ecp21181359

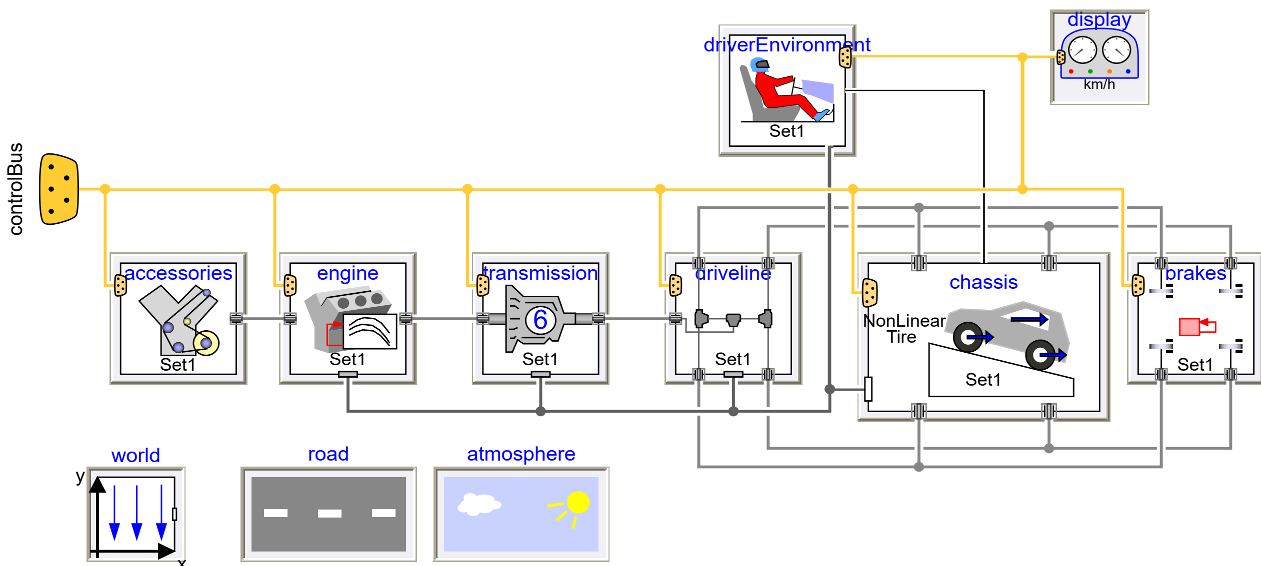

Figure 1. Common vehicle's architecture as defined in the VehicleInterfaces library and inherited in the PowerTrain library.

```
    "Result file(s)";
  input Boolean cornering40 = false
    "ISO 4138 Cornering: R = 40 m";
  input Boolean laneChange80 = false
    "ISO 3888-1 Double lane change: v_x =
       80 km/h";
  ...
protected
  String problem;
  String modifier;

algorithm
  if cornering40 then
    modifier :=
      "VehicleLibrary.Drivers." +
      "SteadyCornering" +
      " driverEnvironment(" +
      "vInit=2,Rcurve=40)";
    problem :=
      modelSim + "(" + priorRedeclarations
        + "redeclare " + modifier + ")";
    ok := simulateModel(
      problem = problem,
      startTime = 0.0,
      stopTime = 250,
      resultFile = resultFile
        + "_StationaryCircle40m");
  end if;

  if laneChange80 then
    modifier :=
      "VehicleLibrary.Drivers.LaneChange" +
      " driverEnvironment(" +
      "vInit=22.22,variant=ISO3888_1)";
    problem :=
      modelSim + "(" + priorRedeclarations
       + "redeclare " + modifier + ")";
    ok := simulateModel(
      problem = problem,
      startTime = 0.0,
      stopTime = 15,
      resultFile = resultFile
        + "_DoubleLaneChange80kmh");
  end if;
  ...
end runManouvers
```

The core of the function are single simulation calls of Dymola's `simulateModel` function for each of the maneuvers, carried out for the predefined `ConventionalVehicle` model. Particular maneuver conditions are given by the predefined attribute `modifier`. Furthermore, maneuver specific simulation conditions are set, such as the simulation stop time. Moreover, each driver model contains an option to stop the simulation once maneuver-specific conditions are reached – for example the maximum lateral acceleration for the steady-state cornering.

Besides fixed maneuver-specific attributes, the attribute `priorRedeclarations` additionally enables to modify the remaining blocks of the `ConventionalVehicle`. Thus, for example variants of power train, environment or controller can also be defined.

Example 2: Refrigeration cycle When simulating refrigerant cycles, there are usually subsequent tasks performed interactively or via scripting: The first step might be a parameter variation to define a suitable refrigerant charge. Then, stationary results can be verified, e.g. by checking the pressure-enthalpy-diagram. After this verification step, larger simulations studies on control points and environmental parameters, like air temperature, moisture and speed, can be defined.

3.3 Model complexity

After the definition of the simulation scenarios, the complexity of the models should be considered. The decision about complexity is based on the scenarios, the model is implemented for.

3.3.1 Adequate modeling detail

Models describing every effect of a (technical) system, don't exist. A model is always created for a specified set of scenarios. Utilizing the scenario-driven model for another scenarios can either lead to inappropriate quality of results, because the desired effects are not modeled, or it needs an unnecessary parametrization effort and a lot of computational resources, because it is too complex for the expected outcome. An example of a mismatching vehicle model is shown in Figure 2. Assuming a line-change maneuver as a scenario, both submodels are inappropriate. With the single-track model the roll angle of the vehicle can not be evaluated and the FE-model, calculating the deformation of the rim, is far too detailed for this scenario. Instead it can be assumed to be rigid .

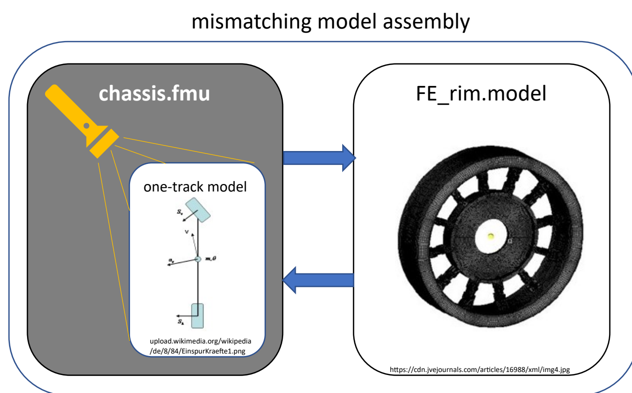

Figure 2. Mismatching model assembly. A simple single-track model of a vehicle (left) versus a complex finite element model of a wheel's rim (right, from (Wei et al. 2016)).

The accuracy of the worst submodel in an assembly (e.g. the less detailed) generally determines the total accuracy. So it's reasonable to assemble models of similar complexity, depending on the aim of the use case. In the example either a rigid rim or a more complex chassis-model should be used. The adequate complexity of a model also depends on the desired outcome of the whole system. If the model has only a small influence on the outcome, it can be modeled less complex.

A model developer should keep some questions in mind while creating a scenario-driven model:

- What is the desired output of the model?
- Which accuracy does the model need?
- Which accuracy has the data used for calibration and validation?
- Based on which data can the model be validated?

If a model is encapsulated as a FMU, its complexity is not known in general if the source-code is not published. In the above-mentioned example, this might be the reason why the two models of largely different complexity are even assembled. The complexity can be estimated by simulating the FMU and regarding the output (e.g. the translation statistics of the tool). This additional effort could be avoided if a "scale" for the model complexity could be stored in the FMU.

3.3.2 Model detail levels

The detail level of a model can be roughly characterized by the following classification (adapted from (Kuhn, Otter, and Raulin 2008), which in turn is based on a classification sometimes used in aerospace industry):

- Level 1: Architectural level
  Steady-state power consumption.
  Models are described by *algebraic equations* based for example on the energy balance between ports (without dynamic response). Typical use: Rough system design with power budgets.

- Level 2: Functional level
  Steady-state power consumption and mean-value transient behavior (e.g. inrush current or consumption dynamics with regard to input voltage transients).
  Models are described by *differential-algebraic equations* (without switching elements). Typical use: Stability studies, controller design.

- Level 3: Behavioral level
  Detailed description of transient behavior (e.g. switching and high frequency injection behavior).
  Models are described by *hybrid differential-algebraic equations* with events and switching elements. Typical use: Network power quality studies, verification of controllers.

- Level 4: Distributed level
  Very detailed description of spatially distributed, transient behavior (e.g. magnetic field in electrical motor, stress field in a structure, flow around an airfoil).
  Models are described by *partial differential equations*, which are solved with FEM (Finite Element Method), FVM (Finite Volume Method), CFD (Computational Fluid Dynamics), or DEM (Discrete Element Method). Typical use: Detailed vibration investigations, design of structures or of the windings of electrical motors.

In Figure 3, the simulation of a DC/DC buck converter is shown for model levels 1, 2 and 3. In (Kuhn, Otter, and Raulin 2008), various alternatives are discussed how to implement multi-level models in Modelica.

3.3.3 Selecting the model detail

A typical modeling process starts with a simple model for one use case and then the complexity and number of use cases is increased step by step. In the end the model contains various physical effects and cover many use cases, but for the initially anticipated simple use case it's way too complex.

DOI
10.3384/ecp21181359

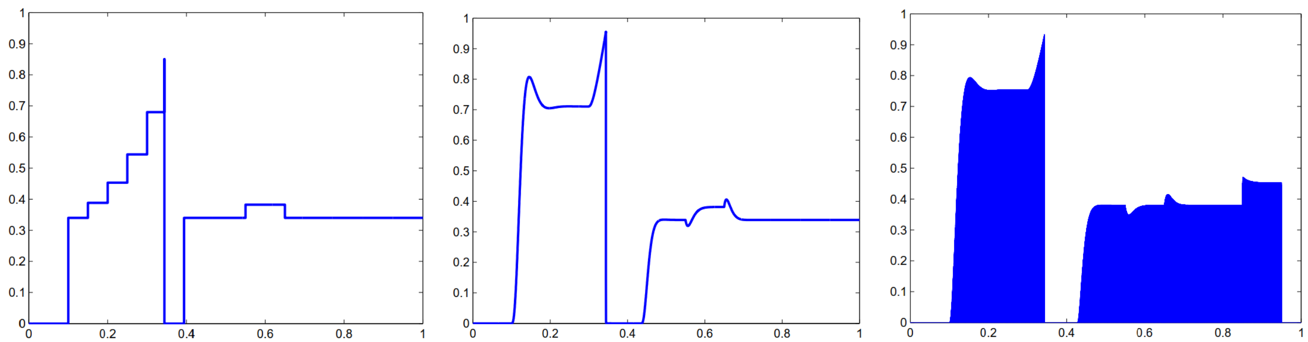

Figure 3. Exemplary simulation results of architectural (left), functional (middle) and behavioral (right) model of a DC/DC buck converter (Kuhn, Otter, and Raulin 2008). The right image contains high frequency switching leading visually to a "filled area".

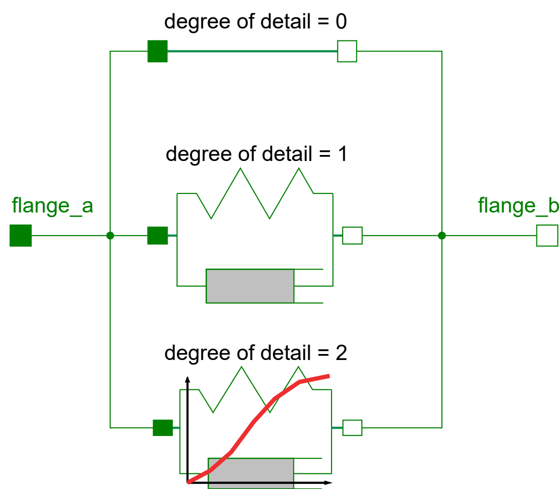

Figure 4. A Modelica model containing three conditional components reflecting different level of model's detail.

A good practice to avoid such over-engineering is to store development stages of different complexity and then choose a suitable complexity for a specific use case. The Modelica language provides the following possibilities to switch between different modeling levels:

- Partial models.
- Conditional components.
- Replaceable components.
- If-clauses.

This modeling practice also simplifies the documentation, because it's obvious to describe the differences between the levels of complexity, including the effects added to the model step by step. On the contrary, the necessary assumptions and simplifications can be described, to use one of the less detailed models.

In Figure 4, a simple example for conditional components is shown. Three different degrees of details are modeled: a rigid connection (architectural-level, cf. Section 3.3.2), a linear (functional-level) and a nonlinear (behavioral-level) spring-damper behavior between flange a and b. In this example, the particular degree of detail – "stiff", "linear" or "nonlinear" – can be selected by changing the value of the parameter `degree_of_detail`, as shown in Listing 3.

Listing 3. Conditional components

```
model ConditionalComponents
  ...
  parameter Integer degree_of_detail=0 "
      Parameter to switch between models
      with different degree of detail";

  Components.Detail_0 detail_0 if
      degree_of_detail == 0;
  Components.Detail_1 detail_1 if
      degree_of_detail == 1;
  Components.Detail_2 detail_2 if
      degree_of_detail == 2;

equation
  if degree_of_detail==0 then
    connect(flange_a, detail_0.flange_a);
    connect(detail_0.flange_b, flange_b);
  elseif degree_of_detail==1 then
    connect(flange_a, detail_1.flange_a);
    connect(detail_1.flange_b, flange_b);
  else
    ...
  end if;
  ...
end ConditionalComponents;
```

The selection of an appropriate modeling stage depends on the scenarios of the whole system. In Listing 3, the rigid connection can be used if the component, represented by this model, is very stiff in relation to other components in the system. In contrast, the linear spring behavior is not adequate, if the linear elastic range can be exceeded due to large forces appearing in the system.

Similarly, to the connect-statements inside Listing 3, if-clauses can be used to switch between different sets of equations, with the disadvantage, that each branch must have the same number of equations in Modelica. This may lead to many dummy definitions. Another option in Modelica are replaceable models. Unfortunately, they are only beneficial if the interfaces (inputs, outputs, parameters) of the different modeling stages are similar, because

they have to be adapted each time the replaceable model is changed.

A disadvantage of storing different levels of detail is the difficult testing strategy and higher effort for model adaptations. Functionality tests and model updates must be performed for each level. Therefore, it is advisable to store only major development steps of different complexity.

Typically, the level of detail should not only be switchable for a system model, but also for its sub-components. Technically, such an approach cannot be implemented fully satisfactory in the current Modelica language, because a feature to replace sub-components based on Boolean expressions is missing.

If an FMU should be built, the parametrized switch between different modeling stages (which is a structural parameter) cannot be transferred, because the number of state variables cannot be changed after the translation of the model. An FMU can only contain one single degree of complexity. It would be useful to support different levels of modeling details in a future Functional Mock-Up Interface (FMI) standard.

3.3.4 Physical versus data driven models

An important indicator for the credibility and the range of validity of a model is the modeling basis. Models can be based on physical equations or on data from measurements, expert guesses or other sources.

Physical equations are usually public knowledge with well defined assumptions. They usually cover a wide range of physical applications and can be extrapolated without immediately leaving their range of validity. Unfortunately, a proper parametrization is often difficult, because finding accurate parameter values (e.g. the friction coefficient between two bodies in contact) can be costly.

Data driven models are using for example measurement data. These data contain every single influence (e.g. the environment temperature) on the system during the measurement – even unknown ones. Consequently, the model is only valid for exactly these circumstances and, moreover, its extrapolation is limited. To put measurement data into a Modelica model so called Combi-Tables can be used. In this case the option "extrapolation triggers an error" of the Combi-Tables should be selected.

Characteristic maps based on measurement data also often induce a higher numerical effort for Modelica tools. This is because interpolation between the data points is necessary, non resolvable non-linearities can appear, and the variable described with the characteristic map can not be selected as a state-variable.

A model based on measurement data must contain all relevant information about the measurement (such as measured range, measurement methods and tools, measurement precision, ...) best in a formal way, and not just in the documentation.

Other kinds of data driven models can use mathematical approximations such as neural networks, response surfaces or optimization functions. Mathematical approximations are either used as a simplification of a physical model to reduce computation time or as an attempt to generalize measurement data by learning from the output of multiple measurements. There are several publications on this topic, also from the Modelica point of view (e.g. (Bruder and Mikelsons 2020) and (Tundis et al. 2017)).

3.4 Model Calibration

Once a model is implemented, the modeling process goes on with the calibration, validation and verification, to ensure that the model appropriately achieves the aims it was made for. Since the scope of model calibration, validation and verification is very large, we can only give here a short overview in the context of a typical use case for a multi-physical Modelica model. For a broader overview on the topic, a recent paper (Riedmaier et al. 2021) goes into more details and gives many additional references.

Multi-physical Modelica models are typically used either to represent a real physical system or to reproduce the behavior of such a system as a part of a model-based feed-forward or feed-back control system. Especially the direct generation of inverse models from Modelica models is a powerful feature which can also be used during the calibration process, see (Reiner 2011) or (Mesa-Moles et al. 2019).

3.4.1 Goal definition

The aim of the calibration process is to parameterize models with the help of measurements with regard to defined goals and criteria. For models, this generally means that they should map the behavior of the real-world system as precisely as possible. Figure 5 shows an overview of the model calibration process.

The calibration of the parameters of these models is important in order to achieve a good representation of the real system or to have a good controller performance, in the case of model-based control.

For many physical systems the knowledge of the individual involved parameters can be quite different. For example, for the model of a robot, the mechanical parameters such as link lengths or masses could be known very precisely from data-sheets or CAD data, whereas the friction or damping in connecting joints can be highly unknown. Well known parameters should, thus, be included in the models directly, in order to reduce the number of unknown parameters for the calibration process. Additionally, the source of these parameters should be documented.

The aim of the calibration for model-based controllers is an optimal control performance, as well as sufficient disturbance suppression and vibration damping (in the case of feed-back control). For Modelica models used as part of a control system, this often means that the model is calibrated directly on a HIL (Hardware-in-the-Loop) setup (Reiner 2011).

DOI
10.3384/ecp21181359

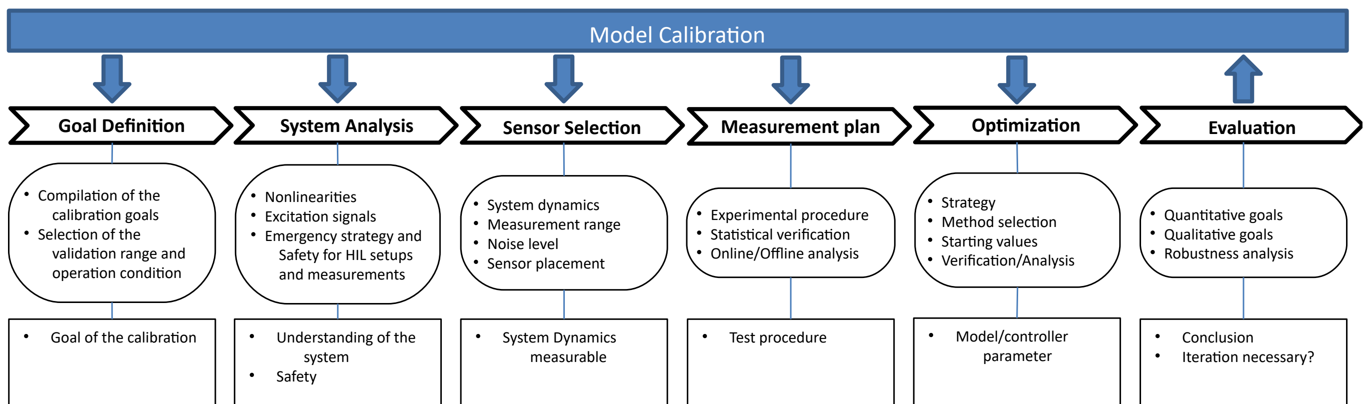

Figure 5. Overview for the Modelica model calibration process using measurement data.

3.4.2 System Analysis

There are numerous approaches for the identification process in the literature. In particular, however, a distinction must be made between methods for verification and identification in the frequency domain and in the time domain. Methods in the frequency domain are useful for systems that have (approximately) linear behavior, or for individual operating points of a system for which this assumption applies. Methods in the time domain are also suitable for non-linear systems, but are usually associated with greater effort (e.g. with regard to required computing time and experiment effort / duration).

Multi-physics Modelica models are usually used for modeling of complex non-linear systems. However, also the generation of linear systems from Modelica models is possible using numerical linearization, which is supported by many Modelica tools. Nonetheless, the focus is on nonlinear models in the following.

In a second step of the calibration the physical system has to be analyzed after the aim of the calibration has been be defined.

Normally, models of a physical system do not contain every detail or the entire operating range of the system, see Section 3.3.1. It must, therefore, be investigated to what extent an undesired or non-modeled behavior can be isolated. For the verification of model-based controllers on (HIL) test benches, further considerations must be made, such as taking precautions to ensure safe operation of the test bench in case of a controller failure (e.g. unstable behavior or violation of manipulated variable restrictions). In addition, suitable emergency strategies (e.g. emergency stop switch, mechanical emergency braking) must be implemented. If the controlled system is unstable without a controller, a robust parallel controller can also be helpful, which can be activated in the event of a fault and bypasses the controller to be verified.

3.4.3 Sensor selection

Suitable sensors have to be selected based on the calibration objectives. It is important to consider the dynamics and measuring ranges of the sensors used. If no single sensor can capture the entire relevant dynamics of the system, it must be examined whether a suitable result can be achieved by merging several measurements and / or sensors. Sensors generally have measurement noise and offsets, that must be considered during measurements and appropriately compensated / calibrated. This can also be done during a pre-processing step. After an analysis of the system, the sensor placement must be selected in such a way that the dynamics of the system is reproduced as clear as possible. This is especially important for elastic mechanical systems (e.g. eigenmode shapes). In case of doubt, different placements should be examined (in the case of elastic systems, the measurements can be influenced by vibration modes, for example).

3.4.4 Measurement plan

After suitable sensor selection and placement, a measurement plan should be documented, for which the statistical nature of measurements must also be considered. Critical measurements (e.g. measurements of parameters with a large impact on the system dynamics) must always be carried out several times and, if there is a broader spread of the measurement results, they must also be processed appropriately. Particularly in the case of large deviations, the sensor selection and placement must be critically examined again.

For the verification and tuning of controller parameters directly on the test bench, online methods are available, in which the evaluation takes place directly after or already during the measurements on the test bench. HIL setups are suitable for this, in which the Modelica-based model controller parameters can be changed with minimal effort.

Alternatively, however, offline methods can also be used in this case by first identifying suitable Modelica models of the controlled system in order to be able to design appropriate controllers with the help of simulations. For the identification of Modelica models and model-based observer systems of the physical system, an "offline" method can always be used, because the change of the Modelica models' parameters does not change the measurement data, which is, on the contrary, the case, if

the model is part of a feed-back system. This enables more elaborate methods to be used. Since Modelica models typically also do not represent the physical system up to very high frequency ranges, noisy measurement data should be low-pass filtered before the calibration process if possible using forward-backward filtering to avoid a phase shift in the data.

3.4.5 Optimization

Suitable identification strategies for Modelica models are optimization-based methods. Therefore, mathematical criteria are to be defined which can then be minimized using a suitable optimization algorithm by varying the parameters of the model. Appropriate start parameters have to be selected for this purpose. An important difference to classical optimization, however, is the statistical nature of the measurement results, which must be appropriately considered for the criteria specification. In the case of HIL optimization, mainly only optimization methods with a small number of steps (function evaluations) are to be used, since HIL experiments are usually significantly more time-consuming compared to pure numerical evaluations. This normally means that local optimization methods are to be preferred (e.g. gradient based methods or surrogate optimization techniques).

For offline methods, nearly all optimization algorithms can be used. However, because of the noise in the data, usually those algorithms converge better, which are gradient-free or robust against noise. For Modelica models, a wide range of optimization tools are available. There are methods for optimizing the Modelica model directly within Dymola using e.g. the DLR *Optimization* library (Pfeiffer 2012), as well as many other external tools. The latter use the Modelica model directly as an executable or exported as an FMU within a chosen environment, such as Python or Matlab, since the parameters of the Modelica models can still be changed, even after the compilation and export, see e.g. (Leimeister 2019).

3.4.6 Evaluation

After the Modelica model of the system has been verified the obtained results must be assessed quantitatively and qualitatively with regard to the selected objectives. To ensure robustness and to avoid over-fitting, additional measurements should be used for this step, which were not used within the optimization process of the parameters. If not all goals could be met during the calibration process, the process must be carried out again iteratively after an analysis of the results and, if necessary, a new modified model or controller structure must be used.

As a final result, the obtained set of model parameters should be documented, together with the model and the original measurement data, as well as a detailed description of the overall process. For Modelica models, such a documentation can be done directly within the model (see also sec. 3.2.1).

4 Model management

The modeling process described in Section 3 is a complex procedure with many steps and iterations. During the modeling process as well as afterwards, the different modeling stages should be traced. Therefore, a version and revision management is needed, as well as an archiving of measurement data, simulation results, etc.

4.1 Source code management using *git*

Modelica models and packages are stored in ASCII-files. It is therefore obvious to store these files in a source code management system, such as *GitHub*[18] or *GitLab*[19], which provide a lot of additional functionality compared to pure *git*[20]. Model developers can then profit from issue tracking, merge/pull requests, release handling, etc. One disadvantage of these widespread source code management systems is that access rights can be only defined for a whole repository. In industrial projects, it is typically not desired that everyone has access to the whole information and it is then necessary to split the information over several repositories, for example:

- A repository contains the source code of the Modelica model or the Modelica package. Only model developers have access to this repository.
- A repository contains the released versions (possibly with encrypted Modelica packages), as well as an issue tracker. Users of the model library have access to this repository.
- A repository contains administrative information, such as contracts, clearance, license information. Managers have access to this repository.

Data to parametrize the models is often *not* stored in a source code management system but is extracted from database systems. Simulation results and measurement data is often stored in binary format and, therefore, a source code management system is not well suited. Instead, this data is typically stored at different locations on a pure file system without version management. Altogether this means that the information about a model might be spread across several locations. Consequently, it is advisable to maintain a document with information about the different storage locations. Regarding the model credibility, there is room for improvement.

Due to its textual nature, any change in the Modelica model can be traced, read and understood by humans. But, keeping minimal textual differences is a hard job for Modelica tools, especially when users mix graphical and textual modeling. As the formatting of Modelica code is not specified, the problem even increases if different Modelica tools are used within one project. To our experience, it

[18] https://github.com/
[19] https://about.gitlab.com/
[20] https://git-scm.com/

DOI
10.3384/ecp21181359

is a significant early step to establish implementation rules for a project collaboration in order to keep model changes traceable. A public starting point for these modeling conventions is `Modelica.UsersGuide.Conventions`[21]. Another best practice concerning Modelica code management is to seperate git commits for larger graphical and documentation changes and for actual changes to the model (like introducing new components or changing or adding equations). Furthermore, all auto-formatting in tools should be switched off.

4.2 Version management

While source-code management of Modelica models works fine within one organization, most of the time this important information is lost when delivering models across organizational boarders, e.g. from supplier to OEM. Modelica has multiple ways of storing the current revision information in annotations e.g. the `revisions` and/or `revisionID`. See an example of MSL 4.0.0 in Listing 4. One important limitation is that `revisionId` and `dateModified` are usually stored only on the top level package, so it cannot be used to see when an actual class has been changed. For specific classes, one has to rely on the non-formalized `revisions` annotation. Conventions for how to structure and update the `revisions` annotation are library specific, so far. The `revisionId` is not automatically handled by most Modelica tools and it is typically lost when generating an FMU based on Modelica code.

While Modelica tools can give support on versioning and releases, a library release still implies coordination between multiple team members. One public resource for release workflows is the *Modelica Standard Library Developers Wiki*[22]

4.3 Model resources

By convention, non-Modelica language information, such as manufacturer data sheets, data-files for CombiTables, images, is stored in folder *Resources* which is located at the top level directory of a library. It might be helpful to distinguish – also in the structure of the Resources folder – between *model-data* (parametrization data, e.g. for CombiTables) and *documentation-data* (e.g. example results or information used for modeling). Model-data are necessary for the functionality of the model, while documentation-data is “only” nice to have, but can be extremely helpful for the comprehension of the model. There is a smooth transition between model-data and documentation-data. For example, icon-graphics is irrelevant for the functionality of a library, but improves the user’s handling significantly by giving a quick graphical impression of the component.

As already mentioned in Section 4.1, larger measurement databases are usually not stored within the Modelica library. But, in order to understand the quality of calibrated models, we propose to store a minimal set of measurement data, actually used for parameter optimization within the Resources folder as documentation-data.

While the file structure of Modelica code is automatically updated by a Modelica tool on the file system, the central resources folder has to be organized manually. This is error-prone and, depending on the chosen structure, hard to update when e.g. re-structuring the Modelica package. External links to larger measurement data sources is not handled by the current Modelica package concept and local resolving of the `loadResource()`[23] function.

In order to be able to extract working system models out of a larger library, Modelica tools are typically able to store a so-called *total model*, including all required resources. The storage of total models is not standardized in Modelica, yet. The `loadResource()` function in combination with a Uniform Resource Identifier (URI) helps to avoid path issues and allows a Modelica tool to analyze which resources are needed. As models can access files in multiple ways, it still remains a tedious task to check e.g. for missing resources or too much files to be copied.

4.4 Archiving of simulation results

Simulation results are usually stored in a file, in binary or ASCII format. This includes reference data as input for a model, as well as reference results that a simulation should reproduce within some tolerance. Various result file formats are used in the Modelica community, but there is no satisfactory, standardized solution.

When results need to be exchanged, such as reference results of the Modelica Standard Library, or of FMUs, they are often stored in CSV format[24] due to its widespread support in tools. Hereby, the result is seen as a table, where every column has a name (optionally with "."s to mark hierarchical structures or "[..]" to mark elements of an array) and represents a time series. The first column contains the monotonically increasing value of the independent variable (usually `Time`). A discontinuity is signaled by two identical time instants. For an example see listing 5.

Listing 5. Example of a CSV file with time event at 0.1 s

```
Time, control.w_ref, motor.w, on[3]
0.0,     0.0       ,    0.0,  0
0.1,     0.0       ,    0.0,  0
0.1,     1.0       ,    0.0,  1
0.2,     1.0       ,    0.1,  1
0.3,     1.0       ,    0.2,  1
```

The essential advantage of this format is its simplicity, but there are numerous drawbacks. Especially, it is not suited for large data sets as needed to archive simulation results.

[21] https://doc.modelica.org/Modelica%203.2.3/Resources/-helpDymola/Modelica_UsersGuide_Conventions.html

[22] https://github.com/modelica/ModelicaStandardLibrary/wiki

[23] Modelica.Utilities.Files.loadResource: https://doc.modelica.org/Modelica%204.0.0/Resources/helpDymola/-Modelica_Utilities_Files.html#Modelica.Utilities.Files.loadResource

[24] https://en.wikipedia.org/wiki/Comma-separated_values

Listing 4. Annotations relevant to versioning using the example of the Modelica Standard Library

```
package Modelica

// Sub-packages removed

annotation (
  version="4.0.0",
  versionDate="2020-06-04",
  dateModified = "2020-06-04 11:00:00Z",
  revisionId="6626538a2 2020-06-04 19:56:34 +0200",
  uses(Complex(version="4.0.0"), ModelicaServices(version="4.0.0")),
  Dymola(checkSum="469888163:3572996634"),
  conversion(
    from(
      version={"3.0", "3.0.1", "3.1", "3.2", "3.2.1", "3.2.2", "3.2.3"},
      script="modelica://Modelica/Resources/Scripts/Conversion/ConvertModelica_from_3.2.3
        _to_4.0.0.mos")),
      ...);
end Modelica;
```

For larger data sets often the *dsres* (dynamic system results) storage format is used that was developed around 1996 by Martin Otter and used by Dymola. It was later also used by OpenModelica[25]. There are several importing and exporting scripts available, especially for Matlab and for Python[26]. The dsres-format consists of a set of matrices that are either stored in MATLAB MAT v4 binary format or in a textual format. The logical view is:

1. String vector **name** contains the names of the signals. An index i of this vector characterizes the corresponding signal i.

2. String vector **description** contains a description text for the signal, typically with its unit.

3. Integer matrix **dataInfo** contains information where and how a signal is stored: A signal i is stored in a matrix j in column k with an interpolation type l and an extrapolation type m. If k is negative, column $|k|$ has to be multiplied with -1.

4. The core data is stored in **data_j** matrices where every column of a matrix contains the time series of one signal. The first column is the independent variable. Different matrices can have different time axes, that is, different number of rows. Typically, two data matrices are present: One matrix with two rows, that stores the parameters as time series with two time points, and one matrix with the time-varying signal data that corresponds to the data stored in CSV file format.

Due to the connector definition, a Modelica model has typically many variables that are identical or have opposite sign. The time series of these signals are stored in a compact way with the dsres-format, because the actual time series of variables that are related by equations $v_1 = v_2 = -v_3 = -v_4 = ...$ are stored in *one* column of a data matrix. If all variables of a Modelica model are stored in a result file, then often the size of the file is reduced by a factor of 4-5 by this technique. Furthermore, the second data matrix is stored in such a way, that the binary result file can be recovered, even if a simulation run crashes during integration.

There had been a few attempts to define a standardized time series file format based on HDF5[27], an open source file format that supports large, complex, heterogeneous data and meta information stored hierarchically in one binary file: In particular, the MTSF format (Modelica Association Time Series File Format) (Pfeiffer, Bausch-Gall, and Otter 2012) and the SDF format (Scientific Data Format)[28]. In (Pfeiffer, Bausch-Gall, and Otter 2012) it is reported that simulation results up to 200 Gbyte could be stored and retrieved in HDF5 format on file. Although, HDF5 looks attractive for scientific data sets and especially simulation result data, it has severe drawbacks, especially because it is complex and not suited for today's cloud-services. For a more detailed discussion, see (Tiller and Harman 2014).

A more modern design is the recon[29] format developed by (Tiller and Harman 2014): Simulation results and meta data are stored in a network friendly way using the JSON format[30] where the core data is packed with msgpack[31].

4.4.1 Result meta data

The result format should store more than just numeric values. In order to interpret the results correctly, the following additional information seems especially valuable:

[25] https://openmodelica.org/doc/OpenModelicaUsersGuide-/latest/technical_details.html#the-matv4-result-file-format

[26] https://github.com/jraedler/DyMat/ https://github.com/kdavies4/ModelicaRes/

[27] https://www.hdfgroup.org/solutions/hdf5/

[28] https://github.com/ScientificDataFormat

[29] github.com/xogeny/recon

[30] https://www.json.org/json-en.html

[31] https://msgpack.org/

DOI 10.3384/ecp21181359

- Which simulation tool and of which version generated the result? Which simulation settings have been used?
- Which system model, which validated (sub-)models and which boundary conditions/scenarios were used to produce the simulation results?
- What are the interesting variables for analysis? In large Modelica result files, finding relevant variables for analysis might be difficult. Two possible ways to improve this situation could be:
  1) Predefined plots – based on figure annotations of MLSv35 (Modelica Association 2021) – could be also available in the result format.
  2) The file modelDescription.xml of FMI defines the system model interface – inputs, outputs and parameters – in a re-usable, standardized way. This interface information could also be available in result formats.
- What is the expected accuracy of the simulation? The upcoming FMI for embedded systems (eFMI) specification[32] (Lenord et al. 2021) defines tolerances.

When generating reference results for the MSL, some of this meta data is stored in a separate file creation.txt, see an example in the MSL[33].

The *SSP Traceability Specification*[34] is currently under development within the Modelica Association Project "System Structure and Parameterization of Components for Virtual System Design"[35]. The approach is based on a so-called *glue particle*, an XML file providing a consistent data schema along the simulation process. For specifics of the proposed file format, see *Simulation Task Meta Data* in file STMD.xsd. If this concept would be applied to Modelica and supported by Modelica tools, it could save a lot of today's manual documentation work.

4.4.2 Results for post-processing

Most result formats are designed in a way, that the Modelica tool can conveniently write those files during simulation. For archiving purposes, it is also important to retain the readability of the files even when the source tool is no longer available. Moreover, post-processing should also be possible in any external tool, like Excel, Julia, Matlab, Python, etc. If a standardized result file format would be available, both issues could be solved by standardized reading procedures for different languages.

5 Conclusions and outlook

The paper summarizes current challenges in Modelica-related continuous development processes with regards to modeling, simulation, data management and calibration/verification. A particular focus was given on the development and handling of Modelica models and libraries.

Several aspects have been identified that need to be improved to arrive at a reliable process for the development of credible models and digital twins based on coherent Modelica Association standards (Modelica language, FMI, SSP, DCP, eFMI)[36]. The upcoming eFMI standard (Lenord et al. 2021) goes already in the right direction by including tolerance-defined reference results in an eFMI model to support automatic tests and verification of generated production code. The glue particle approach in the SSP-project might be used for all MA standards so that tool chains from Modelica models to FMI, SSP, DCP, and eFMI components are completely traceable and no information is lost. Furthermore, additional information needs to be added to define the domain of validity of a model. It might also be necessary to add further quality measures. Simulation results need to be stored in a standardized way both for exchange between tools, as well as for archiving purposes. The recon format with glue particle information included might be considered for all Modelica Association standards and reference files.

As an overall target, the UPSIM project description states: "Enable companies to safely collaborate with simulations, in a repeatable, reliable, and robust manner, and for implementing simulation in a Credible Digital Twin setting as a strategic capability to become an important factor in quality, cost, time-to-market, and overall competitiveness." This view could become a guideline for further development of the Modelica Association standards.

Acknowledgements

This work has been partly supported by the European ITEA3 Call6 project UPSIM[37] – Unleash Potentials in Simulation (number 19006). The work was funded by the German Federal Ministry of Education and Research (BMBF, grant numbers 01IS20072H and 01IS20072G).

We would like to thank the reviewers of this paper as well as Tobias Bellmann for quite a lot of constructive improvement proposals. We would also like to thank Thomas Alpögger, Daniel Bouskela, Andreas Junghanns, Martin Krammer, Lars Mikelsons, Mugur Tatar for their comments on some details that we have taken into account.

References

Alistair, Mavin Mav and Philip Wilkinson (2019). "Ten Years of EARS". In: *IEEE Software* 36.5, pp. 10–14. DOI: 10.1109/MS.2019.2921164.

Bals, J., W. Fichter, and M. Surauer (1997). "Optimization of magnetic attitude- and angular momentum control for low earth orbit satellites". In: *Proceedings Third International Conference on Spacecraft Guidance, Navigation and Control Systems 1996*. ESTEC, Noordwijk, The Netherlands. URL:

[32] https://emphysis.github.io/pages/downloads/efmi_specification_1.0.0-alpha.4.html#definition-of-csv-data

[33] https://github.com/modelica/MAP-LIB_ReferenceResults/blob/v4.0.0/Modelica/Blocks/Examples/PID_Controller/creation.txt

[34] github.com/PMSFIT/SSPTraceability

[35] ssp-standard.org

[36] https://modelica.org/

[37] https://itea3.org/project/upsim.html

https://ui.adsabs.harvard.edu/link_gateway/1997ESASP.381..559B/ADS_PDF.

Bouskela, Daniel, Alberto Falcone, et al. (2021). "Formal Requirements Modeling for Cyber-Physical Systems Engineering: an integrated solution based on FORM-L and Modelica". In: *Requirements Engineering - accepted for publication*.

Bouskela, Daniel and Audrey Jardin (2018). "ETL: A New Temporal Language for the Verification of Cyberphysical Systems". In: *2018 Annual IEEE International Systems Conference (SysCon)*. URL: https://ieeexplore.ieee.org/document/8369502.

Bruder, Frederic and Lars Mikelsons (2020). "Towards Grey Box Modeling in Modelica". In: *Kuo CH., Lin PC., Essomba T., Chen GC. (eds) Robotics and Mechatronics. ISRM 2019. Mechanisms and Machine Science, vol 78*. DOI: 10.1007/978-3-030-30036-4_17.

Buse, Fabian and Tobias Bellmann (2021). "General Purpose Lua Interpreter for Modelica". In: *Proceedings of the 14th International Modelica Conference*.

Corso, Antony et al. (2020). "A Survey of Algorithms for Black-Box Safety Validation". In: *arXiv:2005.02979*. URL: https://arxiv.org/abs/2005.02979.

Dempsey, M. et al. (2006). "Coordinated Automotive Libraries for Vehicle System Modelling". In: *5th International Modelica Conference*. Vienna, Austria, pp. 33–41. URL: https://modelica.org/events/modelica2006/Proceedings/sessions/Session1b2.pdf.

Department of Defense (2016). *Aircraft Electric Power Characteristics (MIL-STD-704F_CHG-1)*. Tech. rep. URL: http://everyspec.com/MIL-STD/MIL-STD-0700-0799/MIL-STD-704F_CHG-1_55461/.

Elmqvist, Hilding et al. (2021). "Modia - Equation Based Modeling and Domain Specific Algorithms". In: *Proceedings of the 14th International Modelica Conference*.

Heinkel, Hans-Martin and Kim Steinkirchner (2021). *Credible Simulation Process*. Tech. rep. Robert Bosch GmbH and PROSTEP AG. URL: https://setlevel.de/neuigkeiten/credible-simulation-process.

Joos, Hans-Dieter (2015). "Application of Optimization-Based Worst Case Analysis to Control Law Assessment in Aerospace". In: *Advances in Aerospace Guidance, Navigation and Control*. DOI: 10.1007/978-3-319-17518-8_4.

König, Christian et al. (2020). "Traceability in the Model-based Design of Cyber-Physical Systems". In: *Proceedings of the American Modelica Conference 2020*. Boulder, USA, pp. 168–178. DOI: 10.3384/ecp20169168.

Kuhn, Martin R., Martin Otter, and Loic Raulin (2008). "A Multi Level Approach for Aircraft Electrical Systems Design". In: *6th International Modelica Conference*. Bielefeld, Germany, pp. 95–101. URL: https://modelica.org/events/modelica2008/Proceedings/sessions/session1d1.pdf.

Labusch, Andreas et al. (2014). "Worst Case Braking Trajectories for Robotic Motion Simulators". In: *IEEE International Conference on Robotics & Automation (ICRA)*. Hong Kong, China. DOI: 10.1109/ICRA.2014.6907333.

Leimeister, Mareike (2019). "Python-Modelica Framework for Automated Simulation and Optimization". In: *Proceedings of the 13th International Modelica Conference*. DOI: 10.3384/ecp1915751.

Lenord, Oliver et al. (2021). "eFMI: An open standard for physical models in embedded software". In: *Proceedings of the 14th International Modelica Conference*.

Mesa-Moles, L. et al. (2019). "Robust Calibration of Complex ThermosysPro Models using Data Assimilation Techniques: Application on the Secondary System of a Pressurized Water Reactor". In: *Proceedings of the 13th International Modelica Conference*. DOI: 10.3384/ecp19157553.

Modelica Association (2021-02). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.5*. Tech. rep. Linköping: Modelica Association. URL: https://specification.modelica.org/maint/3.5/MLS.pdf.

Otter, Martin et al. (2015). "Formal Requirements Modeling for Simulation-Based Verification". In: *11th International Modelica Conference*. Versailles, France, pp. 625–635. DOI: 10.3384/ecp15118625.

Pfeiffer, Andreas (2012). "Optimization Library for Interactive Multi-Criteria Optimization Tasks". In: *9th International Modelica Conference*. Munich, Germany, pp. 669–680. DOI: 10.3384/ecp12076669.

Pfeiffer, Andreas, Ingrid Bausch-Gall, and Martin Otter (2012). "Proposal for a Standard Time Series File Format in HDF5". In: *9th International Modelica Conference*. Munich, Germany, pp. 495–505. DOI: 10.3384/ecp12076495.

Reiner, M. (2011). "Modellierung und Steuerung von strukturelastischen Robotern". PhD thesis. Technische Universität München, Fakultät für Maschinenwesen.

Riedmaier, S. et al. (2021). "Unified Framework and Survey for Model Verification, Validation and Uncertainty Quantification". In: *Archives of Computational Methods in Engineering 28*, pp. 2655–2688. DOI: 10.1007/s11831-020-09473-7.

Schamai, Wladimir (2013). "Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool". PhD thesis. University of Linköping. URL: http://liu.divaportal.org/smash/record.jsf?pid=diva2:654890.

Tatar, Mugur and Jakob Mauss (2014). "Systematic Test and Validation of Complex Embedded Systems". In: *ERTS 2014 - Embedded Real Time Software and Systems*. Toulouse, France. URL: https://www.researchgate.net/publication/259871632_Systematic_Test_and_Validation_of_Complex_Embedded_Systems/.

Thuy, Nguyen (2014). "FORM-L: A Modelica Extension for Properties Modelling Illustrated on a Practical Example". In: *10th International Modelica Conference*. Lund, Sweden, pp. 1227–1236. DOI: 10.3384/ecp140961227.

Tiller, Michael and Peter Harman (2014). "recon – Web and network friendly simulation data formats". In: *Proceedings of the 10th International Modelica Conference*. DOI: 10.3384/ecp140961081.

Tundis, Andrea et al. (2017). "Model-Based Dependability Analysis of Physical Systems with Modelica". In: *Hindawi, Modelling and Simulation in Engineering, Volume 2017*. DOI: 10.1155/2017/1578043.

Wei, Wang et al. (2016). "Vibration performance analysis of vehicle with the non-pneumatic new mechanical elastic wheel in the impulse input experiment". In: *Journal of Vibroengineering, Vol. 18, Issue 6, 2016*. DOI: 10.21595/jve.2016.16988.

Zimmer, Dirk, Martin Otter, and Elmqvist (2014). "Custom Annotations: Handling Meta-Information in Modelica". In: *10th International Modelica Conference*. Lund, Sweden, pp. 174–182. DOI: 10.3384/ecp14096173.

DOI 10.3384/ecp21181359