

American

Conference


Modelica & fmi

Atlanta USA 2026

October 12-14



**Georgia Institute
of Technology**


Modelica
Language

PROCEEDINGS OF THE AMERICAN MODELICA & FMI CONFERENCE 2026

Georgia Institute of Technology, Atlanta, USA, October 12-14, 2026

EDITORS

Michael Tiller, Hubertus Tummescheit, Luigi Vanfretti, Christopher Laughman, and Michael Wetter

PUBLISHED BY

Modelica Association and Linköping University Electronic Press

Series info: Linköping Electronic Conference Proceedings Nr. 219

ISBN: 978-91-8118-519-5 (PDF)

eISSN: 1650-3740

DOI: 10.3384/ecp219

ORGANIZED BY

North America Modelica Users' Group: na.modelica-users.org

in co-operation with:

Modelica Association c/o PELAB, Linköpings Univ.

SE-581 83 Linköping

Sweden

CONFERENCE LOCATION

Aerospace Systems Design Laboratory (ASDL)

Georgia Tech

275 Ferst Dr NW #3

Atlanta, GA 30332

USA

Copyright © Modelica Association, 2026



Oliver Lenord

Research and Development Engineer,
Leader of OpenScaling project
Affiliation: Bosch Corporate Research

BIO

Oliver Lenord earned his Phd at the Mechatronics Institute of the Univ. of Duisburg, Germany. Since 2016 he is affiliated with Robert Bosch - Corporate Research. Formerly he led the simulation software development group at Bosch Rexroth and worked as product manager for Siemens PLM in California, USA. Currently he is leading the European ITEA4 project OpenSCALING (openscaling.org). Earlier he also initiated and led the German research project PHyMoS, concerned with integrating scientific machine learning with classical modeling and led the award winning ITEA3 project EMPHYSIS that delivered the new eFMI standard. He also serves as vice-chairman of the Open Source Modelica Consortium (OSMC).

ABSTRACT

The scalability of virtual engineering has become an increasingly critical demand driven by the innovation pressure towards carbon neutral, energy-efficient and cost-effective solutions. Open standards like Modelica, FMI, eFMI and SSP play a major role in making system simulation accessible to a broad user base, fostering a rich ecosystem of research and technology development across academia and industry. The European publicly funded project OpenSCALING (Open Standards for SCALable Virtual EngineeriNG) has further strengthened this ecosystem through standard enhancements and new methods, enabling simulation and credible processes to scale to the next level. This talk will highlight key project achievements in the fields of:

- Modelica compiler enhancements
- Integration of scientific machine learning with Modelica
- Uncertainty quantification and credible workflows

Reference to newly developed methods and standard enhancements are given including array-preserving compilation, pre-compiled components, and improved support of sensitivities in FMI. In addition the newly proposed layered standards ls-sa (for sensitivity analysis) and ls-uq (for uncertainty quantification) are briefly introduced.

Benchmarks and examples will illustrate these advances, demonstrating the ability to:

- Increase maximum model size by orders of magnitude
- Drastically reduce compile and simulation start-up times
- Accelerate simulations by orders of magnitude using surrogate modeling and machine learning
- Integrate and efficiently process Artificial Neural Networks within Modelica
- Leverage FMI in diverse machine learning environments for robust model training
- Enable traceable credibility workflows through standardized metadata in FMI and SSP
- The practical relevance of these advances will be demonstrated referring to selected industrial use cases from different application domains.

Finally, an outlook will be given on the future potential of agentic AI for Modelica and the proposed Base Modelica language.



Daniel Mikkelson

Integrated Energy Modeling and
Simulation Engineer

BIO

Dr. Daniel Mikkelson is an integrated energy modeling and simulation engineer at Idaho National Laboratory. His research interests include systems modeling and analysis of advanced nuclear reactor systems, thermal energy storage, experimental systems, and integrated energy systems leveraging nuclear power in electric and non-electric applications. Analyses have investigated control strategy, capacity optimization, dispatch scheduling, and evaluation of integrated system feedback. Dr. Mikkelson is the lead developer of HYBRID, the Modelica modeling repository of the Integrated Energy Systems program at INL. Dr. Mikkelson received his Ph.D. in Nuclear Engineering from North Carolina State University in 2021.

ABSTRACT

Nuclear Systems Modeling for Integrated Energy Systems Analyses

The HYBRID modeling repository has been developed to analyze the control, dispatch, and feedback of integrated energy systems that leverage nuclear power for thermal and electrical applications. As global priorities endorse nuclear energy, enabling rapid, open-source, dynamic evaluation of potential nuclear applications can enable faster development of integrated systems that will improve the utilization of nuclear energy in diverse applications for both brownfield and greenfield integrations. An overview of the nuclear approach to modeling and simulation will present areas where Modelica presents an ideal tool to investigate configuration viability in dynamic deployment scenarios.

CONFERENCE CO-CHAIRS

Dr. Michael Tiller, JuliaHub
 Dr. Hubertus Tummescheit, Model Based
 Innovation LLC

PROGRAM CO-CHAIRS

Prof. Luigi Vanfretti, RPI
 Dr. Michael Wetter,
 Lawrence Berkeley National Laboratory

CONFERENCE EXECUTIVE COORDINATOR

Dr. Christopher Laughman,
 Mitsubishi Electric Research Laboratories

LOCAL CONFERENCE CHAIR

Prof. Dimitri Mavris, Georgia Inst. of Technology

LOCAL CONFERENCE COORDINATOR

Megan Scheidt, Georgia Inst. of Technology

MEMBER

Dr. Behnam Afsharpoya,
 Dassault Systèmes

PROGRAM COMMITTEE

Bernhard Bachmann, Fachhochschule Bielefeld
John Batteh, JuliaHub
Albert Benveniste, INRIA
Christian Bertsch, Robert Bosch GmbH
Scott Bortoff, Mitsubishi Electric Research Laboratories
Robert Braun, Linköping University
Johannes Brunnemann, XRG Simulation GmbH
Dag Brück, Dassault Systèmes AB
Benoît Caillaud, IRISA / INRIA Rennes
Yan Chen, Pacific Northwest National Lab
Clément Coïc, Siemens Healthineers AG
Olaf Enge-Rosenblatt, Fraunhofer
Gianni Ferretti, Politecnico di Milano
Leo Gall, LTX Simulation GmbH
Rui Gao, Nanjing Yuansi SimTek Co., Ltd.
Dan Henriksson, Dassault Systèmes
Yutaka Hirano, Woven Planet Holdings, Inc.
Daeoh Kang, IVH
Christopher Laughman, MERL
Oliver Lenord, Robert Bosch GmbH
Alessandro Maccarini, Aalborg University
Pierre Mai, PMSF IT Consulting Pierre R. Mai
Lars Mikelsons, University of Augsburg
Ulf Christian Müller, Lucerne University of Applied Sciences
 and Arts
Thierry Stephane Noudui, The United African University of
 Tanzania
Maurizio Palmieri, University of Pisa
Erika Pellegrino, Leonardo
Kaustubh Phalak, Ingersoll Rand
Adrian Pop, Linköping University
Clemens Schlegel, Schlegel Simulation GmbH
Klaus Schuch, AVL List GmbH
Michael Sielemann, Modelon Deutschland GmbH
Martin Sjölund, Linköping University
Wilhelm Tegethoff, TLK-Thermo GmbH
Michael Tiller, JuliaHub
Hubertus Tummescheit, Model Based Innovation LLC
Alfonso Urquia, Universidad Nacional de Educación a Distancia
 (UNED)
Luigi Vanfretti, Rensselaer Polytechnic Institute
Antoine Viel, Siemens DISW
Volker Waurich, TU Dresden
Michael Wetter, Lawrence Berkeley National Laboratory
Stefan Wischhusen, XRG Simulation GmbH
Dirk Zimmer, DLR

Towards an FMI Layered Standard for DAE Applications for Simulation and Optimization	9
A Modelica-Based Cyber-Physical Simulation Architecture for Firmware-in-the-Loop Testing of Robotic Vehicles	21
Implementation of a High Fidelity Model of a Racecar into a Path Optimization Problem using Functional Mock-Up Units	30
Terminals in SSP	41
The Modelica Buildings Requirements Library for In-Simulation Requirements Verification for HVAC Applications	48
A Workflow for Transparent and Replicable Modelica Implementations of ASHRAE Guideline 36 Controls for Chiller Plants with Waterside Economizers	57
A Unified Multi-Fidelity Modelica Library for Desiccant Wheel Simulation	68
A Templated Modelica Model for Decoupling Physical System and Control in Resilient Community Energy Systems	78
Rumoca: Modelica as a Universal Algebraic Frontend via a Rust-Native Compiler	87
Differentiable Parameter Optimization for DAEs with State-Dependent Events	99
ModelScript: Rethinking the Digital Thread with a Web-Native, Polyglot Compiler for Modelica, SysMLv2 & Beyond	108
MMChvdcLibrary: Modeling Arm Average Value Modular Multilevel Converters in Modelica for Power System Simulation	118
Real-time Simulation and CHiL Testing of Power System Stabilizers on Microcontrollers with Modelica and eFMI	128
Deploying OpenIPSL Models into DlgSILENT PowerFactory via FMI Model Exchange	153

Discontinuity - Aware Solvers for Vapor - Compression Cycles	159
Toward Equation-Based Building Energy System Modeling in Julia: A Modular Prototype and Verification Study	170
Implementation of OpenIPSL in Dyad	180
A Single-Element Multi-Point Compliance for Rigid-Body Connections	189
A 2-DOF MultiBody Track Joint for Race Car Optimization	196
A Singularity Avoiding Track Model for Race Car Optimization	205
Five-Link Rear Suspension for F1: A Two-Stage Agentic AI Study with the Vehicle Dynamics Library	215
A Modelica-based Virtual Testbed for Hybrid-Cooled Data Centers	226
Integrated Microclimate Model in Modelica for Data Center Chiller Farms	234
Agent-Assisted Construction of Modelica System Models from Validated Components: A Data Center Cooling Case Study	242
A Unified Dynamic System Testing Framework	251
Data-Driven Refinement of a Native Modelica Model Predictive Controller for Multi-Variable Gas Turbine Hydrogen Blending Control	258
Quantum Algorithms for Patent Portfolio Analysis Using OpenModelica and IBM Qiskit	276

PAPERS



Towards an FMI Layered Standard for DAE

Applications for Simulation and Optimization

Elmir Nahodovic ¹ Andreas Heuermann ² Joel A. E. Andersson ³ Adwait Verulkar ⁴
 Srikanth Sivaramakrishnan ⁴ Masoud Najafi ⁵ Linus Langenkamp ⁶ Christian Bertsch ⁷
 Torsten Sommer ¹ Erik Henningsson ¹ Hans Olsson ¹ Bernhard Bachmann ⁶

¹Dassault Systèmes, Sweden, {elmir.nahodovic, torsten.sommer, erik.henningsson, hans.olsson}@3ds.com

²Santa Anna IT Research Institute, Sweden, andreas.heuermann@santa-anna.se

³FMIOPT AS, Norway, joel@fmiopt.com

⁴General Motors, USA, {adwait.verulkar, srikanth.sivaramakrishnan}@gm.com

⁵Siemens Industry Software, France, masoud.najafi@siemens.com

⁶Bielefeld University of Applied Sciences and Arts, Germany, {linus.langenkamp, bernhard.bachmann}@hsbi.de

⁷Bosch Research, Germany, christian.bertsch@de.bosch.com

Abstract

The Functional Mock-up Interface (FMI) 3.0 standard for Model Exchange is restricted to hybrid ordinary differential equations, requiring any internal algebraic equations to be solved inside the Functional Mock-up Unit (FMU) before derivatives are returned to the importer. For models originating from, e.g. Modelica, this means that non-linear algebraic equations must be solved through internal Newton iterations, which can reduce accuracy, increase computational cost, introduce hidden solver states, and cause robustness issues in downstream simulation and optimization workflows. In this article, we present a proposal for a layered standard, *fmi-ls-dae*, that exposes algebraic equations and their associated algebraic variables as part of a semi-explicit index-1 differential-algebraic equation. We describe the proposed extensions to the FMI XML schema and demonstrate the approach through prototype implementations: Dymola and CasADi generate FMUs that expose this semi-explicit index-1 formulation, while CasADi, FMIOPT, Simcenter Twin Activate, fmusim, and MOO (the dynamic optimization tool of OpenModelica) import them for simulation and dynamic optimization. On an industrially relevant multilink suspension corner model, the proposed DAE-FMU formulation enables the optimization routine to converge on an optimal control problem on which the equivalent ODE-FMU fails to converge. We outline ongoing work towards consistent initialization, event handling and future support for higher-index DAEs.

Keywords: *FMI, Modelica, DAE, Differential Algebraic Equations, Layered Standard, Model Exchange, Dynamic Optimization*

1 Introduction

The Functional Mock-up Interface (FMI) version 3.0 for Model Exchange (Modelica Association 2024) provides a standardized interface for models of dynamical systems in the form of Ordinary Differential Equations (ODEs) with state-machine event dynamics, often referred to as hybrid ODEs. FMI 3.0 for Model Exchange allows an importer to simulate the model by supplying time, states, and inputs, and retrieving state derivatives and outputs. However, this ODE-based interface imposes a restriction on the class of models that can be represented: any internal algebraic equations must be solved entirely inside the FMU before state derivatives are returned to the importer.

In Modelica, models are inherently described by Differential Algebraic Equations (DAEs). During code generation, a Modelica tool performs symbolic transformations with the aim of reducing the high-index DAE (Modelica Association 2023). Usually it is reduced to an index-1 DAE, which is then typically converted into an ODE by solving the algebraic equations at each simulation step (Cellier and Kofman 2006). Some tools, such as OpenModelica and Dymola, also support a *DAE-Mode*, where the semi-explicit index-1 DAE is instead passed directly to an appropriate implicit solver such as IDA or Radau IIA. Rather than being solved separately, the non-linear equations are then solved along with the non-linear equations of the implicit integrator method, which can improve simulation time of Modelica models containing large systems of non-linear equations (Braun et al. 2017; Henningsson et al. 2019). However, such a DAE simulation is currently not possible if the model is exported as a Functional Mockup Unit (FMU) adhering to FMI 3.0 for Model Exchange, since the standard only supports ODEs.

Recent work has also shown growing interest in using Model Exchange FMUs directly within optimization

workflows, in particular with FMUs exported from Modelica models (e.g., (Pfeiffer et al. 2025)). Optimization problems involving FMUs, including trajectory optimization problems and steady-state optimization problems, are typically solved iteratively with gradient-based methods. In such formulations, it is often preferable to handle the algebraic equations at the optimization level rather than inside the FMU.

In this article, we show how to expose algebraic equations during continuous time simulation of an FMU in a standardized way. This standardizes existing tool-internal DAE-Mode so it can be used across the FMI ecosystem, allowing an importing tool with support for DAEs to use them more efficiently. This work uses the concept of *layered standards*, which was developed by the Modelica Association to enable extensions for new use cases while maintaining compatibility with the core standard (Bertsch et al. 2023). This article covers the 1.0.0-alpha.1¹ release, planned for 2026.

We present prototypes demonstrating how exposing non-linear algebraic equations through a Layered Standard for FMI Model Exchange can substantially improve the precision and robustness of optimization and simulation tasks involving such FMUs. A prototype with Dymola and CasADi generates FMUs that include this additional information. On the import side, the usability of the FMI-based Layered Standard for DAEs is demonstrated with prototypes in CasADi, Simcenter Twin Activate, fmusim, and MOO, the dynamic optimization tool used by OpenModelica.

2 A Layered Standard for DAEs

The proposed layered standard for differential algebraic equations, *fmi-ls-dae*, extends FMI 3.0 to support the exchange of dynamic models in DAE form (MAP FMI 2026).

The concept of layered standards on top of the FMI standard was introduced to keep the core FMI standard lean and small, while extending it for new use cases. The FMI standard has certain extension mechanisms for layered standards, such as annotations, a defined subfolder in the FMU for additional files, and a manifest XML file. But even with the additional optional DAE support, the FMU shall still be a valid Model Exchange FMU that can be simulated in all tools with FMI capability without using the additional features of the new layered standard.

This article focuses on the specific use case of exposing systems of non-linear algebraic equations during continuous-time integration, such that a single FMU represents an index-1 semi-explicit DAE. This is achieved by augmenting the active equations during *Continuous-Time Mode* to include the residual equations of the algebraic constraints.

We will briefly discuss remaining work for the first release and the considerably broader scope of fully support-

ing higher-index hybrid DAEs in section 5.

2.1 Previous work on DAE support in FMI

Support for Differential Algebraic Equations in FMI has been discussed in the FMI Project for a long time. In 2015–2018 a working group was formed that collected requirements, discussed different approaches, and created a first internal draft. At that time, the concept of Layered Standards for FMI did not yet exist, so an optional extension of the FMI for Model Exchange was discussed for a version v2.1 or later v3.0 of the FMI standard. The main motivation at that time was the modular exchange and coupling of simulation models that are better described as DAEs (e.g., multibody systems). However, for the final FMI 3.0 feature set it was decided not to include DAE support, as there were unresolved questions, with respect to initialization at the start of the simulation, and at events. The current proposal is based on this previous work and extends it, but is motivated mainly by new use cases such as optimizing FMUs.

2.2 Mathematical Formulation

We consider a semi-explicit index-1 DAE in state-space form:

$$\begin{aligned}\dot{x} &= f(x, w, u, t) \\ 0 &= g(x, w, u, t) \\ y &= h(x, w, u, t),\end{aligned}\tag{1}$$

where x are the continuous-time states, w the algebraic variables, u the inputs, t the independent variable (time), and y the outputs. The functions f , g , and h define the state derivatives, residuals, and outputs, respectively. We assume that $\frac{\partial g}{\partial w}$ is non-singular, so that w is defined implicitly by g . In this article, g represents the system of non-linear equations to be exposed, with w as the iteration variables of those systems. We emphasize that the index-1 property ensures that $\frac{\partial g}{\partial w}$ is structurally non-singular (i.e., the system is well-posed in principle), but does not guarantee numerical non-singularity. Near-singular Jacobians can still occur in practice due to physical operating points or parameter regimes.

2.3 Standard Model Exchange with Hybrid ODEs

The Model Exchange interface is governed by a state machine in which different sets of equations become active depending on the current *mode*, see Figure 1 giving a simplified view.

Three of the modes presented in Figure 1 contain governing equations that are of particular interest when expanding to DAEs. We give a simple overview of these modes:

The *Initialization Mode* establishes consistent initial values for all variables before the simulation begins.

The *Event Mode* is active whenever a discrete change occurs, such as a state or time event. Here the discrete-time equations are evaluated and the discrete states are

¹GitHub: modelica/fmi-ls-dae milestone Alpha-1

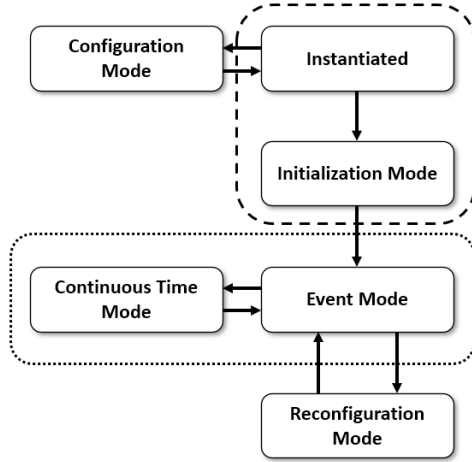


Figure 1. Simplified State Machine for Model Exchange with FMI 3.0. The dashed and dotted lines indicate *Super States* of the FMU. The dashed line around *Instantiated* and *Initialization Mode* is *FMU State Settable* while the dotted line around *Continuous-Time Mode* and *Event Mode* is *Initialized*. The modes *Configuration-* and *Reconfiguration Mode* allow the importer to set `structuralParameters` and `tunable structuralParameters` (Modelica Association 2024, see Section 3.1, State Machine for Model Exchange).

updated. The FMU may also reinitialize continuous states after an event.

And finally, the *Continuous-Time Mode* is the integration phase between events. A simplified view is that the importer sets the current time t , the continuous inputs u , and the continuous states x , using the returned state derivatives $\dot{x}$ and outputs y evaluated by the FMU.

The current FMI 3.0 Model Exchange standard supports hybrid ODEs only. For the DAE formulation of Equation 1, this means that the algebraic variables w and the corresponding equations $g(x, w, u, t) = 0$ are treated as internal to the FMU. By the implicit function theorem, provided $\partial g / \partial w$ is non-singular (as is the case for an index-1 system), w can be locally eliminated and written as $w = g^{-1}(x, u, t)$, usually evaluated iteratively using some variant of Newton’s method when g consists of systems of non-linear equations. Consequently, the equations governing Continuous-Time Mode that are exposed through the FMU C-API do *not* represent $f(x, w, u, t)$ and $h(x, w, u, t)$ from Equation 1, but the composed functions $f(x, g^{-1}(x, u, t), u, t)$ and $h(x, g^{-1}(x, u, t), u, t)$, with the algebraic structure hidden from the importer. In regular Model Exchange, the importer thus sees the following equations during Continuous-Time Mode:

$$\begin{aligned} \dot{x} &= f(x, g^{-1}(x, u, t), u, t) \\ y &= h(x, g^{-1}(x, u, t), u, t). \end{aligned} \quad (2)$$

The variables representing $\dot{x}$ and y in the FMU can be acquired by calling `fmi3Get{VariableType}` which will evaluate the functions f, g^{-1} , and h in (2). The eval-

uation of g^{-1} in (2) typically requires a Newton iteration inside the FMU.

Internal Newton iterations can be problematic for multiple reasons, as discussed below.

Lower accuracy

The internal Newton iteration lowers accuracy. While model equations without algebraic loops can often be evaluated to machine precision, an iterative solver is limited to the stopping criteria of the method, often lower order. This inaccuracy can lead to convergence issues for example in an outer optimization problem, which is typically also solved iteratively using a Newton-type method with similar tolerances.

Higher computational cost

Eliminating algebraic variables leads to nested non-linear solves and prevents exploiting the coupled Jacobian structure, often resulting in higher computational cost compared to a single Newton solve in a DAE formulation.

Hidden state inside the solver

Solving algebraic variables internally introduces hidden solver state (e.g., initial guesses, iteration history), making function evaluations path-dependent and potentially non-reproducible.

Exposing algebraic variables in a DAE formulation avoids this issue and improves robustness, in both optimization and simulation contexts.

Failure to Converge

In a direct DAE solver, if the algebraic constraint is hard to meet, the solver naturally reduces the time step (Δt) to help the global Newton iteration converge. In the nested approach: The inner w -solver has no knowledge of the ODE’s time step. If the inner solver fails to find w (because the initial guess from the last step was too far off), the whole simulation may fail. It is much harder to implement robust “error recovery” when the failure happens inside a black-box function evaluation.

Singularity of g^{-1} and Near-Singular Jacobians

In the current *fmi-ls-dae* standard, only semi-explicit index-1 DAEs are considered. The index-1 assumption requires $\frac{\partial g}{\partial w}$ to be locally non-singular, allowing the algebraic variables to be defined implicitly through the algebraic constraints. In practice, however, $\frac{\partial g}{\partial w}$ may become poorly conditioned near physically relevant operating points, such as bifurcations, parameter boundaries, or stiff regimes.

The standard itself does not prescribe a specific treatment for such situations. Rather, by exposing the coupled DAE system

$$\mathbf{J}_{\text{DAE}} = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial w} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial w} \end{pmatrix},$$

to the simulation environment, it enables dedicated DAE solvers to apply robustness measures such as adaptive step-size control, scaling, regularization, and consistent initialization. In contrast, the nested ODE formulation relies on the local elimination of the algebraic variables, for which the sensitivity

$$\frac{\partial w}{\partial x} = - \left(\frac{\partial g}{\partial w} \right)^{-1} \frac{\partial g}{\partial x}$$

is directly affected by the conditioning of $\frac{\partial g}{\partial w}$. Consequently, numerical errors may be significantly amplified when $\frac{\partial g}{\partial w}$ is nearly singular.

Although the DAE formulation does not eliminate singularity-related difficulties, it avoids the explicit construction of the reduced dynamics and gives the solver direct access to the coupled system, generally improving the detection and management of ill-conditioned situations (K. E. Brenan 1989; Ernst Hairer 1991). Cases in which $\frac{\partial g}{\partial w}$ becomes singular correspond to a local breakdown of the index-1 assumption and are outside the scope of the current version of the standard, remaining a topic for future extensions.

2.4 Extensions via *fmi-ls-dae*

The core idea of *fmi-ls-dae* is to augment the active equations during Continuous-Time Mode (Equation 2) with *residual* variables r and *algebraic* variables w :

$$\begin{aligned} \dot{x} &= f(x, w, u, t) \\ r &= g(x, w, u, t) \\ y &= h(x, w, u, t), \end{aligned} \quad (3)$$

to properly represent the mathematical formulation of Equation 1. It is the responsibility of the importer to find values for w such that the residuals r are zero.

The layered standard adds an FMI-LS-DAE XML manifest file which tags the DAE related variables from the `ModelVariables` element in the `modelDescription.xml`, and adds a new model structure which contains residual equations (the constraints), available in the `extra/` directory (Modelica Association 2024, see Section 2.6. Versioning and Layered Standards).

In the current version, there is no extension of the FMI C API.

2.4.1 DAE-Mode

A layered standard requires that any importer not supporting this extension can simply ignore that semantic and simulate the Model Exchange FMU as a hybrid ODE. This means that an importer should “opt-in” to simulate the FMU in DAE-Mode with the residuals exposed to be solved by the importer. To handle this we propose that an FMU supporting *fmi-ls-dae* must expose a structural parameter called *enableDAEModeParameter* of type Boolean with variability *fixed* or *tunable* and start set to *false*, as shown in Listing 1.

Listing 1. Structural parameter *enableDAEModeParameter* exposed by an FMU to opt in to DAE-Mode as defined by *fmi-ls-dae*.

```
<Boolean name="<free to choose>"
valueReference="<free to choose>"
causality="structuralParameter"
variability="tunable|fixed"
start="false"
description="Set to true to enable DAE mode
as defined by FMI-LS-DAE."/>
```

This parameter can be set in *(Re)Configuration Mode* (see Figure 1 for the simplified view of the state machines of the FMU); setting it to *true* activates the equations in (3).

2.4.2 Initialization

Initialization in DAE mode is not yet fully specified, since the first alpha release focuses on exposing the DAE formulation during Continuous-Time Mode. Nevertheless, this section details the current ideas regarding initialization.

Computing a *consistent* initial state $(x(t_0), w(t_0))$ using

$$0 = g(x(t_0), w(t_0), u(t_0), t_0) \quad (4)$$

and initial conditions is a non-issue for the importer of the FMU in ODE mode, since the FMU will handle it internally. When *enableDAEModeParameter* is *false* during initialization the FMU will be initialized in ODE-Mode. After the Initialization Mode is exited the FMU is in Event Mode and allowed to enter Reconfiguration Mode to switch into DAE-Mode.

Another option is to switch into DAE-Mode during Configuration Mode and delegate the computation of a consistent initial condition to the importing tool, which may apply suitable numerical or symbolic methods, or rely on expert-supplied initial guesses. In the most direct case, the user provides a fully consistent set of initial values $(x(t_0), w(t_0))$. When start values are provided only for the differential states $x(t_0)$, initialization reduces to solving the algebraic constraint (4) for $w(t_0)$ alone via Newton’s method.

2.4.3 Algebraic variables

The algebraic variables w are specified by the `<AlgebraicVariables>` element in the manifest file, referencing existing entries in the model description XML element `<ModelVariables>` by their `valueReference`.

For the C-API this means that the FMU must allow the importer to call `fmi3Set{VariableType}`, which is normally not permitted for local variables. Moreover, they must have the possibility to be exposed as an `<InitialUnknown>` in the new `<ModelStructure>` (see section subsection 2.4.4), since the initial equations defined in Initialization Mode now also may initialize the algebraic variables.

2.4.4 Model Structure

A DAE-version of element `<ModelStructure>`, extended with `<Residual>` elements is added to the FMI-LS-DAE XML manifest file. Each `<Residual>` encodes the constraint r from (3), and carries a `valueReference` identifying the residual variable, an optional `dependencies` list of known value references, and an optional `dependenciesKind` list.

For the purpose of exposing the semi-explicit index-1 formulation in (1) it should be assumed that the DAE formulation represented by the new `<ModelStructure>` defines an index-1 system. Including higher-index formulations are planned and discussed in section 5.3.

The `dependencies` attribute of the `<Residual>`, `<Output>`, and `<ContinuousStateDerivative>` may include algebraic variables.

2.4.5 Example

To illustrate the extensions of the XML schema, we will be studying a simple academic DAE. It is described by 6 scalar equations:

$$\begin{aligned} \dot{x}_1 &= \sin(x_1) + \sin(w_1 w_2 u_1) \\ \dot{x}_2 &= \sin(x_1 x_2) + \sin(w_1 w_2 u_2) + u_1^2 \\ 0 &= w_1 u_1^2 + \tanh(3w_1) + u_2 x_1^3 \\ 0 &= \frac{\exp(w_1 w_2 u_1)}{3} - \sin(w_2 x_2) \\ y_1 &= u_1 x_1 w_1 + \sin(u_2) \\ y_2 &= u_2 x_2 w_2 + \sin(u_1 u_2) \end{aligned} \quad (5)$$

Here x_1, x_2 are state variables with state derivatives $\dot{x}_1, \dot{x}_2$, w_1, w_2 are algebraic variables, u_1, u_2 inputs and y_1, y_2 outputs. Initial conditions are provided by $x_1(0) = 0.5$ and $x_2(0) = 0.5$. In this case f , g , and h are represented by the vectors

$$\begin{aligned} f &= \begin{bmatrix} \sin(x_1) + \sin(w_1 w_2 u_1) \\ \sin(x_1 x_2) + \sin(w_1 w_2 u_2) + u_1^2 \end{bmatrix}, \\ g &= \begin{bmatrix} w_1 u_1^2 + \tanh(3w_1) + u_2 x_1^3 \\ \frac{\exp(w_1 w_2 u_1)}{3} - \sin(w_2 x_2) \end{bmatrix}, \\ h &= \begin{bmatrix} u_1 x_1 w_1 + \sin(u_2) \\ u_2 x_2 w_2 + \sin(u_1 u_2) \end{bmatrix}. \end{aligned} \quad (6)$$

Consider an FMU of Equation 5, with a `valueReference` for each variable according to Table 1. The proposed element listing the algebraic variables, as well as extended `<ModelStructure>` for Continuous-Time Mode is presented in Listing 2 and Listing 3 respectively:

Listing 2. The two algebraic variables from Equation 5 are exposed through a new element in the `fmi-ls-manifest.xml`.

```
<AlgebraicVariables>
  <AlgebraicVariable valueReference="4"/>
  <AlgebraicVariable valueReference="5"/>
</AlgebraicVariables>
```

Table 1. Value references for the variables in (5). The variables r_1 and r_2 represent the residuals for the non-linear equations defined by g .

Variable	valueReference
x_1	0
x_2	1
$\dot{x}_1$	2
$\dot{x}_2$	3
w_1	4
w_2	5
u_1	6
u_2	7
y_1	8
y_2	9
r_1	10
r_2	11

Listing 3. Example for the extended `ModelStructure` (for Continuous-Time Mode) for Equation 5

```
<ModelStructure>
  <Output
    valueReference="8"
    dependencies="0 4 6 7"/>
  <Output
    valueReference="9"
    dependencies="1 5 6 7"/>
  <ContinuousStateDerivative
    valueReference="2"
    dependencies="0 4 5 6"/>
  <ContinuousStateDerivative
    valueReference="3"
    dependencies="0 1 4 5 6 7"/>
  <Residual
    valueReference="10"
    dependencies="0 4 6 7"/>
  <Residual
    valueReference="11"
    dependencies="1 4 5 6"/>
</ModelStructure>
```

All existing elements of the `<ModelStructure>` in Listing 3 above retain their standard semantics; the only change is that their `dependencies` attributes may now also reference algebraic variables, and that the algebraic variables can be listed among the `<InitialUnknown>`.

It should be emphasised that the `dependencies` and `dependenciesKind` attributes will differ between ODE- and DAE Mode in more ways than just including `valueReferences` of the algebraic variables. For example with the SimpleDAE in Equation 5, if it were represented as an ODE, then the derivative $\dot{x}_1$ will implicitly depend on the input u_2 through the algebraic variables appearing in the residual equations. However, in the DAE formulation these algebraic variables appear directly in $\dot{x}_1$, and the input dependencies are instead encoded in the residual equations.

2.4.6 Partial derivatives

The standard directional and adjoint derivative APIs (`fmi3GetDirectionalDerivative`, `fmi3GetAdjointDerivative`) are used as defined in the FMI 3.0 core standard, extended so that algebraic variables may appear as knowns for any unknown, and residual variables may appear as unknowns.

3 Implementations

This section details prototype implementations developed alongside the proposals described in this article. It should be noted that *fmi-ls-dae* has not yet released any formal specification for the semi-explicit index-1 formulation; what is presented here reflects the authors' current proposals along with the *fmi-ls-dae* working group's discussions, and is subject to change as the standard evolves.

3.1 Dymola

Dymola (Dynamic Modeling Laboratory) is a commercial tool for modeling and simulation based on Modelica and FMI. A Modelica model is in general described by a high-index implicit DAE:

$$\begin{aligned} 0 &= F(\dot{x}, x, w, u, t, d) \\ d &= \alpha(\dot{x}, x, w, u, t, \text{pre}(d)) \\ y &= H(\dot{x}, x, w, u, t, d). \end{aligned} \quad (7)$$

Where x are the dynamic variables with derivatives $\dot{x}$, w are algebraic variables, and t is time. Here d are the discrete variables, which we ignore for now. The variables u and y represent any top-level inputs and outputs. During code generation, a Modelica tool applies symbolic transformations to reduce Equation 7 to a form suitable for simulation. Algebraic variables that do not affect the dynamics (auxiliary variables) are eliminated in this process. Loops that can be solved symbolically (such as linear systems or inversions of elementary functions like $\exp$ and $\sin$) are solved at code generation time. What remains are the irreducible non-linear systems of equations that affect the differentiated variables, where the non-linear equations depend on the variables that appear differentiated ($\dot{x}$) and iteration variables (w), which together with the dynamic equations yield the index-1 semi-explicit DAE of (1). It is precisely this form of semi-explicit DAE with g consisting of non-linear equations that are exposed with the generated DAE-FMU.

3.1.1 Generating the FMU

As of the release of Dymola 2026x Refresh 1 on April 17, 2026, to support the layered standard for Differential Algebraic Equations, Dymola gives the option to expose these non-linear equations and iteration variables in the FMU, to be handled by the importer. The non-linear equations (residuals) are exposed as variables in the FMU with the algebraic variables tagged in the `fmi-ls-manifest.xml`. The partial derivative functions `fmi3GetDirectionalDerivative` and

`fmi3GetAdjointDerivative` have been updated to take in residuals as unknowns and algebraic variables as knowns.

Note that after exiting Initialization Mode, the FMU will initialize the iteration variables to solve the residuals, then during Continuous-Time Mode, it is the importer's responsibility to solve the residual equations. Therefore, the DAE FMUs generated from Dymola 2026x Refresh 1 are automatically not valid ODE FMUs since the *enableDAEModeParameter* hasn't been implemented at the time of that release.

One current limitation is that models with events exported as FMUs do not work in general, since there is not yet a formal proposal for extending the Model Exchange interface to handle discrete behavior in hybrid DAEs. Consequently, dynamic state selection is also not supported for DAE-FMUs yet, as the selection of states is generally discontinuous.

At the time of writing, all the above limitations regarding *enableDAEModeParameter*, events, and dynamic state selection are being implemented in a future Dymola release.

3.2 CasADi

CasADi (Joel A E Andersson et al. 2019) is an open-source framework for C++, Python, MATLAB and Octave. In particular, CasADi provides building blocks for efficiently formulating and solving non-linear optimization problems, specifically problems involving dynamic systems. CasADi has been used to implement various optimization algorithms in industry and academia and is used in the backend of multiple open-source and commercial tools for dynamic optimization.

Since 2023, CasADi allows standard Model Exchange FMUs to be imported and used as twice differentiable functions within the framework (Joel A. E. Andersson 2023). At the same time, FMI 3.0 export was added to CasADi, allowing users to export models formulated into CasADi as standard FMUs. CasADi generated FMUs include serialized CasADi expressions via layered standard, providing a standard mechanism to recover the original CasADi expressions from the FMU when imported back into CasADi.

As part of this work, CasADi's FMU import and export functionality was extended to DAEs, via the *fmi-ls-dae* layered standard.

3.3 FMIOPT

FMIOPT is a commercial, general-purpose dynamic optimization framework built on top of CasADi. It aims at providing a robust and efficient workflow for solving dynamic optimization problems, starting with challenging simulation models. Its method-agnostic and FMI-based middle-layer allows the numerical solution to be performed in ephemeral solver instances in high-performance parallelized hardware, including in a cloud environment. Features include multiple sequential and simultaneous so-

lution methods, minimal-time formulations, multi-stage formulations, visualizations and advanced diagnostics.

3.4 Simcenter Twin Activate

Simcenter Twin Activate is a general-purpose modeling and simulation platform designed for hybrid dynamical systems. It supports ordinary differential equations (ODEs) and fully-implicit index-1 and semi-explicit index-2 differential-algebraic equations (DAEs), along with ODEs combined with constraints (invariants). The software includes multiple numerical solvers for both ODE and DAE systems.

Historically, many models resulted in high-index DAEs that could not be exported as Model Exchange FMUs. The only viable option was to export them with FMI for Co-Simulation, which performed poorly when connected to other FMUs. The introduction of DAE-based dynamical systems in the Model Exchange FMU standard has opened new possibilities, allowing high-index models to be exported as FMUs compatible with other vendors' tools.

To validate this capability, Simcenter Twin Activate has enhanced its FMU import functionality to support the new DAE features and imported and tested two DAE-FMUs in subsection 4.1.

3.5 MOO

MOO (Modelica / Model Optimizer) is a library for dynamic optimization and forms the basis for the new optimization runtime of OpenModelica. The presented FMI interface prototype extends MOO to support dynamic optimization of FMUs on a fixed time horizon $[t_0, t_f]$. The supported problem class is given by

$$\begin{aligned} \min_{u,p} \quad & M(x_0, w_0, x_f, w_f) + \int_{t_0}^{t_f} L(x, w, u, t) dt \\ \text{s.t.} \quad & \dot{x} = f(x, w, u, t) \\ & 0 = g(x, w, u, t) \\ & c^L \leq c(x, w, u, t) \leq c^U \\ & \psi^L \leq \psi(x_0, w_0, x_f, w_f) \leq \psi^U \end{aligned} \quad (8)$$

The objective consists of a Mayer term M and a Lagrange term L . Dynamics are described by the differential equations $\dot{x} = f(\cdot)$ with states x , controls u , and explicit algebraic variables w with associated algebraic constraints $g(\cdot) = 0$. Path and boundary constraints are defined by the functions c and ψ , respectively. MOO additionally supports static parameter optimization and free initial and final times. These features could be integrated into the FMI workflow by exposing FMU parameters and time variables as optimization variables, but are not yet part of the current prototype.

In the current prototype, differential and algebraic equations are identified from the FMU, while selected FMU value references can be used to define the Mayer and Lagrange terms, as well as path and boundary constraints with their corresponding bounds. The interface provides

the required function evaluations and Jacobians to the optimizer. Second-order derivatives are currently not available, but could be provided using finite differences. The prototype is available on GitHub².

3.6 fmsim

fmsim³ is a command line tool to import, validate, and simulate Functional Mock-up Units, written in Rust. fmsim has been expanded with the option to import FMUs conforming to *fmi-ls-dae* and supplies the Sundials IDA solver (Gardner et al. 2022; Hindmarsh et al. 2005) for its simulation of such Model Exchange FMUs.

4 Examples

Four DAE-FMUs were exported using Dymola 2026x Refresh 1:

- `SimpleDAE.fmu`: Based on Equation 5.
- `Fourbar1.fmu`: Based on Modelica model `Modelica.Mechanics.MultiBody.Examples.Loops.Fourbar1` from the Modelica Standard Library version 4.1.0. A simple kinematic loop consisting of 6 revolute joints, 1 prismatic joint and 4 bars that is often used as basic constructing unit in mechanisms - was exported as `Fourbar1.fmu`.
- `Crane.fmu`: Describes a crane with mass m_C , actuated by a force u and carrying a load with mass m_L suspended by a rope of length l :

$$\begin{aligned} m_L \ddot{x}_L &= \frac{x_L - x_C}{l} T, & m_L \ddot{y}_L &= m_L g - \frac{y_L}{l} T, \\ m_C \ddot{x}_C &= \frac{x_L - x_C}{l} T - u, & (x_L - x_C)^2 + y_L^2 &= l^2, \end{aligned} \quad (9)$$

where T denotes rope tension and x_L , y_L , and x_C are positions of the load and crane.

- `MultilinkControl_DAE.fmu`: Multilink rear suspension corner model. See Figure 6 for a schematic and description. A corresponding ODE-FMU with the regular Model Exchange interface was also generated for comparison on the optimization task in subsection 4.2.

Moreover, a new FMU has been added to the list of Reference FMUs⁴ that conforms to the current version of *fmi-ls-dae*. This is the `Roberts.fmu` which implements a problem from chemical kinetics, due to Robertson (Robertson 1966). It implements the following equations

²GitHub repository: AMIT-HSBI/MOO branch *fmi-ls-dae*

³GitHub repository: <https://github.com/modelica/fmsim>

⁴GitHub repository: <https://github.com/modelica/Reference-FMUs>

in (10).

$$\begin{aligned} \frac{dy_1}{dt} &= -0.04y_1 + 10^4 y_2 y_3, \\ \frac{dy_2}{dt} &= 0.04y_1 - 10^4 y_2 y_3 - 3 \times 10^7 y_2^2, \\ 0 &= y_1 + y_2 + y_3 - 1, \end{aligned} \quad (10)$$

with initial conditions

$$y_1(0) = 1, \quad y_2(0) = y_3(0) = 0. \quad (11)$$

4.1 Simcenter Twin Activate - Simulation

The two DAE-based FMUs `SimpleDAE.fmu` and `Fourbar1.fmu` were imported and tested. As shown in Figure 2, the internal DAE equations from the FMU were independently implemented in a built-in block to enable direct comparison and validation of the results.

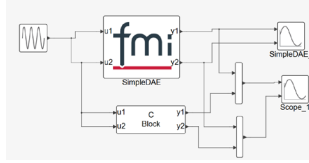


Figure 2. The simpleDAE FMU imported and a built-in block in Simcenter Twin Activate.

Figure 3 presents a comparison of the simulation results from both the built-in block and the FMU, demonstrating a perfect match between the two implementations.

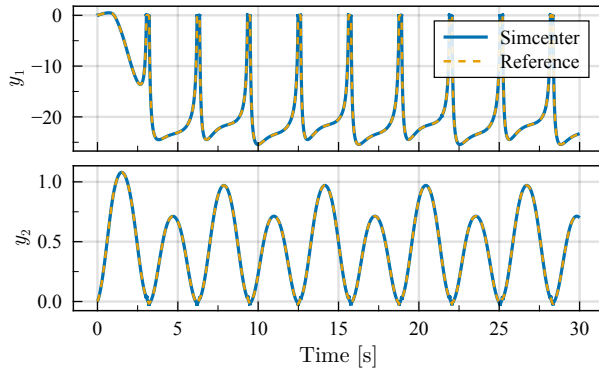


Figure 3. Simcenter Twin Activate simulation results for `SimpleDAE.fmu` with input $u_1(t) = u_2(t) = \sin(t)$ compared to reference solution performed by OpenModelica: outputs y_1 (top) and y_2 (bottom).

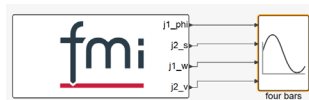


Figure 4. The fourbar1 FMU imported in Simcenter Twin Activate.

Figure 5 displays the simulation results of the FMU integrated into the model shown in Figure 4, which are identical to those produced by both Dymola and OpenModelica with the original Modelica model.

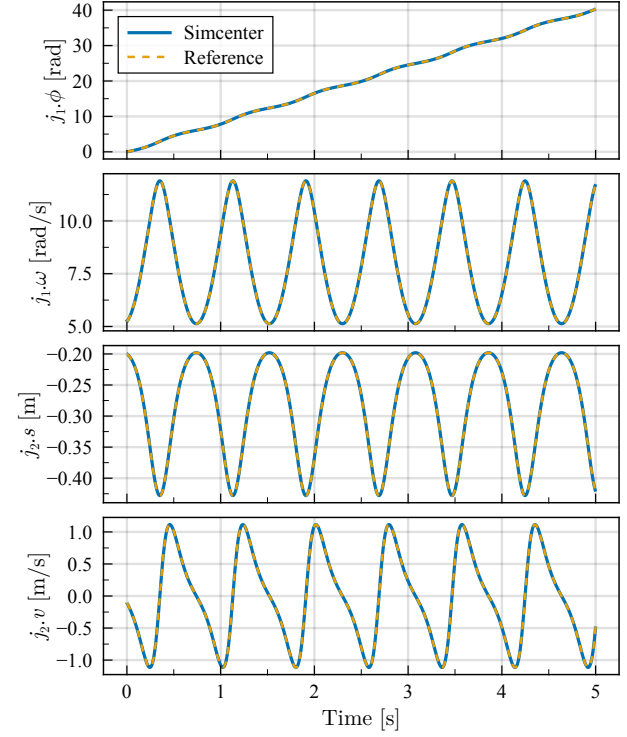


Figure 5. Simcenter Twin Activate simulation results for `Fourbar1.fmu` compared to the Modelica reference solution: revolute joint angle ϕ_1 , angular velocity $\dot{\phi}_1$, prismatic joint displacement s_2 , and prismatic joint velocity $\dot{s}_2$.

4.2 FMIOPT - Dynamic Optimization

To exercise the layered standard on an industrially relevant model, we consider a multibody model of a rear multilink suspension corner. This model was derived from the Claytex Suspensions library and exported as the FMU `MultilinkControl_DAE.fmu` mentioned in the beginning of this section. A schematic of the model is shown in Figure 6.

After the kinematic constraints of the linkage are accounted for, the suspension corner has a single mechanical degree of freedom, yielding two differential states ϕ and $\omega = \dot{\phi}$. A vertical force f_z is applied at the wheel hub and the lower control arm joint angle ϕ is tracked against a sinusoidal reference. This setup is similar to a vertical wheel bump experiment used to characterize suspension kinematics and the wheel rate. However, rather than imposing the motion quasi-statically, the optimal control problem inverts the suspension dynamics to find the actuator force that produces a prescribed angular trajectory.

The model contains a kinematic loop that cannot be reduced to an ODE analytically, which results in a nonlinear system of equations that must be solved at each time

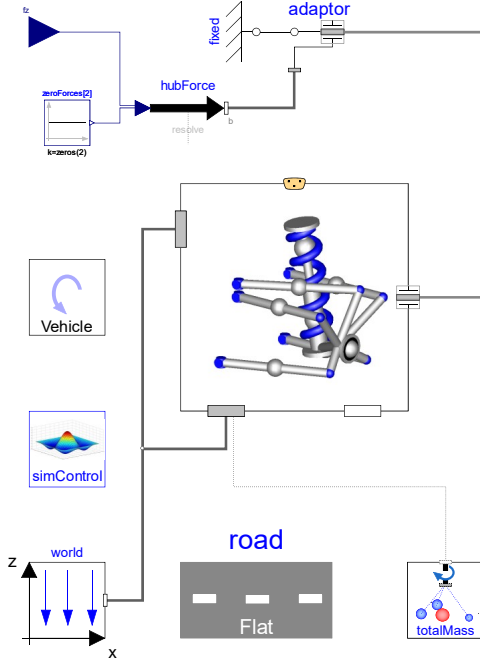


Figure 6. Multilink rear suspension corner model. The actuator force f_z is the control input and the joint angle ϕ is the tracked output.

step during simulation. This makes it an ideal candidate for testing the layered standard for DAEs. Moreover, in SI units the model is poorly scaled, with forces at the wheel hub on the order of 10^3 – 10^4 N and angles on the order of 10^{-1} rad, which can lead to numerical issues especially if the algebraics are not exposed to the optimizer, leading to a Newton-over-Newton solve that can produce noisy derivatives and failures in the inner Newton iteration.

After symbolic processing in Dymola, the model results in a semi-explicit index-1 DAE of the form (1) with two differential states (ϕ and ω), a vector $\mathbf{z}$ of algebraic iteration variables originating from the kinematic loops of the suspension, and the scalar control input f_z . The model is then exported as LS-DAE FMU using the *fmi-ls-dae* export described above, and as a standard ODE FMU 3.0 for comparison. The optimal control problem is set up as follows:

$$\begin{aligned}
 \min_{\mathbf{x}(\cdot), \mathbf{z}(\cdot), f_z(\cdot)} \quad & J = \int_0^T (w_\phi (\phi - \phi_{\text{ref}})^2 + w_u f_z^2) dt \\
 \text{s.t.} \quad & \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{z}, f_z, t), \\
 & \mathbf{0} = \mathbf{g}(\mathbf{x}, \mathbf{z}, f_z, t), \\
 & \mathbf{x} = [\phi, \omega]^\top, \\
 & \phi(0) = \phi_0, \quad \omega(0) = \omega_0, \\
 & f_{z,\min} \leq f_z(t) \leq f_{z,\max},
 \end{aligned} \tag{12}$$

on the fixed horizon $t \in [0, T]$ with $T = 2$ s. The reference trajectory and consistent initial conditions are

$$\begin{aligned}
 \phi_{\text{ref}}(t) &= \phi_0 + A_\phi \sin(2\pi f t), \\
 \omega_0 &= 2\pi f A_\phi,
 \end{aligned} \tag{13}$$

with amplitude $\phi_0 = -0.28$ rad, $A_\phi = 0.13$ rad, and frequency $f = 1$ Hz. The actuator bounds $f_{z,\min} = -f_{z,\max} = -2 \cdot 10^6$ N are intentionally loose and act as non-binding numerical safeguards rather than as physical limits. The cost weights $w_\phi = 10^3$ and $w_u = 10^{-12}$ follow a Bryson-style normalization: with ϕ on the order of 10^{-1} rad and the optimal f_z on the order of 10^3 – 10^4 N, the two terms in the integrand are brought to comparable magnitude, with tracking dominating slightly so that f_z is regularized only mildly.

The OCP is transcribed by direct collocation on $N = 150$ uniform intervals using a 3-point Radau IIA scheme. On each collocation interval, the FMU is queried for $\mathbf{f}$, $\mathbf{g}$, and the corresponding first-order partial derivatives, and the algebraic variables $\mathbf{z}$ are introduced as additional decision variables. The residual values $r = \mathbf{g}(\mathbf{x}, \mathbf{z}, f_z, t)$ exposed by the FMU through the layered standard are imposed as equality constraints $r = 0$ at every collocation node. The resulting NLP is solved by IPOPT (Wächter and Biegler 2006) with the MA57 linear solver and a limited-memory BFGS Hessian approximation.

Figure 7 shows the optimal trajectories of the tracked angle, the corresponding angular velocity, and the optimal actuator force f_z obtained from the DAE-FMU. The reference is followed essentially exactly over the full horizon, while f_z remains well within its prescribed bounds.

An attempt was also made to solve the same OCP using a regular (ODE) FMUs, where the kinematic loop is closed by the FMU-internal Newton iteration. Using the same transcription and solver settings, the ODE formulation run failed to converge, with IPOPT exiting with a *Restoration Failed!* status after 369 iterations. The corresponding trajectories in Figure 7 are also clearly unphysical: the angular velocity ω exhibits spurious artifacts around $t = 1$ s and the optimal actuator force collapses to zero rather than tracking the reference. We caution against drawing broad conclusions about the failure of the ODE FMU formulation – more investigation is needed to determine the root cause of the failure, which may be unrelated to the ODE vs. DAE formulation.

4.3 MOO - Dynamic Optimization

The DAE-FMU version of the multilink suspension corner OCP (12) was additionally solved using MOO with the FMI interface described in subsection 3.5. The OCP is transcribed using $N = 400$ discretization points using direct collocation. It is solved with IPOPT (Wächter and Biegler 2006) and MUMPS (Amestoy et al. 2001) chosen as linear solver, and converges to the optimum in 57 NLP iterations. Figure 8 shows the optimal joint angle ϕ against the reference ϕ_{ref} as well as the actuator force f_z .

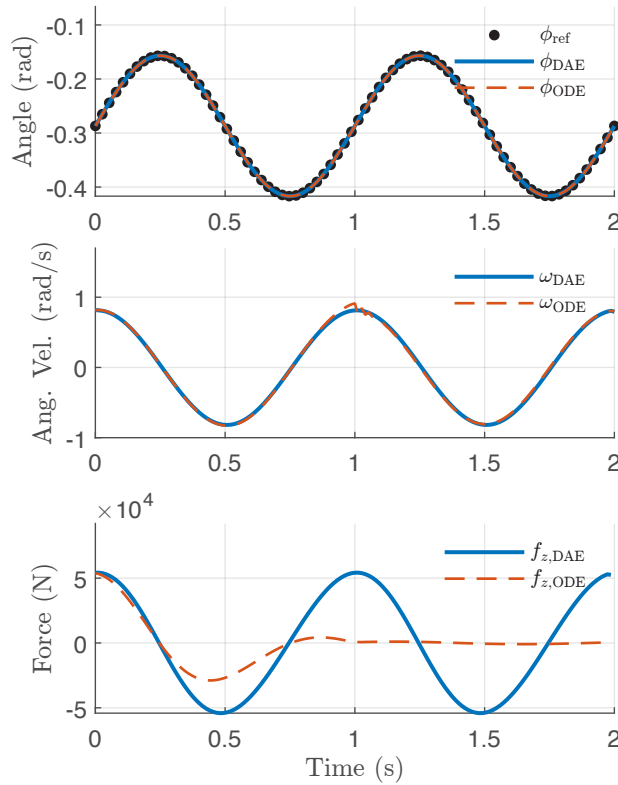


Figure 7. Optimal trajectories for the multilink suspension corner: tracked joint angle ϕ versus reference ϕ_{ref} , angular velocity ω , and optimal actuator force f_z .

4.4 CasADi - Optimal control, FMU export

The example `fmu_collocation.py` was added to CasADi's example collection, demonstrating how to find the optimal control that takes the load of the crane in Equation 9 to a desired position while minimizing a quadratic cost. Using the generated `Crane.fmu`, it implements a direct collocation method, solved with IPOPT (Wächter and Biegler 2006) with first and second-order derivatives provided by CasADi. The same example also demonstrates how an equivalent FMU can be exported from CasADi.

4.5 fmsim - Simulating the Roberts FMU

The `Roberts.fmu` can be imported and simulated as an *fmi-ls-dae* FMU (using the default experiment in the FMU) in `fmsim` for example with the following command line arguments:

```
fmsim simulate Roberts.fmu --enable-dae
--solver ida --interface-type me
--log-time-scale --show-plot
--show-markers --output-interval 2
```

For a full list of possible options, one can run `fmsim --help` and `fmsim simulate --help`. Running the above commands results in Figure 9 which shows the two states and one algebraic variable over the time horizon defined by the default experiment in the

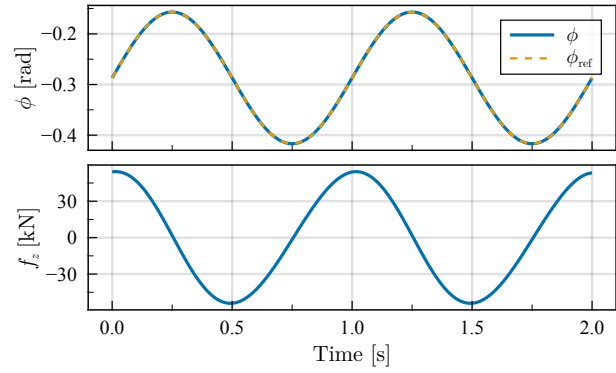


Figure 8. Solution of the multilink suspension corner problem solved using MOO. The joint angle ϕ closely follows the reference ϕ_{ref} and the actuator force f_z agrees with the DAE-FMU solution of FMIOPT.

FMU.

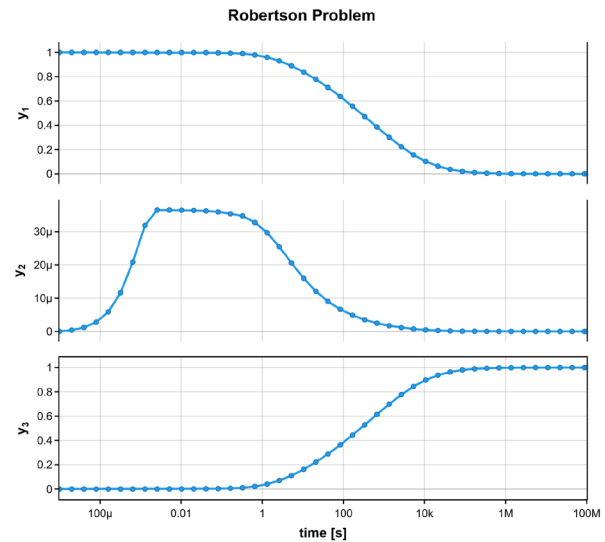


Figure 9. Solution of the `Roberts.fmu` simulated with `fmsim`, plotting the two states y_1 and y_2 , and the algebraic variable y_3 .

5 Outlook

This article focuses on exposing the DAE formulation for continuous time simulation, which will be included for the pre-release version 1.0.0-alpha.1, planned for 2026. The details of handling initialization and event iteration are under active development within the *fmi-ls-dae* working group and are outlined in subsection 5.1 and subsection 5.2. Once these are finalized, the next milestone in development⁵ is the publication of pre-release version 1.0.0-alpha.2, which is also planned for 2026. subsection 5.3 provides an outlook on potential support for higher-index DAE-FMUs in a later major release of the

⁵GitHub: [modelica/fmi-ls-dae](https://github.com/modelica/fmi-ls-dae) milestone Alpha-2

layered standard.

5.1 Consistent Initial Condition

In addition to the discussed initialization methods from subsection 2.4.2 some simulation environments also allow other methods for index-1 DAEs, depending on the available information and the modeling intent.

Partially inconsistent initial conditions apply a correction or projection step to compute a nearby consistent state before starting the integration. In addition, a DAE may be initialized in steady state by solving the system under the assumption that all time derivatives of the differential states are zero, yielding an equilibrium point that satisfies both the differential and algebraic equations.

Although the exact approach has not yet been finalized, the initialization type could be specified by selecting appropriate variables in the `<InitialUnknown>` element.

5.2 Event Handling

Another important aspect for the final release of *fmi-ls-dae* is event handling and the associated reinitialization procedure. Events may result from switching conditions, zero-crossing detections, synchronous events, or explicit state jumps. Such events can invalidate the consistency conditions of the DAE and therefore require reinitialization before integration can continue. After applying any prescribed jumps to the differential states, the algebraic variables must be recomputed such that the algebraic constraints are satisfied. Depending on the model, this may require solving a non-linear system to establish a new consistent operating point. Model invariants and conservation laws should also be enforced during this process. Reinitialization takes place during event mode in cooperation with the importing simulation environment.

The current *fmi-ls-dae* specification does not yet define a standardized mechanism for event-driven reinitialization, and the procedure is therefore delegated to the importer and its underlying DAE solver using the information provided by the FMU. Defining a portable and solver-independent reinitialization workflow remains an important topic for future standardization efforts.

Dymola's (non-FMI) DAE-Mode localizes the event time efficiently and accurately based on the ODE representation of the DAE (Henningsson et al. 2019). OpenModelica's (non-FMI) DAE-Mode requires the causalized ODE representation to handle synchronous events and discrete variables during event iteration (Braun et al. 2017). Other tools might only use their (non-FMI) DAE-Mode for all of the event handling. The authors believe that with the possibility to switch between DAE- and ODE-Mode during Reconfiguration Mode all of these cases can be incorporated by *fmi-ls-dae*. Development of reference DAE-FMUs showcasing event handling is planned for the first release of *fmi-ls-dae*.

5.3 Higher-Index Formulations

This article focuses on the specific use case of exposing algebraic equations in an index-1 semi-explicit formulation, but the end goal of the layered standard is to support higher-index DAEs as well. Part of the ongoing discussion at the time of writing is to include a standardized way to specify *invariants* for *fmi-ls-dae*. Although they are not directly related to higher-index problems, one idea is to use the proposed `<Invariant>` element, such that by letting the element contain a derivative attribute pointing to a `<Residual>` element, it could define the higher-index residuals. An importing tool may use this structural information to apply, for example, index-reduction techniques or drift-correction methods.

6 Conclusion

We proposed *fmi-ls-dae*, a layered standard on top of FMI 3.0 Model Exchange that exposes algebraic equations of a model and their associated algebraic variables as part of a semi-explicit index-1 DAE, together with the extensions to the FMI XML schema required to describe the residuals and algebraic variables in a backwards-compatible way. With this extension we demonstrated the standard end-to-end through prototype implementations in five different tools: Dymola and CasADi were extended to export DAE-FMUs conforming to the layered standard, and CasADi, FMIOPT, Simcenter Twin Activate, fmsim, and MOO were extended to import them for simulation and dynamic optimization. For one industrially relevant multi-link rear suspension corner model, the DAE-FMU formulation was able to converge to the optimal solution, while the equivalent ODE-FMU formulation failed to converge.

Author Contributions

All authors contributed to the specification through collaborative meetings and discussions, and all authors have contributed to the writing of the paper.

- The paper was coordinated by Elmir Nahodovic, Andreas Heuermann, and Joel A. E. Andersson.
- Andreas Heuermann and Joel A. E. Andersson served as working group leaders.
- Elmir Nahodovic led the index-1 semi-explicit DAE specification and developed the Dymola prototype for this proposal, with technical support from Erik Henningsson and Hans Olsson.
- Joel A. E. Andersson designed and implemented the CasADi and FMIOPT prototypes.
- Adwait Verulkar and Srikanth Sivaramakrishnan re-initiated the *fmi-ls-dae* project by proposing it in the FMI Design Meetings and provided the industrial Multilink study with FMIOPT.

- Masoud Najafi extended Simcenter Twin Activate to support the import of such FMUs.
- Linus Langenkamp developed the prototype with MOO, with support from his supervisor Bernhard Bachmann. Andreas Heuermann supported in the prototype for MOO.
- Christian Bertsch, as the FMI project leader, contributed with his knowledge of layered standards for FMI.
- Torsten Sommer contributed by extending his command line tool `fmsim` to import `fmi-ls-dae` FMUs, and with the new `Roberts.fmu` in the Reference FMUs.

Acknowledgements

Part of this work was conducted within the EU-REKA ITEA4 OpenSCALING project (openscaling.org) by the German Federal Ministry of Education and Research (BMBF) under grant number 16IS23062E and the Swedish Innovation Agency (grant number 2023-00969). The authors would like to express their sincere appreciation to the OpenSCALING project for their support, collaboration, and shared commitment in advancing open standards for modeling and simulation technologies.

We want to thank the FMI Project members, especially also the current and former members of the DAE working group!

References

- Amestoy, P.R., I. S. Duff, J. Koster, and J.-Y. L'Excellent (2001). "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1, pp. 15–41.
- Andersson, Joel A E, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl (2019). "CasADi – A software framework for nonlinear optimization and optimal control". In: *Mathematical Programming Computation* 11.1, pp. 1–36. DOI: 10.1007/s12532-018-0139-4.
- Andersson, Joel A. E. (2023-10). "Import and Export of Functional Mock-up Units in CasADi". In: *Proceedings of the 15th International Modelica Conference*. Vol. 2855, pp. 321–326.
- Bertsch, Christian et al. (2023-10). "Beyond FMI – Towards New Applications with Layered Standards". In: *Proceedings of the 15th International Modelica Conference*. Vol. 2855, pp. 381–387.
- Braun, Willi, Francesco Casella, and Bernhard Bachmann (2017). "Solving Large-scale Modelica Models: New Approaches and Experimental Results using OpenModelica". In: *Proceedings of the 12th International Modelica Conference*. Vol. 132. Linköping Electronic Conference Proceedings 63. Prague, Czech Republic: Linköping University Electronic Press, pp. 557–563. DOI: 10.3384/ecp17132557.
- Cellier, François E. and Ernesto Kofman (2006). *Continuous System Simulation*. Berlin, Heidelberg: Springer-Verlag.
- Ernst Hairer, Gerhard Wanner (1991). *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problem*. Springer Series in Computational Mathematics, vol. 14.
- Gardner, David J, Daniel R Reynolds, Carol S Woodward, and Cody J Balos (2022). "Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)*. DOI: 10.1145/3539801.
- Henningsson, Erik, Hans Olsson, and Luigi Vanfretti (2019-03). "DAE Solvers for Large-Scale Hybrid Models". In: *Proceedings of the 13th International Modelica Conference*. Vol. 157. Linköping Electronic Conference Proceedings. Regensburg, Germany: Linköping University Electronic Press, p. 50. DOI: 10.3384/ecp19157491. URL: <http://dx.doi.org/10.3384/ecp19157491>.
- Hindmarsh, Alan C, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward (2005). "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3, pp. 363–396. DOI: 10.1145/1089014.1089020.
- K. E. Brenan S. L. Campbell, Linda R. Petzold (1989). *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. North-Holland / Elsevier.
- MAP FMI (2026). *Layered Standard for Differential Algebraic Equations (unreleased)*. Tech. rep. Unreleased draft. Modelica Association. URL: <https://modelica.github.io/fmi-ls-dae/main/>.
- Modelica Association (2023). *Modelica – A Unified Object-Oriented Language for Systems Modeling*. 3.6. Accessed: 2026-04-10. Modelica Association. URL: <https://specification.modelica.org/>.
- Modelica Association (2024-11). *Functional Mock-up Interface Specification Version 3.0.2*. Tech. rep. Linköping: Modelica Association. URL: <https://fmi-standard.org>.
- Pfeiffer, Andreas, Christoph Winter, Fabian Jarmolowitz, and Christian Bertsch (2025). "Optimization with FMI and CasADi: Analysis in Industrial Applications". In: *16th International Modelica & FMI Conference*. URL: https://fmi-standard.org/assets/literature/Modelica_and_FMI_Conference_2025/Bosch_DLR.pdf.
- Robertson, H. H. (1966). "The Solution of a Set of Reaction Rate Equations". In: *Numerical Analysis: An Introduction*. Ed. by J. Walsh. Academic Press, pp. 178–182.
- Wächter, A. and L.T. Biegler (2006). "On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming". In: *Math. Program.* 106.1, pp. 25–57.

A Modelica-Based Cyber-Physical Simulation Architecture for Firmware-in-the-Loop Testing of Robotic Vehicles

Minhyun Cho¹ Minjoon Lee¹ Micah K. Condie¹ Michael P. Rooney² Evan R. English² James M. Goppert¹ Arslan Khan³ Dongyan Xu² Inseok Hwang¹

¹School of Aeronautics and Astronautics, Purdue University, United States,
{mhcho, lee5826, condie, jgoppert, ihwang}@purdue.edu

²Department of Computer Science, Purdue University, United States, {rooney13, englis20, dxu}@purdue.edu

³School of Electrical Engineering and Computer Science, The Pennsylvania State University, United States,
{arslankhan}@psu.edu

Abstract

Firmware-level cyber-physical testing of robotic vehicles often requires the same firmware to be evaluated under different physical models, fidelities, disturbances, and sensor or actuator faults. In simulator-centric workflows, these physical models and effects are often distributed across simulator-specific plugins, model descriptions, and middleware, making the physical plant difficult to modify independently and complicating formal-methods workflows. This paper presents a Modelica/FMI-based architecture that represents the physical plant as a replaceable FMI 3.0 Co-Simulation FMU behind a stable firmware interface. Vehicle dynamics, actuators, sensors, and environmental effects are modeled in Modelica and exported through Rumoca, allowing physical equations and model fidelity to change without modifying the firmware execution harness. Our physics backend, COURBET, hosts the FMU and couples it to unmodified ArduPilot firmware executing in FastDyn's QEMU-based firmware-in-the-loop environment. FastDyn uses instruction-counted virtual board time, rather than independent wall-clock progression, to synchronize firmware execution and FMU advancement. We demonstrate the proposed architecture on copter, rover, and fixed-wing configurations.

Keywords: *Modelica, FMI-3.0, Functional Mock-up Unit (FMU), Firmware-in-the-Loop (FITL), Cyber-Physical Systems, Robotic Vehicle Simulation, ArduPilot*

1 Introduction

Robotic vehicle simulation platforms such as Gazebo (Koenig and Howard 2004), CARLA (Dosovitskiy et al. 2017), JSBSim (Berndt 2004), AirSim (Shah et al. 2018) and Isaac Sim (NVIDIA 2023) have been widely adopted commercial and research platforms for developing, testing, and validating autonomous systems. Gazebo, for instance, provides rigid-body physics, collision detection, rendering, and actuator and sensor models through an open-source robotics simulation stack, while Isaac Sim provides high-fidelity rendering, perception simulation, and synthetic-data generation through NVIDIA Omni-

verse. These tools are particularly effective when the primary objective is to reproduce rich environments, evaluate perception and navigation behavior, or integrate robotic software with middleware such as the Robot Operating System (ROS).

Firmware-level cyber-physical testing, however, places a different emphasis on the simulation architecture. Such experiments often require the same firmware to be executed repeatedly while physical assumptions, model fidelity, environmental conditions, or sensor and actuator fault mechanisms are varied in a controlled manner. This is particularly important for parameter sweeps, falsification, robustness analysis, and other formal testing workflows in which the physical model itself becomes an experimental variable. In simulator-centric implementations, vehicle dynamics, actuator and sensor behavior, perturbation logic, and their firmware interfaces may be distributed across simulator plugins, SDF/URDF model descriptions, message interfaces, and middleware processes. Extending or replacing a physical model can therefore require changes not only to the governing physics but also to simulator-specific integration code (Pant et al. 2023; Cacace et al. 2020). Consequently, it becomes difficult to modify the mathematical plant independently, to systematically substitute models of different fidelity, or to reproduce an experiment using a self-contained physical-model artifact.

The Functional Mock-up Interface (FMI) can provide a complementary abstraction for this class of experiments (Junghanns et al. 2021). FMI packages a dynamic model, its interface metadata, executable implementation, and supporting resources into a Functional Mock-up Unit (FMU). Combined with an equation-based modeling language such as Modelica (Wetter 2009), this allows the physical assumptions of a system to be expressed through equations, parameters, events, and component connections rather than being embedded primarily in simulator-specific procedural plugins. A vehicle model can therefore be modified by changing its governing equations or subsystem models and re-exported behind the same FMI interface contract. This separation provides useful prop-

erties for experimentation. First, the plant can be customized without modifying the firmware execution harness as long as the interface contract is preserved. Second, models of different fidelity can be substituted while maintaining the surrounding firmware and experiment configuration. Third, the FMU packages the physical model and its interface into a portable artifact that can be regenerated from pinned Modelica, Rumoca, and FastDyn revisions, facilitating reproducible simulation configurations. FMI provides an explicit and replaceable mathematical boundary between firmware execution and physical simulation rather than guaranteeing greater physical accuracy compared to conventional robotic simulators.

This paper presents a Modelica/FMI-based architecture for firmware-integrated robotic vehicle simulations. The term *firmware-integrated* does not mean that the FMU is embedded into the firmware binary or executed on physical hardware. Instead, FastDyn executes an unmodified ArduPilot firmware binary inside a QEMU-based emulation of the target-board environment, while the physical plant simulation uses an external FMU hosted by COURBET. This configuration exercises hardware-facing firmware execution paths such as board-timer interrupts, driver-level sensor and actuator accesses, and interactions with emulated peripheral interfaces that are abstracted in conventional host-native software-in-the-loop (SITL) execution. Thus, the presented firmware-in-the-loop (FITL) execution is not constrained to wall-clock real time; instead, firmware and plant execution are coordinated under a common virtual-time semantics derived from the emulated board time. The firmware and FMU therefore do not progress as independently clocked simulations. The same separation between the firmware-facing interface and the external physical plant also provides a path toward future hardware-in-the-loop configurations, although physical hardware execution is outside the scope of the experiments presented here.

Based on this architecture, we transition the physical simulation substrate of our previous framework (Bashir et al. 2026) from a Gazebo-centered implementation to an FMI-based plant interface. Vehicle dynamics and associated physical subsystems are described in Modelica and compiled through Rumoca (Condie et al. 2025) into FMI 3.0 Co-Simulation FMUs. COURBET hosts and advances the FMU-based physical model, while FastDyn manages QEMU-based firmware execution, virtual board time, and the firmware-facing interfaces used to exchange actuator commands and sensor information. Because the physical model is isolated behind the FMI interface, the same firmware execution architecture can be coupled to different vehicle configurations, model fidelities, and physical perturbation models without rewriting the QEMU firmware harness. The current implementation supports Rumoca-generated FMI 3.0 plants for ArduCopter, ArduRover, and ArduPlane configurations (ArduPilot Development Team 2025), while Gazebo remains available as a legacy or complementary backend.

The contribution is twofold. First, we introduce an FMI-based physical-plant boundary for firmware-level cyber-physical testing, allowing physical equations, model fidelity, and fault or disturbance models to be changed independently of the firmware execution harness while preserving a common interface. Second, we develop a virtual-time coupling architecture in which QEMU-rehosted firmware provides the board-time reference used to advance an FMI Co-Simulation plant, thereby preserving firmware-side scheduling and hardware-facing execution while avoiding dependence on independently progressing wall-clock simulation processes. We demonstrate the same execution architecture across copter, rover, and fixed-wing vehicle configurations and provide a configuration-driven Modelica-to-FMU workflow that supports replaceable and reproducible physical-model artifacts for cyber-physical testing.

The remainder of the paper is structured as follows. Section 2 presents the overall architecture of the proposed firmware-integrated simulation framework, including the virtual-time execution model and firmware-physics synchronization mechanism. Section 3 describes the Modelica-to-FMU workflow using Rumoca, including FMI-3.0 export and integration into the firmware-in-the-loop environment. Section 4 demonstrates the framework on ArduPilot, highlighting configurable vehicle dynamics. Section 5 discusses current limitations, implementation considerations, and directions for future work in cyber-physical testing and firmware-integrated simulation.

2 Simulation Architecture

The proposed architecture moves the physical simulation layer from a simulator-centric implementation to an FMI-centric physics representation. The motivation for this transition is not the absence of physical simulation capability in existing robotic simulators. Simulators widely adopted for robotic vehicle firmware development, such as Gazebo, already provide mature physics engines, rendering capabilities, sensor models, and actuator models. The limitation arises when the simulator becomes the primary location where plant dynamics, sensor behavior, perturbation logic, and firmware interfaces are implemented. In such architectures, the mathematical plant model becomes tightly coupled with simulator-specific backend application programming interfaces (APIs) and plugin implementations. As a result, changing the model fidelity, adding sensor fault/attack models, or replacing vehicle dynamics often requires rewriting simulator plugins rather than exchanging a well-defined component artifact.

Figure 1 illustrates this architectural shift. In the conventional simulator-centric architecture, the vehicle model, actuator interface, sensor interface, and firmware coupling logic are distributed across simulator plugins, Gazebo APIs, and SDF/URDF (Simulation Description Format/Unified Robot Description Format) descriptions. Although this architecture is effective for realistic visu-

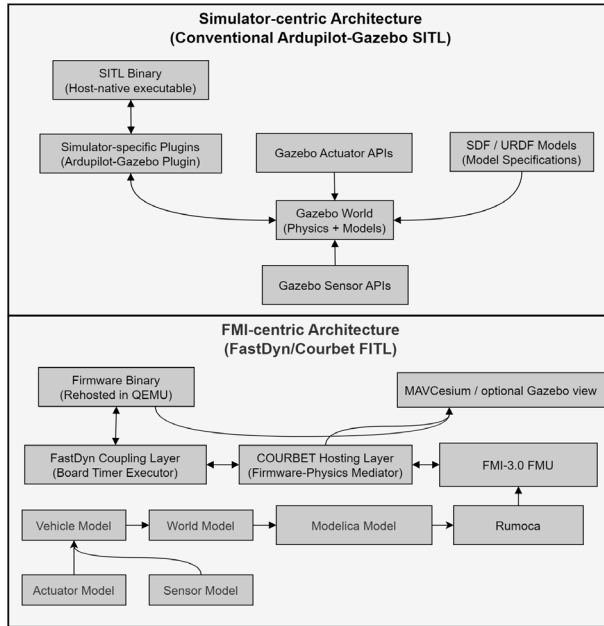


Figure 1. Comparison between a simulator-centric robotic vehicle software-in-the-loop (SITL) and the proposed FMI-centric firmware-in-the-loop (FITL). In the proposed architecture, the physical plant is represented as an FMI-3.0 FMU rather than being embedded in simulator-specific plugins.

alization and robotics middleware integration, the physical plant is not naturally exposed as a portable and replaceable artifact. In contrast, the proposed FMI-centric architecture encapsulates the physical model as an FMI-3.0 FMU. The FMU becomes the authoritative representation of the vehicle dynamics and the physical state from which sensor feedback is generated. The surrounding FastDyn/COURBET coupling layer handles firmware execution, timing, actuator transfer, sensor injection, and optional visualization. Within this layer, FastDyn orchestrates QEMU to provide a board-level execution environment for running the unmodified firmware binary. Rather than modeling the physical vehicle, QEMU hosts the cyber-side firmware execution, while the coupling interface synchronizes firmware actuator outputs and sensor inputs with the FMU-based physical plant.

The resulting architecture separates the cyber and physical concerns of firmware-integrated simulation. The cyber side consists of the firmware binary, QEMU-based execution environment, board timing model, MAVLink communication, and command handling. The physical side consists of the FMU physics model, actuator response, sensor generation, vehicle dynamics, and disturbance models. The coupling layer maps actuator outputs produced by the firmware into FMU inputs and maps FMU states or sensor outputs back into the firmware. This separation makes it possible to replace the physical model without rewriting the firmware harness, and to modify firmware-specific configurations without changing the underlying vehicle dynamics model.

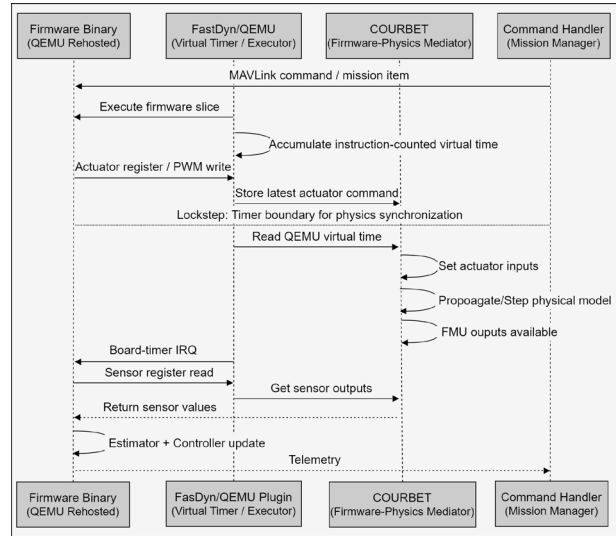


Figure 2. Timer-boundary synchronization between QEMU-hosted firmware, FastDyn/COURBET, and FMU plant. Firmware actuator writes are captured as latest actuator commands. At each board-timer boundary, FastDyn reads QEMU virtual time, advances the plant to that timestamp, and only then raises the board-timer IRQ. Subsequent firmware sensor reads are served from the updated plant outputs.

A key feature of the architecture is that firmware and plant execution are synchronized through board-timer-driven virtual time rather than host wall-clock progression. The firmware runs inside a QEMU environment, while FastDyn maintains an instruction-counted virtual-time model of firmware execution. For fidelity-oriented runs, the framework advances the plant at a configurable board-timer lockstep boundary, such as 1 ms. At each lockstep boundary, the coupling layer reads the most recent actuator commands that the firmware has written to the emulated actuator hardware interface, converts them into FMU inputs, and applies them to the plant. The FMU is then advanced to the next simulation time, and the resulting plant states and sensor outputs are published back to the firmware-facing sensor interfaces. The firmware subsequently observes these updated sensor values through its normal sensor-driver reads. Thus, actuator commands are not generated by the timer tick itself; rather, they are produced asynchronously with respect to the lockstep boundary whenever the firmware control logic writes to the emulated actuator hardware registers. This mechanism allows the firmware, plant, and sensor stream to advance under a common virtual-time semantics, reducing dependence on host scheduling latency and wall-clock real-time factor.

Figure 2 illustrates the execution sequence of the proposed firmware-integrated FMI simulation architecture. The command handler first sends mission items or MAVLink commands to the firmware. QEMU executes a slice of firmware instructions while the FastDyn/QEMU integration layer accumulates instruction-counted virtual

time using the QEMU `icount` mechanism. During this firmware execution, the onboard estimator and controller may read sensor-driver values and write actuator commands to the emulated actuator hardware interface according to the firmware's native scheduling and control logic. When the next board-timer lockstep boundary is reached, the coupling layer transfers the most recent actuator commands from this emulated hardware interface to the FMU-based plant model. COURBET then advances the physical model through an FMI co-simulation step using the current simulation time and communication step size. The updated plant states and sensor outputs, including IMU, GPS, barometer, and actuator-state signals, are returned to the coupling layer and made available through the firmware-facing sensor interfaces. The firmware then consumes these updated sensor values through its normal sensor-driver reads and may produce subsequent actuator commands through later writes to the actuator hardware interface, thereby closing the cyber-physical simulation loop. Telemetry and mission-state information may also be forwarded to mission handling utilities or visualization tools.

Listing 1. Example FastDyn/COURBET TOML configuration binding the QEMU-executed ArduCopter firmware binary to an FMI-3.0 quadrotor model

```
[Machine]
platform = "STM32F427"
icount = { shift = 5, sleep = false,
           align = false }
timer_irq_period_ns = 1000000

[CPU]
arch = "arm"
machine = "cortexm"
cpu = "cortex-m4"
plugin_library = "build/libfastdyn.so"
binary = "virtuals/physics/
           flight_controllers/courbet/bin/
           arducopter_v462"

[FMU]
active = "quadrotor"
auto_build = true

[FMU.models.quadrotor]
model = "FastDyn.Copter"
model_file = "modelica/FastDyn/Copter.mo"
output = "out/fmi3/Copter"

[FMU.models.quadrotor.parameters]
# Purdue University Airport (KLAF)
lat0 = 40.414929
lon0 = -86.932387
ground_alg_wgs84 = 149.0
```

Listing 1 shows an example TOML-based configuration used by the proposed FastDyn/COURBET framework. The configuration binds a QEMU-rehosted ArduCopter firmware binary to an FMI-3.0 quadrotor plant

model generated by Rumoca from a Modelica source model. The same configuration mechanism orchestrates firmware execution, FMU initialization, virtual timing configuration, mission-handler processes, MAVLink communication utilities, and optional visualization backends such as Gazebo or MAVCesium. QEMU instruction-counted virtual time is configured through the `icount` parameters, while the FMU section specifies the active plant model, Modelica source, output artifact location, and environment-specific initialization parameters.

Within the proposed architecture, Gazebo can be retained as an optional visualization and ecosystem-compatibility backend rather than serving as the primary physics and sensor simulation substrate. The authoritative physical representation instead resides within the FMI-compliant FMU artifact. This separation enables the same firmware execution harness to operate with multiple plant implementations, including low-fidelity kinematic models, higher-fidelity multi-body dynamics, and sensor or actuator models with explicit fault and attack mechanisms, provided that the FMI interface contract is preserved.

3 FMU Export and Execution

Rumoca provides the translation layer between high-level Modelica models and executable FMI artifacts. Since Rumoca itself is described elsewhere (Condie et al. 2025), this paper focuses on how its generated FMUs are integrated into the FastDyn/COURBET FITL architecture rather than on the compiler's internal details. At a high level, reusable Modelica models describing vehicle dynamics, actuator behavior, sensors, and environmental effects are compiled into FMI-3.0 co-simulation FMUs that can be loaded by the physics hosting of COURBET. Figure 3 summarizes the resulting integration pipeline.

The current FastDyn/COURBET backend uses the FMI 3.0 Co-Simulation interface rather than Model Exchange or Scheduled Execution. The generated FMU exposes plant parameters, actuator inputs, and sensor/state outputs through FMI value references obtained from the FMU metadata. The interface also makes use of FMI 3.0 array-valued variables to preserve the natural dimensions of vector-valued quantities in COURBET. For example, the four actuator PWM commands for a quadrotor are represented as a four-element array input. At the same time, vector-valued physical states such as the vehicle quaternion are exposed as array-valued outputs rather than being scalarized into independent interface variables. At each firmware-plant synchronization boundary, COURBET writes the current actuator values, advances the FMU using `fmi3DoStep()` with the corresponding communication step, and then retrieves the updated physical and sensor outputs. The experiments in this paper do not use FMI 3.0 Clocks and Scheduled Execution; firmware scheduling and the board-timer interrupt remain part of the FastDyn execution layer. This design intentionally mimics the native scheduling semantics of

the embedded firmware while using FMI Co-Simulation as the interface to the physical plant. FMI 3.0 Clocks and Scheduled Execution remain potential extensions for explicitly scheduled or event-driven physical/control partitions.

Listing 2. Rumoca invocation used by FastDyn to export a Modelica model as an FMI-3.0 target. The configuration is resolved into a model file, Modelica class name, and packaging mode, which are then passed to Rumoca's `compile` command.

```
def cargo_command(build: FmuBuild) ->
  list[str]:
    command = ["cargo", "run"]

    if build.release:
      command.append("--release")

    command.extend([
      "-p", "rumoca", "--",
      "compile", str(build.model_file),
      "--model", build.model])

    for source_root in build.source_roots:
      command.extend(["--source-root", str(
        source_root)])

    command.extend([
      "--output", str(build.output),
      "--target", "fmi3"])

    if build.package:
      command.append("--build")

    return command
```

FastDyn manages FMU generation through a user-defined configuration table shown in Listing 1. When automatic build is enabled, FastDyn resolves the configuration into an internal build specification and constructs the corresponding Rumoca invocation. Listing 2 shows the command construction logic used to generate an FMI-3.0 target from a Modelica source model. The generated FMU is rebuilt automatically when the artifact is missing or stale, allowing simulation configurations to remain reproducible across pinned FastDyn, Rumoca, and Modelica repository revisions.

Internally, Rumoca represents Modelica models as hybrid differential-algebraic equation (DAE) models of the form:

$$\begin{aligned} 0 &= f_x(v, c), \\ z &= \begin{cases} f_z(v, c) & \text{at events } (t = t_e), \\ \text{pre}(z) & \text{otherwise,} \end{cases} \\ m &:= f_m(v, c) \text{ at events } (t = t_e), \\ c &:= f_c(\text{relation}(v)), \end{aligned} \quad (1)$$

where

$$v := [t, \dot{x}, x, y, z, m, \text{pre}(z), \text{pre}(m), u, w, p],$$

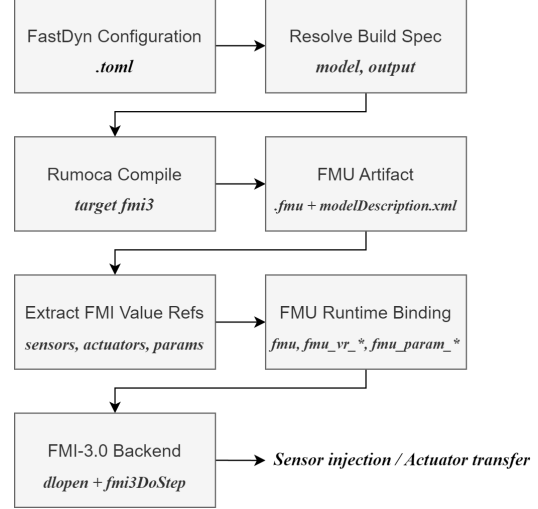


Figure 3. FastDyn FMU export and integration pipeline, from Rumoca-based model compilation to runtime FMI binding for sensor injection and actuator transfer.

and where $x(t)$ denotes the differentiable real states, $y(t)$ algebraic real variables, $z(t_e)$ discrete-time real variables, and $m(t_e)$ discrete-valued event variables. The variables p , t , and t_e represent model parameters, simulation time, and time instance at events, respectively. The symbols u and w denote model inputs and outputs. The operator $\text{pre}(\cdot)$ denotes the value of a variable immediately before an event transition, and c represents the conditions generated from relational expressions and converted `if`-expressions, including `when`-clauses.

The Rumoca compiler pipeline parses Modelica source models using the Rust-based Parol parser generator, which automatically generates parsers from an EBNF grammar while allowing customized parse-tree handling during parsing. The resulting abstract syntax tree (AST) is flattened to remove hierarchical Modelica class structures and lowered into the hybrid-system representation shown above. Continuous dynamics, discrete state updates, relational conditions, and event operators such as `when`, `reinit`, and `pre()` are explicitly preserved before backend-specific FMI or symbolic code generation.

The current workflow exports wrapper models such as Copter, Rover, and Plane for ArduPilot firmware. These wrappers encapsulate firmware-specific interfaces while inheriting reusable rigid-body dynamics and subsystem models from a shared Modelica library. As a result, vehicle-generic physical models can evolve over time independently from firmware integration concerns such as PWM mappings, MAVLink ports, mission helpers, geographic initialization, or sensor routing. The exported FMU therefore acts as a replaceable physical plant artifact behind a stable firmware interface. This separation supports fidelity variation of FMU without modifying the surrounding execution framework. A low-fidelity kinematic approximation, a higher-fidelity rigid-body model,

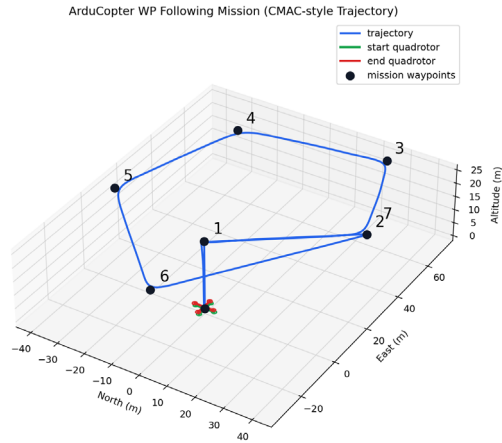


Figure 4. ArduCopter waypoint-following result for the CMAC-style trajectory. The blue curve shows the simulated flight trajectory, black markers indicate mission waypoints, and the green and red airplane markers denote the start and end positions, respectively.

or a fault- or attack-parameterized sensor model can be exported behind the same FMI interface as long as the required inputs and outputs are preserved. Consequently, physical fidelity becomes a property of the FMU artifact rather than a property of simulator-specific plugins or middleware integrations. This allows experiments to vary plant complexity while keeping the firmware, mission logic, and orchestration layer unchanged.

At runtime, the generated FMU is integrated into the FastDyn execution environment through the FMI 3.0 backend implemented within COURBET. The FMU plant executes alongside the QEMU-rehosted firmware and supporting runtime components, including the mission helper, MAVProxy bridge, and optional visualization tools. COURBET coordinates these components within the common virtual-time execution framework, allowing the physical plant to remain replaceable while the surrounding firmware and mission orchestration remain unchanged.

4 Firmware-Integrated Simulations

We evaluate the architecture on three robotic vehicle classes: rover, fixed-wing plane, and multi-copter. These vehicles exercise different aspects of the simulation stack. The rover emphasizes ground interaction, steering, throttle mapping, and low-speed navigation. The fixed-wing plane emphasizes aerodynamic response, attitude stabilization, and mission-level guidance over longer trajectories. The copter emphasizes fast attitude dynamics, motor lag, high-rate inertial sensing, and tight coupling between the firmware control loop and the plant.



Figure 5. MAVCesium visualization of ArduCopter FITL. The interface displays the simulated vehicle attitude, position, and flight status using telemetry.

4.1 Arducopter

For the copter case, FastDyn provides a quick-start workflow that invokes Rumoca to generate the quadcopter FMU, executes ArduCopter inside QEMU, launches the mission handler, and exposes a MAVCesium web visualization endpoint. The default configuration initializes the vehicle on the Purdue University Airport tarmac, uses a 1 ms board tick, maps ArduPilot PWM outputs linearly to aerodynamic motor speeds, and incorporates an explicit first-order motor lag within the FMU plant model. The resulting simulation is presented in Figure 4, while the MAVCesium rendering shown in Figure 5 provides real-time visualization of the vehicle trajectory, attitude evolution, and mission execution state.

4.2 Ardurover

For the rover case, FastDyn generates and executes a ground-vehicle FMU coupled with ArduRover running inside QEMU. The default scenario performs an autonomous waypoint-following mission using the same FMI-based execution pipeline and mission orchestration framework. The rover simulation demonstrates that the proposed architecture can support the nonholonomic ground-vehicle dynamics and closed-loop navigation behaviors, while preserving a unified firmware-integrated simulation interface across vehicle classes.

4.3 Arduplane

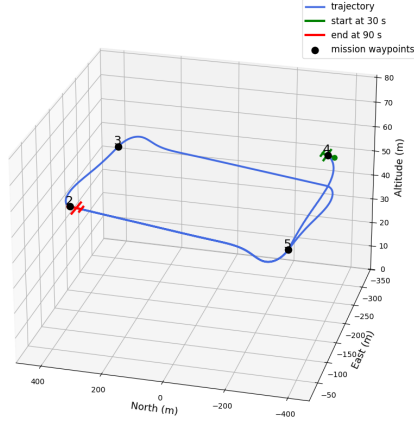
In the fixed-wing case, FastDyn couples a fixed-wing FMU with ArduPlane executing in the QEMU-based firmware environment. The evaluated mission consists of climb, waypoint-following, and descent to exercise trajectory-tracking as shown Figure 6. The resulting simulation demonstrates that the FMI-centric architecture can support the continuous aerodynamic flight dynamics, mission-level autonomy, and FITL without relying on simulator-specific vehicle plugins. However, we have identified a coordinate-frame inconsistency in broadcast-

Table 1. Summary of evaluated vehicle configurations and mission behaviors.

Vehicle	Firmware	MCU	Vehicle Model	Mission
Copter	ArduCopter	STM32F427	Quadcopter (3DR Iris)	TO, WP, LD
Rover	ArduRover	STM32F427	Rover (Aion R1)	WP
Plane	ArduPlane	STM32F427	Fixed-wing (Skywalker X8)	TO, WP, LTR, LD

TO: takeoff, WP: waypoint tracking, LTR: loiter, LD: landing.

ArduCopter WP Following Mission(CMAC-Circuit Trajectory, Window: 30-90 s)

**Figure 6.** ArduPlane waypoint-following result for the CMAC circuit mission over 60 s time window. The blue curve shows the simulated flight trajectory, black markers indicate mission waypoints, and the green and red airplane markers denote the start and end positions, respectively.

ing navigation data into the firmware, which will be addressed in subsequent development.

4.4 Runtime Performance

To characterize the runtime behavior of the proposed Modelica/FMI-based physics backend, we instrumented the ArduCopter configuration and executed a 60 s virtual-time mission five times. For comparison, the same experiment was repeated using the legacy Gazebo-based physics backend. Both configurations use a 1 ms QEMU board-timer period and instruction-counted virtual time. The FMI backend advances synchronously at every board-timer boundary, whereas the legacy Gazebo backend uses an asynchronous catch-up scheme in which Gazebo executes as a separate process and is not advanced in strict lockstep with every QEMU board-timer boundary. Gazebo was executed in headless mode with GUI rendering disabled. For each run, the wall-clock measurement ended when the target QEMU virtual time was reached, before QEMU teardown, thereby excluding process-shutdown latency from the reported execution time. The experiments were conducted on a host equipped with an Intel Core i9-12900K processor and 128 GB of RAM running Linux 6.8.0.

The end-to-end real-time factor (RTF) is defined with

respect to the QEMU virtual board time as

$$\text{RTF}_{\text{E2E}} = \frac{\Delta t_{\text{board}}}{\Delta t_{\text{wall}}}, \quad (2)$$

where Δt_{board} is the elapsed QEMU virtual board time and Δt_{wall} is the corresponding host wall-clock duration. For the FMI backend, Δt_{board} is also the physical-model time by construction because COURBET synchronously advances the FMU to each board-time boundary before firmware execution proceeds. For the Gazebo backend, however, this quantity represents the rate at which QEMU requests simulation progress and does not necessarily equal the instantaneous Gazebo simulation-time rate because the legacy Gazebo coupling advances the simulator asynchronously.

The backend-call fraction is defined as

$$c_{\text{backend}} = \frac{\sum_k T_{\text{call},k}}{T_{\text{wall}}}, \quad (3)$$

where $T_{\text{call},k}$ is the wall-clock duration of the k th instrumented backend call. We additionally report the effective backend real-time factor,

$$\text{RTF}_{\text{backend}} = \frac{\sum_k \Delta t_{\text{advance},k}}{\sum_k T_{\text{call},k}}, \quad (4)$$

which measures the amount of simulation-time advancement per unit of wall-clock time spent in the instrumented backend call. For the asynchronous Gazebo backend, this quantity characterizes the effective catch-up rate observed through the legacy FastDyn-COURBET-Gazebo coupling and should not be interpreted as an isolated benchmark of the Gazebo physics engine.

Table 2 summarizes the results. The proposed FMI configuration achieved an end-to-end RTF of 4.517 ± 0.012 , corresponding to approximately 13.3 s of wall-clock execution for 60 s of QEMU virtual time. The headless Gazebo configuration achieved a similar end-to-end RTF of 4.362 ± 0.014 . This similarity does not imply comparable physical-backend computational cost. The FMI backend advances synchronously every 1 ms, with a mean `fmi3DoStep()` duration of $3.338 \pm 0.138 \mu\text{s}$ and a 99th percentile (P_{99}) duration of $5.824 \pm 0.165 \mu\text{s}$. FMU advancement accounts for only $1.508 \pm 0.065\%$ of the measured wall-clock execution time.

In contrast, the legacy Gazebo backend advances using an asynchronous catch-up mechanism, with an average

Table 2. Runtime characteristics of the proposed FMI-based and baseline Gazebo-based ArduCopter FITL configurations.

Metric	Proposed FMI-based backend	Baseline (Gazebo, Headless)
Synchronization architecture	Synchronous	Asynchronous catch-up
QEMU board timer step	1 ms	1 ms
Backend advance interval	1 ms	9.333 ± 0.095 ms
End-to-end RTF (RTF_{E2E})	$4.517 \pm 0.012 \times$	$4.362 \pm 0.014 \times$
Backend-call fraction (c_{backend})	$1.508 \pm 0.065\%$	$51.198 \pm 0.604\%$
Mean backend-call wall time	$3.338 \pm 0.138 \mu\text{s}$	$1085.339 \pm 26.710 \mu\text{s}$
P_{99} backend-call wall time	$5.824 \pm 0.165 \mu\text{s}$	$8384.978 \pm 4229.015 \mu\text{s}$
Effective backend RTF ($\text{RTF}_{\text{backend}}$)	$299.6 \times$	$8.60 \times$

Values are reported as (mean) $\pm$ (standard deviation) over five 60 s virtual-time runs.

backend advance interval of 9.333 ± 0.095 ms. Its instrumented backend calls require $1085.339 \pm 26.710 \mu\text{s}$ on average and account for $51.198 \pm 0.604\%$ of the measured wall-clock time. Nevertheless, much of this computation does not lie serially on the QEMU execution critical path because Gazebo and QEMU progress asynchronously and may execute concurrently on separate host cores. Consequently, the end-to-end RTF remains close to that of the FMI configuration despite the substantially larger Gazebo backend cost.

The FMI backend achieves an effective backend RTF of $299.6 \times$, compared with $8.60 \times$ for the Gazebo backend. These values characterize the simulation-time advancement achieved per unit wall-clock time spent in the respective backend calls. For the FMI backend, this directly reflects the synchronous `fmi3DoStep()` execution cost. For the Gazebo backend, the backend call corresponds to the instrumented Gazebo bridge/backend-update operation. Because the Gazebo backend advances through an asynchronous catch-up interface, the reported effective backend RTF characterizes the observed Gazebo catch-up rate rather than the standalone RTF of the Gazebo physics engine.

4.5 CPS Testing and Implications

Table 1 summarizes the vehicle class, firmware version, FMU model, target board, and mission type. The proposed FMI-based simulation architecture is intended not only for single-mission demonstrations, but also as a reusable cyber-physical testing workflow for firmware-integrated robotic systems. The vehicle simulations presented in this section demonstrate representative execution scenarios within a broader experimentation framework that can be extended to Monte-Carlo simulation campaigns, parameter sweeps, firmware fuzzing, coverage-guided exploration, and falsification-oriented robustness analysis. Because the physical plant is encapsulated behind a stable FMI interface, perturbation inputs, parameter mutations, sensor faults, environmental variations, and model substitutions can be introduced without modifying the firmware execution harness itself. At the same time, the FastDyn/COURBET execution layer provides access to cyber-domain observations including firmware traces,

MAVLink traffic, timing behavior, and execution coverage. Consequently, the architecture supports scalable formal-methods-oriented workflows in which physical-system perturbations and firmware-level exploration can be combined within a unified execution environment.

5 Discussion and Limitations

The proposed architecture should be understood as complementary to, rather than a complete replacement for, existing robotic simulators. In our case, Gazebo can remain as a strong choice for contact-rich scenes, perception simulation, synthetic data generation, rendering, and integration with robotics middleware. The advantage of the FMI-based architecture is strongest when the experimental objective requires a replaceable mathematical plant model, deterministic firmware-plant synchronization, formal perturbation interfaces, or scalable execution.

The fidelity of an FMI-based simulation is not guaranteed by FMI alone. Accuracy depends on the Modelica equation model, the Rumoca translation, solver settings, event handling, communication step size, actuator and sensor models, and the quality of the firmware interface mapping. Therefore, the contribution of this paper is architectural rather than claiming that our framework is universally more effective than Gazebo. The key claim is that Modelica/FMI provides a cleaner model boundary for firmware-integrated cyber-physical system (CPS) experimentation campaigns.

Another limitation is that high-fidelity terrain, contact-rich environments, and perception models may still be easier to implement in existing robotics simulators. For this reason, the proposed architecture keeps Gazebo as an optional backend and compatibility path. In future work, a hybrid configuration could use FMUs for vehicle dynamics and firmware synchronization, while delegating selected perception or contact-rich subsystems to other simulators, such as Gazebo and Isaac Sim.

6 Conclusion

This paper presented a Modelica/FMI-based cyber-physical simulation architecture for firmware-integrated robotic vehicles. By transitioning from simulator-specific

plugins to Rumoca-generated FMI-3.0 FMUs, the proposed architecture decouples reusable vehicle dynamics models from simulator-specific integration logic and platform-dependent interfaces. Our FITL framework, FastDyn and COURBET, couples these FMUs to QEMU-based firmware execution using instruction-counted virtual time, enabling effective mission execution, model substitution, and scalable campaign orchestration.

Our approach was demonstrated on copter, rover, and plane configurations, showing that the same firmware-integrated execution framework can support multiple robotic vehicle classes through FMU configuration changes. The architecture is particularly suitable for experiments that require controlled fidelity variation, custom sensor or actuator with explicit fault and attack mechanisms, falsification interfaces, and reproducible parallel campaigns. Future work will expand the model library, improve automated validation against Gazebo and real-flight traces, and investigate hybrid backends that combine FMI-based physics with simulator-based rendering and perception.

Acknowledgements

This work is based on research sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-24-2-0500. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

- ArduPilot Development Team (2025). *ArduPilot*. <https://ardupilot.org>. Accessed: 2026-05-10.
- Bashir, Muhammad Hammad, Michael Rooney, Colin Smith, Dongyan Xu, and Arslan Khan (2026). “BoardRunner: Automatic Firmware Rehosting using High-Fidelity Compositional Device Models”. In: *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*. The Hague, The Netherlands: Association for Computing Machinery.
- Berndt, Jon S. (2004). “JSBSim: An Open Source Flight Dynamics Model in C++”. In: *AIAA Modeling and Simulation Technologies Conference and Exhibit*. DOI: 10.2514/6.2004-4923.
- Cacace, Jonathan, Nicola Mimmo, and Lorenzo Marconi (2020). “A ROS Gazebo plugin to simulate ARVA sensors”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 7233–7239.
- Condie, Micah, Abigaile Woodbury, James Goppert, and Joel Andersson (2025). “Rumoca: Towards a Translator from Modelica to Algebraic Modeling Languages”. In: *Modelica Conferences*, pp. 1009–1016.
- Dosovitskiy, Alexey, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun (2017). “CARLA: An Open Urban Driving Simulator”. In: *Proceedings of the 1st Annual Conference on Robot Learning (CoRL)*, pp. 1–16.
- Junghanns, Andreas et al. (2021). “The functional mock-up interface 3.0-new features enabling new applications”. In: *Modelica conferences*, pp. 17–26.
- Koenig, Nathan and Andrew Howard (2004). “Design and use paradigms for gazebo, an open-source multi-robot simulator”. In: *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*. Vol. 3. Ieee, pp. 2149–2154.
- NVIDIA (2023). *NVIDIA Isaac Sim*. <https://developer.nvidia.com/isaac/sim>.
- Pant, Kartik A, Zhanpeng Yang, James M Goppert, and Inseok Hwang (2023). “An open-source gazebo plugin for gnss multipath signal emulation in virtual urban canyons”. In: *AIAA SCITECH 2023 Forum*, p. 2586.
- Shah, Shital, Debadepta Dey, Chris Lovett, and Ashish Kapoor (2018). “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles”. In: *Field and Service Robotics*.
- Wetter, Michael (2009). “Modelica-based modelling and simulation to support research and development in building energy and control systems”. In: *Journal of Building Performance Simulation* 2.2, pp. 143–161.

Implementation of a High Fidelity Model of a Racecar into a Path Optimization Problem using Functional Mock-Up Units

Angel Mario Zarate Villazon¹ Dr. Michael Balchanos¹ Prof. Dimitri Mavris¹

¹Aerospace Systems Design Laboratory, Georgia Institute of Technology, United States of America
amzv3@gatech.edu, michael.balchanos@asdl.gatech.edu,
dimitri.mavris@aerospace.gatech.edu

Abstract

The problem of finding the optimal control for a racecar is a topic that has been studied practically since car racing began. However, this problem is usually solved using many assumptions to simplify it. There have been very few implementations that use a high-fidelity model such as those implemented for a driver-in-the-loop simulator, which are commonly written using the Modelica language. In this paper, a series of tests to establish the feasibility of implementing these high-fidelity models in a collocation optimization solver through model order reduction and the Functional Mock-Up Interface Standard is presented. The model is used by an OEM in their driver-in-the-loop simulator in an international top-tier motor-sport series. Collocation optimization and the reduced-order model leverage the software package Dymos, which solves an optimal control problem using a model formulated with OpenMDAO. These feasibility experiments are divided between those related to the model and the optimizer and demonstrate the feasibility of using the FMU. The results show that a FMU representing the high-fidelity model can be implemented in Dymos. However, problems remain in the evaluation of the derivatives for the simplified model exposed to Dymos, which causes divergence problems in the optimization. Overall, this paper establishes the aspects that endanger the feasibility of the implementation of a high-fidelity model in a collocation optimization solver to determine the optimal path around a racetrack.

Keywords: collocation optimization, high-fidelity models, functional mock-up units, automotive engineering

1 Introduction

The problem of predicting a racing car's lap time by finding the optimal trajectory across a racing track is significant in car and motorcycle racing. Multiple racing car drivers and engineers recognize the importance of identifying the fastest trajectory around the track. While drivers instinctively try to find this path with their driving style, engineers have been attempting mathematically to compute this optimal trajectory since car racing started. Massaro and Limebeer provide an extensive investigation of the history and technical developments on the solution for

this problem (Massaro and D. J.N. Limebeer 2021). Two main approaches have emerged related to finding the optimal trajectory: reinforcement learning and optimal control (Song et al. 2023). While reinforcement learning has been applied to trajectory optimization problems with complex multibody systems defined in Modelica (Reiner 2025), this paper focuses on optimal control techniques. Additionally, it has been identified that the problem approach and its corresponding fidelity are determined from three main areas: vehicle parameters, track parameters, and how to represent driver behavior.

From the vehicle point of view, most of the work goes into representing the parameters of the car in full detail. Perantoni and Limebeer provide a solution to the problem of analyzing the effect of multiple vehicle parameters on the vehicle's behavior (Perantoni and David J.N. Limebeer 2014), while other analyses focus on the sensitivity the tire model parameter selection and uncertainty in the optimal control solution (Zarate Villazon et al. 2026). Similarly, Limebeer presents a study of the impact of asymmetries in a NASCAR racecar to the optimal control problem (D. J.N. Limebeer et al. 2023). These studies are similar since the vehicle is being modeled as a rigid body, simplifying the problem to a certain extent. However, some adjustments were made to increase the fidelity and representativeness of the solution.

The next area of interest for these problems concerns the track surface modeling. Christ, et al. showcased that solutions to the optimal control are highly sensitive to changes in the grip coefficients in the track (Christ et al. 2021). Furthermore, additional studies on the track model, especially concerning cambering are presented in the Limebeer study of a NASCAR vehicle (D. J.N. Limebeer et al. 2023) highlighting the importance of accurate track representations.

Finally, there are multiple advances in the driver modeling side. While optimal control advances take the driver out of the loop and produce an optimal solution only mathematically, there is still interest in understanding how human drivers operate the vehicles. Lockel suggests an approach where machine learning is used to understand how professional drivers operate racing cars (Löckel et al. 2022). Once a model is established, it is deployed in a vehicle simulator and used to calculate optimal path and

lap time. Additionally, Kegelmann highlights that there are initiatives to maintain a dataset of different driving styles and techniques that could motivate the development of a future driver model (Kegelmann et al. 2017).

This project centers around determining the feasibility for the implementation of high-fidelity vehicle models written in Modelica in a trajectory optimization solver called Dymos (R. Falck et al. 2021). The demonstration use case centers around a flat model of Watkins Glen and the implementation of the vehicle model written in Modelica and used in a driver-in-the-loop simulator. Two threads to determine the feasibility of the approach have been identified: one related to the model itself and one related to the optimizer. From a model point of view, the interest lies in seamlessly integrating the optimization environment into the modeling workflow to ensure a single source of truth approach. This means that the same model in the simulator that professional drivers use to train is implemented in the trajectory optimization solver. This is to have an identical basis for the comparison of the results obtained by the optimizer with the behavior professional drivers can enforce. Another essential requirement is to use mathematics alone to provide the optimal path, with no other controllers involved in solving the trajectory optimization. Finally, there should be an opportunity for scaling this problem to higher fidelity track definitions.

The second thread of these interests is related to the optimizer selection. The main requirement is to estimate the lap time given a vehicle configuration. This has a complementary requirement where the optimizer needs to be robust enough to accept different vehicle configurations and models. Finally, the optimizers have to be tolerant to nonlinearities and discontinuities, which are abundant in this high-fidelity vehicle model. The paper is structured with a background section providing detailed information about the model and the optimization approach, an explanation of the methodology and the planned tests to establish feasibility, an analysis of the results of these feasibility tests, and a set of conclusions pointing to remaining work in this subject.

2 Background

2.1 High-Fidelity Models

This project leveraged the Modelon Vehicle Dynamics Library (Griffin et al. 2012). This is a library built in Modelica and implemented in Dymola. The model under study is used in a professional motorsports team and supports its driver-in-the-loop simulations.

A key characteristic about the implemented model is that it contains detailed vehicle suspension compliances. Literature shows that suspension compliances are hardly ever implemented for optimal lap time simulations (Masaro and D. J.N. Limebeer 2021). This is because of the challenge of derivative calculations in dynamic vehicle models, specifically in the suspension parameters affected by this assumption.

Additionally, the model is implemented as an open-loop driver experiment. The open-loop driver experiment is a formulation where there is no feedback from any element in the vehicle as shown in Figure 1. This makes it particularly useful to enforce external driver behaviors on it, as it only models the interaction of the vehicle with a respective ambient model and track model. The track model can be approached with multiple levels of fidelity, with OpenCRG (Su et al. 2019) being the standard for high-fidelity track representation. However, for this initial attempt a flat surface simple model is used as the track model. Overall, this experiment sets the model to respond to numerical inputs representative of the virtual operation of the vehicle (for example: steering wheel angle, throttle pedal position, braking force, clutch pedal position). However, even with these simplifying assumption, the model contains more than one hundred thousand parameters to fully define the state of the vehicle.

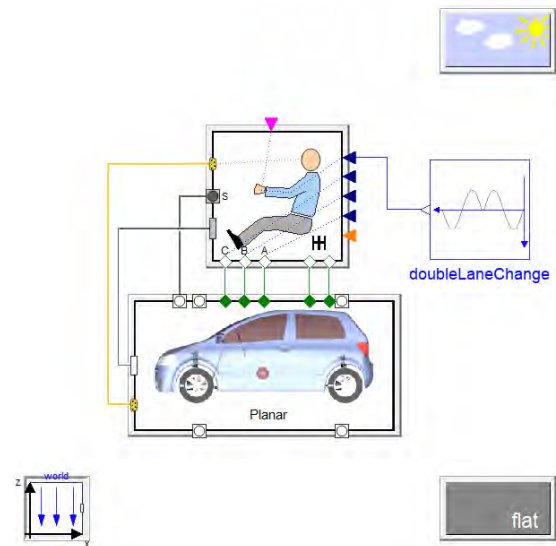


Figure 1. Open-loop driver model experiment representation for a prescribed double lane change maneuver

Two main complexities arise from this formulation. The first relates to the dynamic nature of the model, which necessitates the consideration of the vehicle's states through time. Additionally, there needs to be a way to exercise this model in software environments different from Dymola in order to be able to implement a range of optimization solutions. To help with this situation, the model is exported as a Functional Mock-Up Unit (FMU). FMUs are built according to the Functional Mock-Up Interface standard (Modelica Association 2023), which allows for the execution of exported models in software environments different from those where these experiments were modeled. In particular, these models are compiled versions of the experiment, accompanied by metadata about the inputs, outputs, and connections within the model. There are two main ways in which the FMU can be exported: co-simulation and model exchange. The main

difference lies in whether the FMU has a solver embedded or not. Co-simulation FMUs include a solver while model exchange FMUs do not. For this research, the co-simulation FMU is used to explore the feasibility of the approach before committing to the integration of an external solver to this problem.

In summary, this project focuses on implementing a high-fidelity model of a racing vehicle that is used in professional-grade Driver-In-the-Loop simulators into an optimal control solver to obtain its optimal path around a track. This model was prepared as an open-loop driver experiment in Dymola and exported as a co-simulation FMU. This exported FMU is executed in external tools that enable its use in other problems such as this optimal control problem to estimate the lap time. As a consequence of the open-loop driver experiment decision, the model contains a large number of states, making the final state highly dependent on the time history of each of these states. Furthermore, some of the characteristics defined by the fidelity of the model add significant complexity to the optimal control problem, especially in the derivative definition.

2.2 Optimal Control

Optimal Control is a widely studied subject that has applications beyond motorsports. The classical formulation for the problem is to minimize a cost function that depends on a dynamical system and its control function. It has multiple applications in engineering and even beyond engineering, with some applications being reported in fields such as finance. Across the literature identified in the introduction, three main methods for the solution of optimal control problems have been established: multiple shooting, direct collocation and Model Predictive Control. Model Predictive Control is a method that is at the forefront of autonomous racing applications (Wischniewski et al. 2022). Additionally there have been attempts to use this approach in Modelica models such as the racing car (Imsland et al. 2010).

Two main methods are commonly used to estimate lap time: multiple shooting and direct collocation. The main difference between these methods lies in the discretization approach for the problem's solution. Massaro and Limebeer (Massaro and D. J.N. Limebeer 2021) describe multiple examples of how these methods are implemented for the estimation of lap time in racing cars. However, this project centers on the use of direct collocation.

Direct collocation is a method that discretizes the trajectory optimization problem by converting the original problem into a nonlinear program through a process called transcription (Kelly 2017; Becerra 2013). This is achieved by approximating all of the continuous functions in the problem statement as polynomial splines. Polynomial splines offer the advantage of computing integrals and derivatives based on a relatively small number of polynomial coefficients. The nonlinear program proposes a new solution in each iteration until it reaches a certain toler-

ance. In particular for this project, direct collocation is solved through Dymos.

The optimal control problem is formulated as a minimum lap time problem, with constraints applied to the grip levels of the vehicle, as well as kinematic constraints on the track limits. It is solved in Dymos, a python-based package that builds on top of the OpenMDAO architecture to solve optimal control problems (R. D. Falck and Gray 2019; R. Falck et al. 2021). This provides a framework to easily implement the hybrid approach for the vehicle model being proposed as it leverages the derivative handling characteristics embedded in the Dymos architecture while relying on powerful solvers for these problems like IPOPT (Wächter and Biegler 2006). Additionally, Dymos already has tested reduced order model vehicle formulation presented used for aerodynamic (Buck and J. R. A. Martins 2023) and tire model sensitivity (Zarate Villazon et al. 2026) analyses. This model is taken as a starting point for the development of this research and tailored and expanded for the specific applications under investigation.

2.3 Research questions and capability gap

In order to achieve the objective defined, two main questions need to be addressed in this research effort. The first question is: How to use the open-loop driver experiment in a discretized manner so that it can be used in collocation optimization? This question is primarily driven by the interest in using collocation optimization and the exact model used in the DIL simulator. The second question is: Can the model complexity be reduced in an intermediate step so the optimal control problem is simpler?

The use of the open-loop driver experiment requires a setup process to be able to extract information from it. As mentioned earlier, this formulation requires the inclusion of thousands of parameters to define the whole vehicle state. When it comes to using the model in a discretized manner, there would be three options: to use the optimizer to handle all the thousands of parameters, to use a quasi-steady state solver, or to use an external aid to save the states of the model. Using the optimizer to save all the states is infeasible as the number of states is so large that it would cause memory issues in the optimization code. The use of a quasi-steady state solver to approximate the state is a possibility that has been explored with limited success. The third option that has been identified is called FMU serialization (Modelica Association 2023). This method consists of getting the state of a simulation at a certain point and saving the respective state into a file to restart the simulation at a later stage.

The second question lies in the capacity of reducing the model complexity so the problem handled by the optimizer is simpler. This can be achieved by using the calculations performed by the FMU to inform the states of a lower fidelity model such as the one in the example problem from Dymos. However, this creates questions around how to properly use the FMU to achieve this, specifically around the sensitivity, timestep compatibility, and the fact

that a proper connection would need to be approached not only from the values, but also from the derivatives.

The problem with the derivatives is particularly relevant for this problem. Direct collocation is a method to solve optimal control problems that leverages a model's sparsity. Sparse models are those that have a large fraction of their parameters set to zero. This is particularly relevant when it comes to defining the Jacobian for our model. There have been different approaches that help define Jacobians sparsely using the Functional Mock-Up interface standard (Åkesson et al. 2012). However, due to the fact that the model contains so many suspension compliances and filtered data from look-up tables, the approach to obtain derivatives directly from the FMU is challenging. The alternative explored in this work is to obtain the derivatives in the reduced-order model as it also aligns with the choice to test a co-simulation FMU. Additionally, this leverages the techniques developed in OpenMDAO to couple derivatives from heterogeneous numerical models (Hwang and J. R. Martins 2018). In particular, the derivatives for the reduced-order model can be obtained through a finite-differencing algorithm or through automatic differentiation (Güne, et al. 2018). In summary, the selection of derivatives for this problem requires thoughtful consideration, as each solution presents tradeoffs between feasibility, accuracy and practicality.

Overall, these questions are linked to relevant gaps for the industry that make this problem relevant for entities beyond the specific scope of this problem. One of the identified gaps is the use of these high-fidelity models in a wider variety of applications. The research questions proposed here can aid in expanding the use of Modelica-based high-fidelity models. Serialization can provide a way to keep using a high-fidelity model when performing design space exploration around a specific state. Additionally, the use of these high-fidelity models represented by FMUs in reduced-order models can help architect models with multiple levels of fidelity.

3 Method

The method section is divided into two subsections. The first describes the method for the preliminary experiments that enable the determination of the feasibility of the approach. The second provides a pilot implementation of the general solution of the problem. The FMUs described throughout the section were executed in Python through the FMpy library (Sommer 2023). For the purposes of this paper, the OpenMDAO model that is executed in Dymos can also be referred as the reduced-order model or the lower-fidelity model. This model has three degrees of freedom (two translational and one rotational) and integrates kinematics related to the position of the vehicle against the centerline of the track as well as dynamic properties related to tire force responses and damping characteristics in the vehicle. This is the calculation block referred to as *Lower Fidelity Car Model* in Figure 2. The

model created in Dymola using the Vehicle Dynamics Library is also referred to as the high-fidelity model. As this higher fidelity model can better represent the dynamic characteristics of the vehicle, the emphasis of the integration was to solve for specific states in the lower fidelity by executing the FMU, instead of the direct equations of motion. This allows for information about the dynamics to be computed by the FMU but maintaining the simplified kinematics of the lower fidelity model. This is the calculation block referred to as *Car FMU* in Figure 2 and the specific states being solved following this approach are detailed in the gray blocks at the intersection of the blocks referring to the models. By approaching this problem with two separate models, the kinematic information about the track's parameters and their evolution is maintained, providing a tested basis and controlling directly where fidelity is increased by adding dynamic effects calculated by the FMU directly.

3.1 Preliminary experiments that enable the general approach

A series of feasibility tests were performed to fully test the model formulation of combining a high-fidelity Modelica-based model with a lower-fidelity model through FMUs. Two main threads have been identified to establish the preliminary experiments: one related to the model and one related to the optimizer. The model-related research focuses on establishing the feasibility of using the FMU to represent higher-fidelity dynamics in the lower-fidelity model. The optimizer-related research focuses on establishing the feasibility of Dymos as the collocation solver, especially in its ability to adapt external models to represent the dynamics of the system. The following paragraphs focus on establishing the method for each of the experiments that would enable the general approach described in Section 3.2.

M1: Verify that serialization is compatible with the open-loop driver formulation

This experiment has the purpose of testing the compatibility of using serialization for the FMU associated with the open-loop driver formulation with the race car. The hypothesis for this experiment is that the simulation would have the same results regardless of how many times it has been stopped and restarted from a serialization file in the process.

To test this assumption, the open-loop driver experiment is simulated based on throttle, gear, clutch, braking, and steering inputs that have been recorded from the DIL simulator. These inputs are parsed into the FMU at a frequency of 100Hz and an arbitrary number of serialization files is created, each of these at an equidistant point in either time or position. The outputs calculated from the FMU for the complete run are saved into a dataframe for future analysis. Then, a simulation is executed, parsing the same inputs that were recorded, only that this simulation is stopped at the respective point in time or position

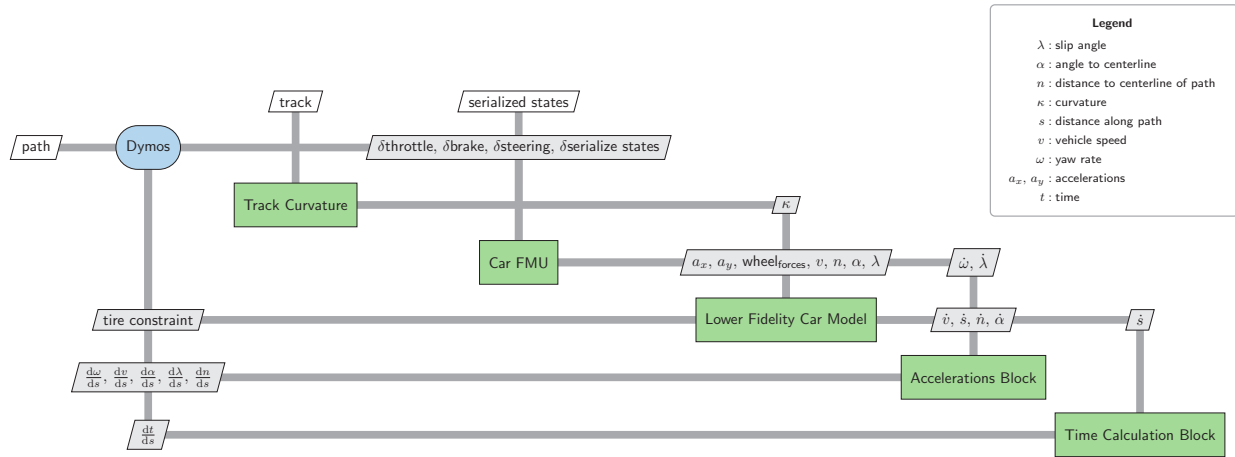


Figure 2. Reduced-order Model Implementation with the vehicle's FMU

where a serialized file was created. There, the FMU is deleted and re-instantiated using the respective serialization file. This process is repeated for every existing point where a serialization file exists. The results for this process are appended in order to assemble the equivalent of a full simulation run. Finally, the results of the full simulation run, the serialized run, and the recording from the DIL simulator are compared.

M2: Determine if the FMU remains sensitive to changes in the controls after being serialized

This experiment has the purpose of determining the sensitivity of the experiment to changes in the controls after being serialized. The validation of this condition is critical for a successful implementation of FMUs in a reduced-order model. Additionally, once this has been established, there is a need to determine a timestep that, on the one hand, allows the controller changes to have an impact on the vehicle's state while, on the other hand, maintains the vehicle at a position representative of the collocation node. These validations provide more information about what is needed to implement the vehicle model's FMU.

To confirm these validations, two different tests are executed. The first consists of fixing the control inputs and initializing the FMU with random serialized files that were generated during the execution of experiment M1. The simulation is executed, and the results are retrieved for different timesteps that go from 0.0005s to 0.0045s. Based on these results, a timestep would be selected. With the selected timestep, variations to the controls are proposed. These variations have the objective of demonstrating the robustness of the approach. Avoiding non-linear behaviors is the main driver for the timestep decision. Seeing variability when the controls are changed will confirm that the timestep selection was correct.

M3: Test the connection of the higher fidelity model with the lower fidelity model that is exposed to the optimizer

This test is part of the debugging effort but already implements the general solution described in Section 3.2. The objective of this test is to determine that a correct connection has been formed between the lower fidelity model and the high fidelity model represented by the FMU. This is especially important as any error in this connection would create a discontinuous state that would spiral into non-convergence of the optimal control problem. An Extended Design Structure Matrix that describes this reduced-order model and how it is implemented with the FMU is shown in Figure 2. The geometric and kinematic variables illustrated in Figure 2 define the vehicle's state relative to the path. It details the specific vehicle states that are computed by the FMU and then passed into the lower fidelity model.

In order to test this connection, the complete problem is executed in Dymos. The main difference is that the limit of iterations is 1, which allows us to identify possible values where the connection is not working. Then, the limit is progressively raised to start testing the effect of the high-fidelity states on the lower-fidelity states. Overall, this process is mostly a debugging tool for the general approach that needs to be complemented with the derivative formulation tests. As described in Section 3.2, every time the model is executed in Dymos, the resulting state is saved again through serialization. To test the evolution of states that Dymos is proposing, a comparison between the data stored in the original serialization files obtained with the DIL data and the data stored in the serialization files of the Dymos optimization results is proposed as a debugging tool to identify potential issues for this problem. In particular, the comparison has to show that the optimizer's model has the same state as the high-fidelity model as it is executed. This ensures that the problem is formulated correctly and that no mistakes arise from the FMU formulation.

O1: Verify that the optimizer can converge with external code

The objective of this test is to establish that the model that interfaces directly with Dymos is compatible with external code executions to calculate its state. The compatibility of Dymos is crucial as the idea to implement the FMU directly hinges on this capability. Additionally, it was particularly relevant to establish its feasibility as there are no examples publicly available where this capability was demonstrated. However, testing this directly with the full vehicle FMU requires additional enablers to work, so this test focuses on a simpler representative problem.

To formulate a representative problem that is simple enough, lower fidelity model's calculation of the lateral force for the tire was replaced by a function call of an external tire model. This model takes the slip angle of the tire and the normal force of the vehicle to estimate the lateral force. A simplifying assumption was made to avoid changing the example model where the body slip angle was used as the tire slip angle in the rear tires and the sum of the body slip angle and steer angle in lieu of the tire slip angle for the front tires. Once the code is implemented, the example model is executed as a standard Dymos problem.

O2: Determine the node location precisely

As explained in Section 2.2, collocation optimization relies on the discretization of the trajectory. Consequently, it is imperative to understand the algorithm that Dymos uses to discretize the track. To fully understand the impact of this algorithm on a racetrack, an experiment is required. The hypothesis is that the nodes are determined equidistantly.

To test this hypothesis, the experiment is executed with the example model. The main difference consists of including specific plots that show where the nodes are located. Additionally, a plot that shows the distance between each of the nodes is used. The distance between nodes is calculated using the Euclidean planar distance.

O3: Testing methods to obtain the derivatives

One of the most important things to determine for this problem is the combination of derivatives that would aid the optimizer in achieving the final result. As explained in Section 2.2, there are three main ways to implement derivatives in Dymos: automatic differentiation through the Google Jax algorithm, finite differencing, and the definition of the analytical derivative calculation. These methods can be mixed and matched as needed for the application being developed. As there is no reference as to which method could work better for each of the different states, not in the literature or in the advice of subject matter experts, the only approach is to test all the possible combinations.

This approach can only be tested by progressively increasing the number of iterations and verifying if it is diverging. Once the optimizer converges, the possible differential of the controls is increased. More precisely, this will reuse the approach presented in M3 to compare the

progression of the states using the serialization files. The comparisons will highlight how the problem is diverging and shine a light on the issues associated with the derivatives.

3.2 General Solution Proposed Approach

The method presented here uses the enablers described in the previous section. In particular, the method starts by running the inputs recorded in a hardware-in-the-loop model through a software-in-the-loop simulation that uses the open-loop driver experiment with a vehicle model. Through serialization, the vehicle states will be saved in the representative positions of each of the collocation nodes. This allows exercising high-fidelity models at each of the nodes while keeping sparsity by defaulting the majority of the states to representative values based on an existing lap. From here, the model is exercised in the optimal control problem through direct collocation to obtain the path optimization scheme. Figure 3 has a graphical representation of the steps that make up this method. The following paragraphs explain in further detail each of the steps described in Figure 3.

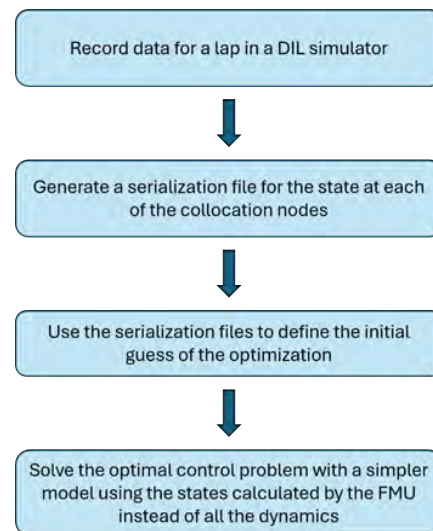


Figure 3. Implementation of Serialization of FMUs in Optimal Control Problem using Direct Collocation

Record data for a lap in a Driver-In-The-Loop (DIL) simulator:

The process to obtain the data consists of recording it using a professional DIL simulator. The data is recorded at 100Hz. The options to initialize the lap include a lap performed by a professional driver and a lap performed by the author which needless to say, is significantly slower than the lap of a professional driver but has the advantage of not bringing the car to the limit of performance, leaving more room for the optimizer to optimize, as well as for model stability in general.

Generate a serialization file for the state at each of the collocation nodes:

This process consists of using an open-loop driver experiment and broadcasting the inputs recorded in the open-loop driving experiment in the FMU in order to replicate the vehicle states in the FMU itself. In particular, throttle, gear, clutch, braking force, and steering wheel angle are used to operate the vehicle through the FMU, and these inputs are based on the recorded data from the previous step. Then, once the vehicle reaches each of the collocation nodes, its state gets serialized and saved in a file for future use. Finally, there is a validation run where the whole lap is simulated and compared with a simulation that uses the serialized states.

Use the serialization files to define the first step of the optimization:

The files obtained that describe the serialized states are used to initialize the FMU at each of the collocation nodes. This allows initializing a FMU at a representative state at each of the collocation nodes. Not only does this approach result in a very good solution to the problem of initializing the open-loop driver FMU without needing a QSS solver, but it also provides the optimizer with a good initial guess so that ideally it would converge faster. Nonetheless, this choice can preclude alternate minima, far from the solution found in the case of a given initial guess, from ever being found. As a consequence, a study on the effects of this choice is warranted, but outside of the scope of this effort to establish feasibility.

Solve the optimal control problem with a simpler model using the states calculated by the FMU instead of all the dynamics:

This part consists of letting Dymos execute the problem. In particular, it contains two main loops, one where the Jacobian is defined and one that works with the optimization directly. Each of these loops depends on updating the state on each of the nodes by using the serialized state obtained previously. These steps are solved in the reduced-order model, where the FMU is used to calculate some of the steps. It is important to note that the solution of the optimal control problem would carry the continuous propagation of the vehicle states between the serialized nodes. Figure 4 depicts how these elements interact with each other.

4 Analysis

The feasibility demonstration experiments described in Section 3 were executed to test the proposed approach. In particular, experiments M1, M2 and O1 described in Section 3.1 are crucial for this purpose. This section will show the results from these feasibility tests in a sequence that allows establishing the feasibility of the general approach presented in Section 3.2 and then present the preliminary results and discuss some of the issues related to the implementation of the general approach.

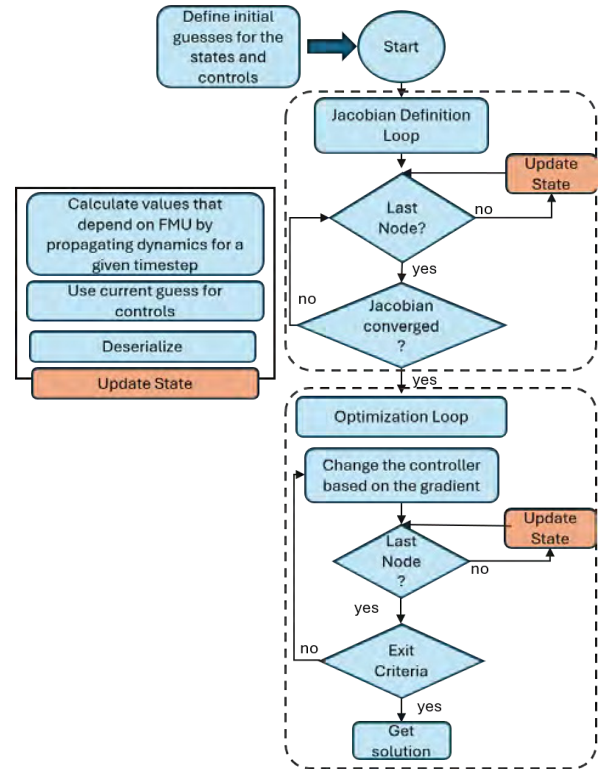


Figure 4. Dymos solution algorithm with the reduced-order model

4.1 Feasibility Results

M1: Verify that serialization is compatible with the open-loop driver formulation

The results presented in Figure 5 are recorded as the car drives around the Watkins Glen racetrack in a motionless Driver-in-the-Loop simulator (also known as GTX). They used the same vehicle model as the one represented by the FMU. The simulation using serialization of the FMU assumes a discretization of 40 equidistant nodes along the track. It can be observed that the results recorded from the GTX are slightly different from those simulated. However, there are no differences between the full simulation and the serialized simulation. This lack of differences demonstrates that serialization is a robust approach to stop simulations and resume them at later points without loss of precision in the results.

M2: Determine if the FMU remains sensitive to changes in the controls after being serialized

Once the feasibility of this approach has been established, the feasibility of changing its controls after deserializing the state needs to be assessed. Figure 6 shows the first batch of tests where the required timestep is analyzed. The results show a randomly selected node; however, the experiment was repeated at many nodes to ensure the robustness of the solution. It can be observed that changes in the steering wheel angle at the beginning of the simulation were able to induce changes in the lateral acceleration

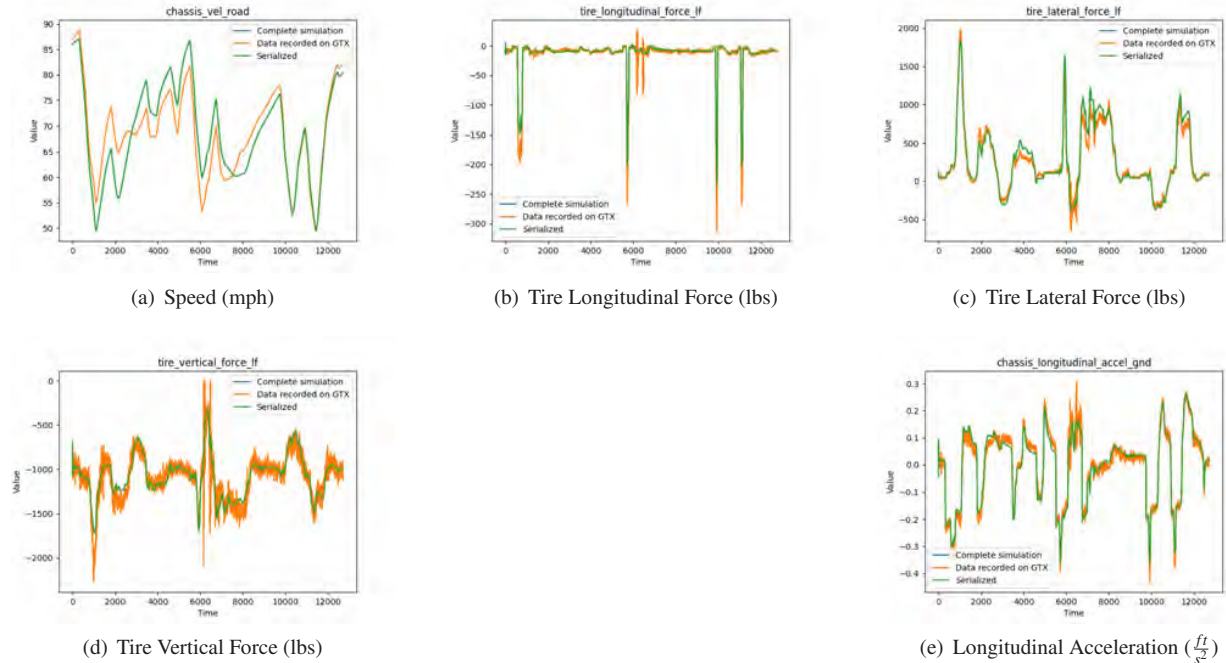
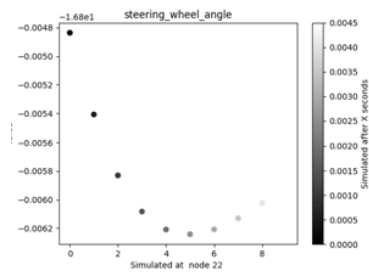
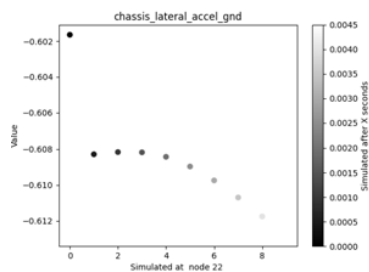


Figure 5. Comparison between data sources (Driver in the Loop recordings, simulation and serialization with FMU) across various channels for a demonstration lap

of the vehicle, confirming the propagation of changes in control inputs into the vehicle dynamics within the first 5 milliseconds. This informed the timestep that can be used for further evaluation in other steps, which was set at 6 milliseconds to allow some additional margin.



(a) Steering wheel angle (deg)



(b) Lateral Acceleration

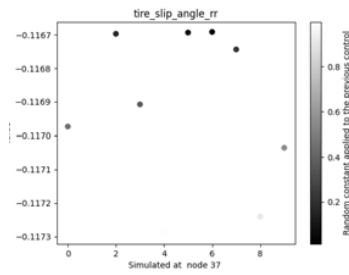
Figure 6. Timestep sensitivity results at a randomly selected node

Once a reasonable timestep was established, the experiment shifted to determining if the model remained reactive to changes in the controls. As such, multiple experiments were executed, where the values in the throttle, steering, and brake inputs were varied. This change was enforced by multiplying the changes in the controls by a random constant that ranges from 0 to 1. Figure 7 shows the results for these experiments. It can be observed that the model remains reactive to these changes in the controls, as the values presented in the figures exhibit significant variation with different values of the constant. The determination of the values for the controls depends on the optimizer for the optimal control approach. The only remaining risk from this experiment consists of the optimizer destabilizing the car and not being able to recover it afterward.

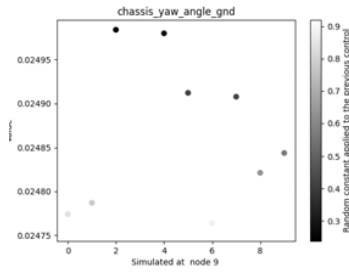
O1: Verify that the optimizer can converge with external code

A compiled version in C of a Pacejka-MF-based tire model was used to test the optimizer's compatibility with external codes. This model substituted all the tire lateral force calculations in the model. These results show that Dymos is capable of robustly integrating external calculations. More importantly, as the derivatives were not exported from the model itself, but rather calculated in Dymos, it can be said that Dymos numerical methods for derivative calculation are enough to calculate the derivatives for these problems.

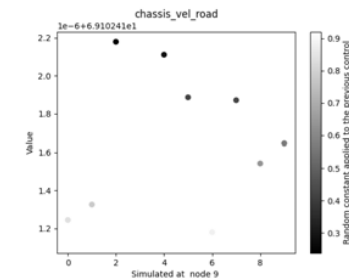
O2: Determine the node location precisely The Watkins Glen track was used to analyze the algorithm with which Dymos places the nodes to solve the collocation



(a) Tire slip angle (deg)



(b) Yaw Angle (deg)

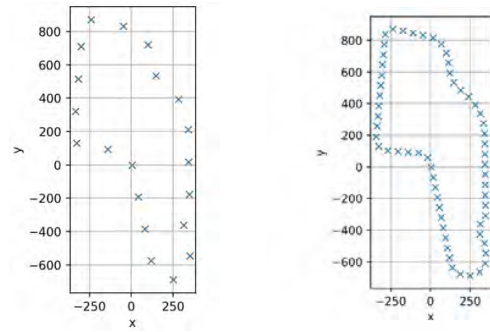


(c) Speed (mph)

Figure 7. Dynamic propagation in the model to random changes in controls after deserializing

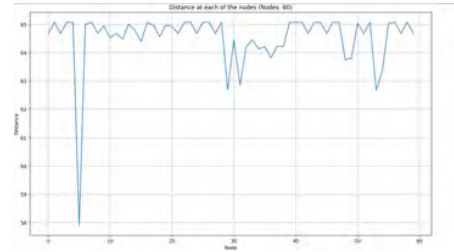
problem. This track has a mixture of corners that can shine a light on potential flawed assumptions in terms of how the distance algorithm works. Figure 8 shows the distribution of nodes around the track. It can be observed that the nodes appear to be evenly distributed in both cases. However, further investigation into the spacing algorithm is needed to validate this assumption as it can impact the approach used to serialize the FMU. Figure 8c shows the distance between each of the nodes for the 60 node separation. It can be observed that the distance between these nodes is not equal. This can be attributed to a difference in the distance calculation approaches that Dymos follows and the one used for these calculations. One potential explanation could be that the Gauss-Lobato transcription used by Dymos spaces the nodes evenly along the curvature, instead of just considering an equal Euclidean distance. Further investigation may be necessary to validate the impact of these assumptions in the serialization of the FMU.

The following experiments start implementing the gen-



(a) 20 nodes

(b) 60 nodes



(c) Distance

Figure 8. Node location definition with different number of nodes

eral approach as described in the methodology. In particular, data from a lap recorded in the Driver-in-the-Loop simulator for the Watkins Glen track was used. This data was then exercised through an open-loop driver experiment to obtain a serialization file at each of the collocation nodes. For this stage of the development of the tool, only 40 collocation nodes were used.

M3: Test the connection of the higher fidelity model with the lower fidelity model that is exposed to the optimizer

Serialization files were created by executing the problem with the initial state. The number of iterations was limited to just one to test the connection between the models. Figure 9 shows the comparison of different representative channels at each of the collocation nodes. It can be seen that, as expected, the model is not affected when exposed to the optimizer and left for just one iteration. However, it can be seen that some channels are not identical, such as the longitudinal acceleration and the derivative of the slip angle. These channels are a sample of the states related to the dynamics that were handled by the FMU and then passed into the reduced-order model. The observed differences in these underscore that the model is connected correctly and is integrated in the evaluation of the reduced-order model.

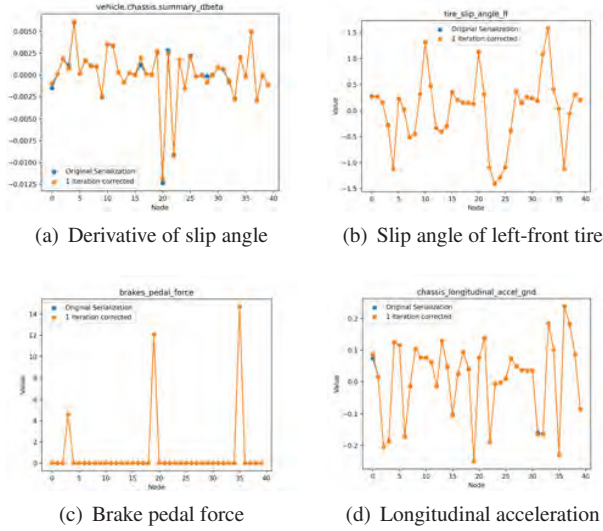


Figure 9. Vehicle states after one iteration of the optimizer based on the serialization

O3: Testing methods to obtain the derivatives The first iteration in this test significantly limited the range of values that the controls can use. This is to ensure that the rest of the states that are not passed directly from the high-fidelity model to the reduced-order model are calculated correctly and numerical instabilities are not induced. Figure 10 shows the evolution of different states through various iterations in the optimization loop using a finite differences method with automatic step size. Nonetheless, the model is still unable to converge, and the algorithm exits based on the number of iterations. Interestingly, the results suggest aggressive braking in this configuration. This suggests that there may be potential issues with the gradient that calculates the brakes. Additionally, an error related to licenses from the FMU made the optimization crash on the 48th iteration. The experiment may be repeated in other environments to validate the cause of this, but it is an issue that requires further research. Finally, while other alternative derivative approaches could be attempted, they were not implemented in this work as the exact FMU under consideration could not support complex-step or automatic differentiation methods.

5 Conclusions

Optimal control theory applied in motorsport typically uses vehicle models with a large number of simplifications and assumptions. These assumptions typically diminish the value of optimization solutions and the overall modeling effort as the uncertainty becomes too high to accept for this application. This paper presents a method to implement a high-fidelity model in an optimal control problem along with initial feasibility checks and a pilot implementation. The main objective is to identify the challenges and high technical difficulties in this kind of problem that demand these assumptions to be made.

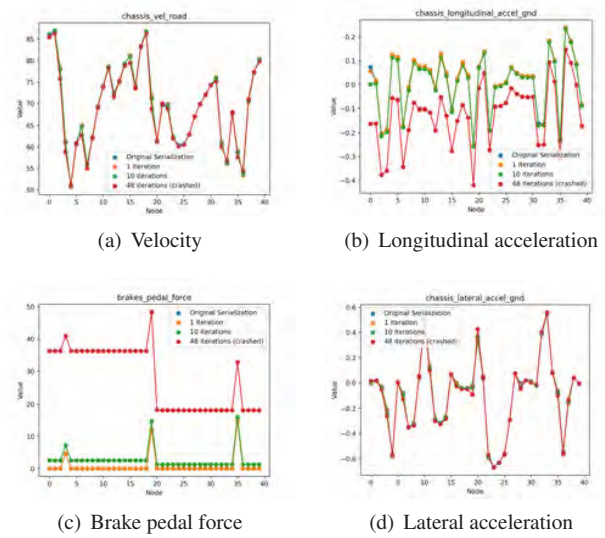


Figure 10. Vehicle state evolution as a result of the optimizer iterations

The model was exported as a FMU and interfaced with the Dymos optimal control library through the FMPy package. Different experiments were proposed to validate that the model was ready to be integrated into the problem. The first set of these experiments aimed to validate the feasibility of the proposed effort, including important enablers such as serialization of FMUs in Dymos. Future work can lie in understanding the link between the data used to initialize the simulation and the final results. Additionally, an important demonstration of the feasibility of Dymos to converge when the model is calculated with external codes was presented. These feasibility tests show that the enablers identified are compatible with this problem, and therefore, further tests on the approach can be made.

The second set of tests focuses on understanding the methods that Dymos uses. In particular, the experiments test the connection between the high-fidelity model and the OpenMDAO model that is exposed to Dymos. In conjunction with the experiments to determine the node location, we can conclude that the formulation using serialization to represent the high-fidelity model in an optimization environment has potential, although it can present integration challenges that require investigation.

The derivatives problem is the most challenging to debug as higher fidelity vehicle models can include states with non-linearities and discontinuities. The approach of restricting the controls significantly is effective, which suggests potential issues in the specific derivatives that depend on the controls. Furthermore, it can be seen that the presented path appears reasonable, although some parameters, such as the velocity, are not completely accurate, which requires further research on the solution methods. Given the success of this approach, the suggested implementation of these high-fidelity models consists of using

data from a lap driven by a professional driver, or in other words, very close to optimal, and using the optimizer to extract additional performance from the lap. However, further research on this problem might show how to solve this problem when larger differences in the control from the original state are desired.

6 Acknowledgments

The authors would like to acknowledge the support of Dr. Burak Bagdatli during the preparation of this work.

References

- Åkesson, J, W Braun, P Lindholm, and B Bachmann (2012). “Generation of Sparse Jacobians for the Function Mock-Up Interface 2.0”. In: *Proceedings of the 9th International Modelica Conference*. URL: <https://ep.liu.se/ecp/076/018/ecp12076018.pdf>.
- Becerra, Victor M (2013). “Practical Direct Collocation Methods for Computational Optimal Control”. In: *Modeling and Optimization in Space Engineering*. Ed. by Giorgio Fasano and János D Pintér. New York, NY: Springer New York, pp. 33–60. ISBN: 978-1-4614-4469-5. DOI: 10.1007/978-1-4614-4469-5_{_}2. URL: https://doi.org/10.1007/978-1-4614-4469-5_2.
- Buck, Pieter de and Joaquim R. R. A Martins (2023-08). “Minimum lap time trajectory optimisation of performance vehicles with four-wheel drive and active aerodynamic control”. In: *Vehicle System Dynamics* 61.8, pp. 2103–2119. ISSN: 0042-3114. DOI: 10.1080/00423114.2022.2101930. URL: <https://doi.org/10.1080/00423114.2022.2101930>.
- Christ, Fabian, Alexander Wischnewski, Alexander Heilmeyer, and Boris Lohmann (2021). “Time-optimal trajectory planning for a race car considering variable tyre-road friction coefficients”. In: *Vehicle System Dynamics* 59.4, pp. 588–612. ISSN: 17445159. DOI: 10.1080/00423114.2019.1704804.
- Falck, Robert, Justin Gray, Kaushik Ponnappalli, and Ted Wright (2021-03). “dymos: A Python package for optimal control of multidisciplinary systems”. In: *Journal of Open Source Software* 6.59, p. 2809. DOI: 10.21105/joss.02809.
- Falck, Robert D. and Justin S. Gray (2019). “Optimal control within the context of multidisciplinary design, analysis, and optimization”. In: *AIAA Scitech 2019 Forum*. American Institute of Aeronautics and Astronautics Inc, AIAA. ISBN: 9781624105784. DOI: 10.2514/6.2019-0976.
- Griffin, John, John Batteh, and Johan Andreasson (2012-11). “Modeling Vehicle Drivability with Modelica and the Vehicle Dynamics Library”. In: *Proceedings of the 9th International MODELICA Conference, September 3-5, 2012, Munich, Germany*. Vol. 76. Linköping University Electronic Press, pp. 599–608. DOI: 10.3384/ecp12076599.
- Güne, Atılım, Güne, Baydin, Barak A Pearlmuter, and Jeffrey Mark Siskind (2018). *Automatic Differentiation in Machine Learning: a Survey*. Tech. rep., pp. 1–43.
- Hwang, John T. and Joaquim R.R.A. Martins (2018-06). “A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives”. In: *ACM Transactions on Mathematical Software* 44.4. ISSN: 15577295. DOI: 10.1145/3182393.
- Imsland, L., P. Kittilsen, and T. S. Schei (2010). “Model-based optimizing control and estimation using Modelica models”. In: *Modeling, Identification and Control* 31.3, pp. 107–121. ISSN: 03327353. DOI: 10.4173/mic.2010.3.3.
- Kegelman, John C., Lene K. Harbott, and J. Christian Gerdes (2017-02). “Insights into vehicle trajectories at the handling limits: analysing open data from race car drivers”. In: *Vehicle System Dynamics* 55.2, pp. 191–207. ISSN: 17445159. DOI: 10.1080/00423114.2016.1249893.
- Kelly, Matthew (2017). “An introduction to trajectory optimization: How to do your own direct collocation*”. In: *SIAM Review* 59.4, pp. 849–904. ISSN: 00361445. DOI: 10.1137/16M1062569.
- Limebeer, D. J.N., M. Bastin, E. Warren, and H. G. Fensham (2023). “Optimal control of a NASCAR-specification race car”. In: *Vehicle System Dynamics* 61.5, pp. 1210–1235. ISSN: 17445159. DOI: 10.1080/00423114.2022.2067573.
- Löckel, Stefan, André Kretsch, Peter van Vliet, and Jan Peters (2022). “Identification and modelling of race driving styles”. In: *Vehicle System Dynamics* 60.8, pp. 2890–2918. ISSN: 17445159. DOI: 10.1080/00423114.2021.1930070.
- Massaro, M. and D. J.N. Limebeer (2021). “Minimum-lap-time optimisation and simulation”. In: *Vehicle System Dynamics* 59.7, pp. 1069–1113. ISSN: 17445159. DOI: 10.1080/00423114.2021.1910718.
- Modelica Association (2023-07). *Functional Mock-up Interface Specification version 2.0.4*. Tech. rep. Modelica Association. URL: <https://github.com/modelica/fmi-standard/releases/download/v2.0.4/FMI-Specification-2.0.4.pdf>.
- Perantoni, Giacomo and David J.N. Limebeer (2014-05). “Optimal control for a Formula One car with variable parameters”. In: *Vehicle System Dynamics* 52.5, pp. 653–678. ISSN: 17445159. DOI: 10.1080/00423114.2014.889315.
- Reiner, Matthias (2025-10). “Modelica FMI based hybrid reinforcement learning enhanced trajectory planning for an ADR scenario for combined control of a satellite with a 7-axis robotic arm using Modelica/FMI”. In: *Linköping Electronic Conference Proceedings*. Vol. 218. Linköping University Electronic Press.
- Sommer, Torsten (2023). *FMPy*. URL: <https://github.com/CATIA-Systems/FMPy>.
- Song, Yunlong, Angel Romero, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza (2023-09). “Reaching the limit in autonomous racing: Optimal control versus reinforcement learning”. In: *Sci. Robot.* 8.82, eadg1462.
- Su, Jingxin, Ryuji Miyazaki, Toru Tamaki, and Kazufumi Kaneda (2019-12). “High-resolution representation for mobile mapping data in curved regular grid model”. In: *Sensors (Switzerland)* 19.24. ISSN: 14248220. DOI: 10.3390/s19245373.
- Wächter, Andreas and Lorenz T. Biegler (2006-03). “On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming”. In: *Mathematical Programming* 106.1, pp. 25–57. ISSN: 1436-4646. DOI: 10.1007/s10107-004-0559-y. (Visited on 2026-09-04).
- Wischnewski, A., M. Euler, S. Gümus, and B. Lohmann (2022). “Tube model predictive control for an autonomous race car”. In: *Vehicle System Dynamics* 60.9, pp. 3151–3173. ISSN: 17445159. DOI: 10.1080/00423114.2021.1943461.
- Zarate Villazon, Angel M., Ian Brown, Michael Balchanos, and Dimitri Mavris (2026-04). “Lap Time Performance Sensitivity to Tire Model Parameters: An Optimal Control Framework for Model Selection and Calibration”. In: *SAE World Congress Experience 2026*. SAE International. DOI: 10.4271/2026-01-0214.

Terminals in SSP

Dag Brück

Dassault Systèmes AB, Lund, Sweden, dag.brueck@3ds.com

Abstract

Structured connectors (ports) are available in many simulation languages, including FMI (Functional Mockup Interface) and Modelica. However, such a concept is missing in SSP (System Structure and Parameterization) making it incompatible with its closest related standards.

This paper describes a possible extension to SSP to (1) enable the use of FMI 3 components with terminals, and (2) support structured connections between SSP subsystems.

Keywords: System Structure and Parameterization, SSP, FMI, Modelica, standardization

1 Background

System Structure and Parameterization (SSP) (Mai et al., 2024) is a tool independent standard to define complete systems. In a typical use of SSP, the system description consists of several interconnected Functional Mockup Units (FMUs), nested system descriptions, and their parameterizations and other related data, as shown in Figure 1. SSP is gaining popularity and is supported by several tools e.g. Dymola (Dassault Systèmes, 2019), easySSP (eXXcellent solutions, 2026), and FMIBench (Mai, 2025).

A bit of historical background is needed to understand the motivation and design of terminals for SSP.

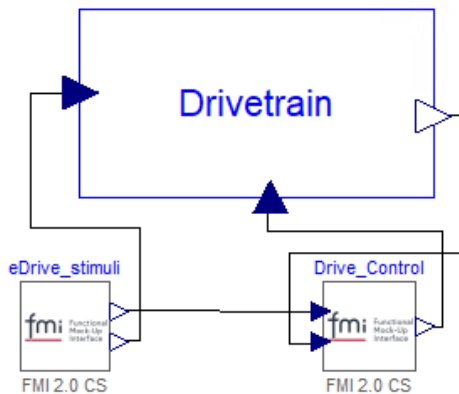


Figure 1. Simple hierarchical system model.

1.1 Why Terminals?

Terminals offer several system design advantages.

- Grouping of connectors (input, output, acausal, nested terminals) so they can be managed as one unit, for example to make a connection. Hence the maze of connection lines is reduced.
- Separation of terminal definition from its use, especially that the terminal can be changed without affecting the connection topology.
- Decluttering of the namespace, as members of the terminal are local.

In SSP there are no structured connectors (ports), although components may use such types internally.

Structured types have existed for a long time in programming languages, such as COBOL (CODASYL, 1960). The concept was missing in languages embraced by the simulation community, notably FORTRAN (Backus, et al., 1956) and APL (Iverson, 1962).

Terminals are a very active topic in the SSP design group. While the design is still being hashed out, it is likely that some form will materialize in a future SSP specification. The key motivation for this paper is to solicit input on various design options.

1.2 SSP History

In one sense, SSP can be seen as the brainchild of FMI (Bertsch & Sommer, 2024), adding (mainly) connection structure between multiple components and parameter sets, which were regarded as beyond the scope of FMI. However, SSP is expressed in more general terms which did not force it into an FMI-only framework. Because early versions of FMI had neither arrays nor structured ports, so was SSP designed without such features.

FMI version 3 has evolved and supports both arrays and a particular form of structured connectors that were backported to FMI 2.0.4. FMI terminals allow grouping of connectors, but unusually (compared to programming languages) requires the terminal members to be defined as connectors in parallel to the terminal. The motivation for

such a design is backward compatibility – it provides a terminal concept that can be ignored.¹

SSP version 2 tried to catch up with FMI 3 with support for additional basic types and arrays, for example. SSP 2.0 is also clearer that it can support non-FMU components if the tool supports it, and Modelica (Olsson, 2026) components are explicitly supported. The terminal concept is more complicated, and it was not clear that reusing the FMI approach would be appropriate. For that reason, terminals have been scheduled for a later release.

2 Outline of the Requirements Space

Terminals in SSP face a number of requirements, some of which seem contradictory.

2.1 FMI 3 Terminals in SSP

Obviously FMUs with terminals must be supported by SSP. SSP offers redundant interface declarations; even though the FMU defines the exact interface, the SSP component definitions repeats the interface. This allows faithful graphical and semantic knowledge of the component without the need to process the content of the FMU. Consequently, SSP needs a syntax to represent FMI terminals.

FMI's design choice to make terminals an ignorable feature should if possible be preserved in SSP for compatibility reasons.

2.2 System Terminals in SSP

An SSP model may comprise not only FMU components but also multiple levels of subsystems defined within the SSP framework as System Structure Descriptions. The need for terminals that arises for FMUs is also present for SSP systems, either to connect FMU terminals to the system interface or to interconnect several SSP subsystems.

When other modeling languages, such as Modelica, are mapped to SSP there is a need for a more capable terminal concept in SSP compared to FMI. One such requirement is the support of acausal connectors. However, attributes such as *final*, *inner*, *outer*, *replaceable*, *flow* and *stream* connectors can only be represented by annotations in SSP. Flow variables are part of FMI 3.0 though.

The particular form of terminals with externally defined connectors can possibly be motivated by FMI compatibility, but that need is not present in native SSP. Hence, SSP opens the possibility of a better encapsulated terminal concept and common definitions of structured connector types.

2.3 Empty Connectors in SSP

SSP is used not only to describe complete, ready-to-simulate models, it has also gained traction as a lightweight description of system architecture. In that context it becomes natural to define connectors without any contents, to be refined in a later design phase.

This particular use is close to the use of (empty) bus connectors in Modelica.

2.4 Parameter Records

Influenced by FMI, parameters are defined as a particular form of connector in SSP. The somewhat far-fetched logic being that parameters are part of the interface.

Parameters are often grouped in a record structure. The terminal concept should support that too, which however may have an impact on the parameter set data format.

SSP terminals correspond to SysML record types, within the semantic limitations of SSP connectors.

2.5 Summary of Requirements on Terminals

SSP may require a terminal concept that comes in two flavors: one to provide FMI 3 compatibility, and another more general end better encapsulated for SSP systems.

A concrete list of requirements include:

- Support for FMI 3 terminals with externally defined connectors.
- Support for encapsulated terminals where connectors are defined within the terminal.
- Possibility to define common terminal types, which can be used to express common interfaces for many components and systems.
- Optionally reuse the terminal concept to define parameter records (as parameters are a particular kind of connector in SSP).
- Optionally to define parameter sets (SSV) with values of record type.

The rest of the paper details a possible terminal design.

3 A Concrete Design Proposal

The following two kinds of terminal definitions are proposed, each addressing one key design requirement. The next section will discuss a potential unification.

The use of XML namespaces is not fully consistent in SSP. This paper mainly uses the SSD namespace (part of the description), but the SSC namespace (common part) might be a better choice.

¹ There is precedence for “feature you can ignore”, e.g. the boolean datatype in C++ (Brück & Koenig, 1993). But it

is still a questionable sacrifice to meet requests for backward compatibility.

3.1 Terminal as a Grouping of FMI Connectors

Meeting the need to map FMI terminals to SSP, a similar terminal concept is introduced.

- It names a terminal (port).
- It enumerates which connectors are considered part of the terminal.
- The connector definitions are separate, but must be at the same hierarchical level.
- Terminals may be connected as if they were connectors.

Here is an example of such a definition where parts not essential for the exposition have been omitted:

```
<ssd:Component name="Source_fmu">
  <ssd:Connectors>
    <ssd:Connector kind="output"
      name="tSource_pos">
      <ssc:Real/>
    </ssd:Connector>
    <ssd:Connector kind="output"
      name="tSource_neg">
      <ssc:Real/>
    </ssd:Connector>
    <ssd:Connector kind="input"
      name="tSource_feedback">
      <ssc:Real/>
    </ssd:Connector>
  </ssd:Connectors>

  <ssd:Terminals>
    <ssd:Terminal name="tSource">
      <ssd:TerminalMemberVariable
        memberName="pos"
        variableName="tSource_pos"/>
      <ssd:TerminalMemberVariable
        memberName="neg"
        variableName="tSource_neg"/>
      <ssd:TerminalMemberVariable
        memberName="feedback"
        variableName="tSource_feedback"/>
    </ssd:Terminal>
  </ssd:Terminals>

  <ssd:ElementGeometry ... />
</ssd:Component>
```

What we see here is a component definition with three traditional SSP connectors (two outputs, one input), followed by a terminal definition `tSource` that groups the connectors.

One can argue that the terminal definition could either define a port (as shown here) or just a terminal type that must be instantiated. Because such terminal definitions refer to local connectors, hence cannot be shared between components, the first was chosen.

Terminals in FMI have additional attributes such as `matchingRule` and `variableKind` that also need to be defined in an SSP context.

3.2 Terminal Definitions for Native SSP

Modelica and most programming languages use an approach where the terminal's content is directly defined, avoiding the redundancy of individual connectors with grouping tacked on. A consequence is of course that the terminal is always used for connections.

Such an approach naturally lends itself to a separation of terminal definitions and connectors, where the terminal definition becomes a type in the conventional sense. Hence a single terminal definition can be used by many components, making intended connection compatibility easy to establish. The (terminal) connector can be defined as any other connector, with an appropriate type reference.

This is an example of a component definition with one (terminal) connector.

```
<ssd:Component name="Source_component">
  <ssd:Connectors>
    <ssd:Connector kind="undefined"
      name="tSource">
      <ssd:Terminal name="tSourceType" />
    </ssd:Connector>
  </ssd:Connectors>
  <ssd:ElementGeometry .. />
</ssd:Component>
```

The designation `kind="undefined"` is SSP's way of defining acausal connectors. The separate terminal type definition section would be:

```
<ssd:Terminals>
  <ssd:TerminalDefinition
    name="tSourceType">
    <ssd:Connector kind="output"
      name="pos">
      <ssc:Real/>
    </ssd:Connector>
    <ssd:Connector kind="output"
      name="neg">
      <ssc:Real/>
    </ssd:Connector>
    <ssd:Connector kind="input"
      name="feedback">
      <ssc:Real/>
    </ssd:Connector>
  </ssd:TerminalDefinition>
</ssd:Terminals>
```

It should be noted that SSP already has a very similar type definition concept for enumerations.

This approach offers several advantages.

- There is a clear separation between terminal type definitions and their usage.

- Terminal definitions can be shared by several components and sub-systems, hence aiding connection compatibility. It also lends itself to defining interfaces when SSP is used as a system architecture language.
- There are no redundant definitions, and in particular the risk of connecting twice (directly to connector and to terminal) is avoided.
- The terminal's use as a connector is clear, because the terminal is just a particular connector type (like enumerations).

A terminal concept like this is conventional, even if it disagrees with FMI. A more complete example is shown in the Appendix.

Sub-typing relationships are very limited in SSP. First, they cannot be explicitly constructed by e.g. inheritance. However, any tool must check and apply structural equivalence to implement connections. A special case could be made for bus connections, where the connection topology defines the bus contents; SSP does not (yet) support buses as a fundamental construct.

3.3 Connections to Terminals

There is no need to syntactically distinguish connections between terminals from connections to current SSP connectors. Proper interpretation of connections in any case requires an analysis by the tool of what is connected. To human readers a connection is a connection.

However, we need a construct to name individual members of a terminal when connecting to scalar non-terminals. Many languages use dot-notation `t.m`, but that would impose a restriction on the user's namespace. A structured XML element is probably a better choice.

4 Is a Unified Design Possible?

The short answer is no and yes.

The key property that makes these two approaches hard to unify is the “feature you can ignore” aspect of FMI terminals. From that follows both the semantics and structure of terminals. For example, it makes it hard to define common terminal types shared by many components.

However, imagine it would be possible to forego that flexibility in SSP; if you use terminals in your FMU you must use them when connecting in SSP. That would make it possible to map FMI terminals to “native” SSP terminals, with the resulting benefits in data abstraction.

The native SSP terminal definition can be default-mapped to the corresponding FMI terminal. There is however a shift in the relation between a terminal and the constituent individual connectors.

- In FMI, the connectors are defined individually, and the terminal groups them.
- In SSP, the terminal defines the connectors, but they can be used to define individual connectors with the same name if needed.

In the case where the flexible relationship between terminal member names and individual connectors is needed, we can provide a shortened mapping definition. Assuming the definition of `tSourceType` as above, we can define the component interface as:

```
<ssd:Component name="Source_fmu">
  <ssd:Connectors>
    <ssd:Connector kind="undefined"
      name="tSource">
      <ssd:Terminal name="tSourceType">
        <ssd:TerminalMapping
          memberName="pos"
          variableName="tSource_pos"/>
        <ssd:TerminalMapping
          memberName="neg"
          variableName="tSource_neg"/>
        <ssd:TerminalMapping
          memberName="feedback"
          variableName="tSource_feedback"/>
      </ssd:Terminal>
    </ssd:Connector>

    <!-- Implied conn. from mappings:
      tSource_pos, tSource_neg and
      tSource_feedback -->

    <ssd:ElementGeometry .. />
  </ssd:Component>
```

In many cases an automatic mapping to FMI terminals (without renaming) would be sufficient.

The alternative would be to go in the opposite direction, to map all SSP terminals to the FMI concept. That would of course keep SSP closer to its FMI roots, but is undesirable due to the more convoluted semantics and loss of data abstraction.

5 Summary

The need for terminals (structured connectors, or ports) in SSP was identified. Two partially contradicting needs have been identified: FMI compatibility and good design with less need for absolute backward compatibility. A unification is possible if we stretch the SSP semantics to cope with FMI terminals.

Several future design decisions lie in the hands of the SSP group, but the design ideas are converging.

Acknowledgements

The author wishes to thank the members of the SSP design group and Dymola colleagues for constructive discussions on various terminal concepts. Much valuable feedback on SSP usage has been provided by the ProSTEP ivip Smart System Engineering group.

References

- Backus, J. W., Beeber, R. J., Best, S. F., Goldberg, R., Herrick, H. L., Hughes, R. A., . . . Ziller, I. (1956). *The FORTRAN Automatic Coding System for the IBM 704 EDPM: Programmer's Reference Manual*.
- Bertsch, C., & Sommer, T. (2024). *Functional Mock-up Interface Specification*.
- Brück, D., & Koenig, A. (1993, September 27). *Extension Proposal: Boolean data type*. Retrieved from ISO WG21/N0350 ANSI X3J16/93-0143: <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/1993/N0350.pdf>
- CODASYL. (1960). *Initial Specifications for a Common Business Oriented Language (COBOL)*.
- Dassault Systèmes. (2019). *What is Dymola?* Retrieved from <https://www.3ds.com/fileadmin/PRODUCTS/CATI A/DYMOLA/PDF/What-is-Dymola-2020x.pdf>
- eXXcellent solutions. (2026). *orchideo | easySSP*. Retrieved from <https://www.easy-ssp.com/>
- Mai et al., P. R. (2019, March 5). *System Structure and Parameterization*. Retrieved from <https://ssp-standard.org/publications/SSP10/SystemStructureAndParameterization10.pdf>
- Mai et al., P. R. (2024, December 20). *System Structure and Parameterization*. Retrieved from <https://ssp-standard.org/docs/2.0/>
- Mai, P. R. (2025). *PMSF FMI Bench*. Retrieved from <https://pmsf.eu/products/fmibench/>
- Olsson, H. (2026, May 22). *Modelica Language Specification, Version 3.7*. Retrieved from <https://specification.modelica.org/master/>

Appendix

The following example shows the use of terminals in a more complete example comprising two components.

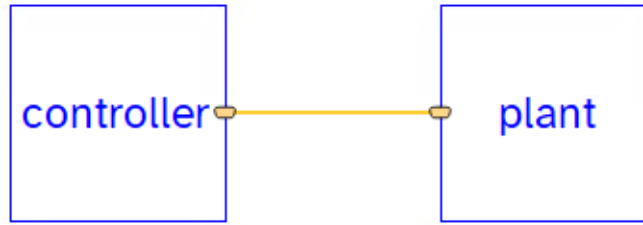


Figure 2 System with terminal connection.

The Modelica code for the model is:

```

model System1 "Example system"
  ControllerModel controller
    annotation (Placement(transformation(extent={{-60,-20},{-20,20}})));
  RobotModel plant
    annotation (Placement(transformation(extent={{20,-20},{60,20}})));
equation
  connect(controller.axisControlBus, plant.axisControlBus) annotation (Line(
    points={{-20,0},{20,0}}, color={255,204,51}, thickness=0.5));
end System1;

model ControllerModel "Simplified controller"
  AxisControlBus axisControlBus
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
end ControllerModel;

model RobotModel "Simplified robot model"
  AxisControlBus axisControlBus
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
end RobotModel;

expandable connector AxisControlBus "Data bus for one robot axis"
  extends Modelica.Icons.SignalSubBus;
  Real angle_ref(unit="rad") "Reference angle of axis flange";
  Real angle(unit="rad") "Angle of axis flange";
  Real speed_ref(unit="rad/s") "Reference speed of axis flange";
  Real speed(unit="rad/s") "Speed of axis flange";
  Real acceleration_ref(unit="rad/s2") "Reference acceleration of axis flange";
  Real acceleration(unit="rad/s2") "Acceleration of axis flange";
end AxisControlBus;

```

The model is in the architecture phase, so the causality of connectors is unspecified in SSP. The exported System Structure definition (with some simplification for clarity) is:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ssd:SystemStructureDescription generationTool="Dymola by Dassault Systemes"
  name="BusExample" version="2.1-alpha">

  <ssd:System description="Example system" name="System1">
    <ssd:Elements>
      <ssd:Component name="controller" source="ControllerModel"
        type="application/x-ssp-definition">
        <ssd:Connectors>
          <ssd:Connector kind="unspecified" name="axisControlBus">
            <ssc:Terminal name="AxisControlBus"/>
            <ssd:ConnectorGeometry x="1" y="0.5"/>
          </ssd:Connector>
        </ssd:Connectors>
        <ssd:ElementGeometry x1="-60" x2="-20" y1="-20" y2="20"/>
      </ssd:Component>

```

```

<ssd:Component name="plant" source="RobotModel"
  type="application/x-ssp-definition">
  <ssd:Connectors>
    <ssd:Connector kind="unspecified" name="axisControlBus">
      <ssc:Terminal name="AxisControlBus"/>
      <ssd:ConnectorGeometry x="0" y="0.5"/>
    </ssd:Connector>
  </ssd:Connectors>
  <ssd:ElementGeometry x1="20" x2="60" y1="-20" y2="20"/>
</ssd:Component>
</ssd:Elements>

<ssd:Connections>
  <ssd:Connection endConnector="axisControlBus" endElement="plant"
    startConnector="axisControlBus" startElement="controller">
    <ssd:ConnectionGeometry pointsX="-20 20" pointsY="0 0"/>
  </ssd:Connection>
</ssd:Connections>

  <ssd:SystemGeometry x1="-100" x2="100" y1="-100" y2="100"/>
</ssd:System>

<ssd:Terminals>
  <ssd:TerminalDefinition description="Data bus for one axis" name="AxisControlBus">
    <ssd:Connector description="Reference angle" kind="unspecified" name="angle_ref">
      <ssc:Real unit="rad"/>
    </ssd:Connector>
    <ssd:Connector description="Angle of axis flange" kind="unspecified" name="angle">
      <ssc:Real unit="rad"/>
    </ssd:Connector>
    <ssd:Connector description="Reference speed" kind="unspecified" name="speed_ref">
      <ssc:Real unit="rad/s"/>
    </ssd:Connector>
    <ssd:Connector description="Speed of axis flange" kind="unspecified" name="speed">
      <ssc:Real unit="rad/s"/>
    </ssd:Connector>
    <ssd:Connector description="Ref accel" kind="unspecified" name="acceleration_ref">
      <ssc:Real unit="rad/s2"/>
    </ssd:Connector>
    <ssd:Connector description="Acceleration" kind="unspecified" name="acceleration">
      <ssc:Real unit="rad/s2"/>
    </ssd:Connector>
  </ssd:TerminalDefinition>
</ssd:Terminals>
  <ssd:DefaultExperiment startTime="0" stopTime="1"/>
</ssd:SystemStructureDescription>

```

The Modelica Buildings Requirements Library for In-Simulation Requirements Verification for HVAC Applications

David Blum¹ Ettore Zanetti¹ Michael Wetter¹

¹Building and Industrial Energy Systems Division, Lawrence Berkeley National Laboratory, USA,
{dhblum, ezanetti, mwetter}@lbl.gov

Abstract

Building and district energy systems and their controls are facing new performance objectives with stricter reliability criteria, such as providing grid flexibility, critical cooling to data centers, multi-sector coupling of district networks, and lower energy costs for building owners. Verifying that operational requirements are met throughout the system design process using dynamic simulations leads to lower commissioning costs, higher up-time, less failure risk, and provides a basis for system specification and contractual agreements. However, manually verifying operational requirements from simulation results is a lengthy, tedious, and repetitive task for modelers, especially for large and complex systems. The open-source Modelica_Requirements Library was previously developed for defining requirements within a Modelica model and verifying them during a simulation. This paper presents the open-source Modelica Buildings_Requirements Library, which extends the Modelica_Requirements Library with template blocks representing common requirements for building and district energy and HVAC system applications. The paper describes the library's structure and component models and demonstrates its use to verify the operational performance of a novel commercial building HVAC central plant.

Keywords: buildings, HVAC, controls

1 Introduction

1.1 Motivation

Energy systems for buildings, districts, and data centers are facing new performance objectives with stricter reliability criteria while also needing to maintain affordability. Examples include enabling grid flexibility (DOE 2021), providing reliable and efficient heating and cooling to data centers (Shehabi et al. 2024) and district networks (Sommer et al. 2020), and integrating storage and heat recovery in novel HVAC architectures (Gill 2021). The result is growing complexity in system designs and their controls, leading to system integration challenges in design, deployment, and operation. Such challenges are being addressed by the development of a Modelica-based ecosystem of dynamic simulation models capable of representing novel building and district energy system architectures

and explicit control logic (Wetter, Zuo, et al. 2014; Wetter, Cimmino, et al. 2025), a model- and verification-based design process for energy systems known as Platform-Based Design (PBD) (Sulzer et al. 2023; Wetter, Sulzer, et al. 2026), a digitalized delivery process for controls (Wetter, Ehrlich, et al. 2022), and AI-assisted modeling workflows (Batteh 2025; Li 2026). Key to each workflow will be the ability to systematically specify and formally verify that operational requirements are being met throughout the design, deployment, and operation processes, which can include many different design iterations and virtual testing scenarios.

Requirements specification and verification throughout a system's life-cycle is common in industries such as automotive, aerospace, and power plants (Otter et al. 2015), and the building and district energy industries would benefit from adopting these practices. Of particular interest are operational requirements imposed on dynamic system operation and control, for example as defined within the model-based Design, Verification, and Risk Reduction workflow of the PBD process used in a recent district energy system application (Wetter, Dee, et al. 2026). Example operational requirements from that platform are:

- O-201: At system-wide level, the heat pumps must operate at least 30 min when activated, and remain off for at least 10 minutes.
- O-202: At system-wide level, all control valves must show stable operation.
- O-303: At energy transfer station level: The space heating water supply temperature set point must be tracked within 1K once the system is in space heating mode for 10 minutes.
- O-503: At central plant level: The mixing water temperature in the district loop after the central plant must be between 10.5°C and 24.0°C.

Verifying these types of requirements using dynamic simulations leads to lower commissioning costs, higher up-time, and a greater chance of avoiding system failures that could cause physical damage, significant delays, and other penalties. Furthermore, it provides a basis for system specification and contractual agreements that can be

carried through each step of a system's design, deployment, commissioning, and operation processes.

Despite the benefits of model-based requirements verification, for large and complex models, tedious manual verification of simulation results can take many hours, and includes the repetitive work of inspecting trajectories of set points versus process variables, cycling rates of equipment, and stability of feedback control loops to name a few. Automating the process would allow the modeler to more quickly identify their source and address failures appropriately.

The Modelica_Requirements Library (Otter et al. 2015) was previously developed to provide basic functionality to define requirements within a Modelica model and verify them during a simulation. The library makes use of 3-valued logic so that requirements can be satisfied, violated, or undecided, with a key advantage over 2-valued logic (strictly satisfied or violated) being that it allows for requirements to be untested during periods of equipment inactivity or other conditions where a determination should not be made. The Modelica_Requirements Library is useful to provide fundamental Modelica types, functions, and component blocks for specifying and verifying requirements. However, given the significant need and growth in model-based workflows for building and district energy and HVAC systems, further development would facilitate its usage specifically within these applications.

1.2 Objectives

This paper presents the Modelica Buildings_Requirements Library, which extends the Modelica_Requirements Library for building and district energy and HVAC system applications and is available and being developed open-source at <https://github.com/lbl-srg/modelica-buildings-requirements>. The objectives of this paper are to present the basic structure and components of the new library, in Section 2, and demonstrate its use in an example application, in Section 3. Further discussion of the example usage is given in Section 4, while conclusions and future work are given in Section 5.

1.3 Novelty

The novelty of the Modelica Buildings_Requirements Library is its extension of the Modelica_Requirements Library to create template Modelica blocks that implement common operational requirements in a consistent manner, and combine requirements often grouped together or with other functionality as-needed. For example, introducing a delay time for activating requirements, as seen in O-303 above, determining the stability of a signal in a computationally efficient manner, and consistently exposing parameters and reporting violations.

2 Library Description

2.1 Structure and Base Classes

The Modelica Buildings_Requirements Library has the structure shown in Figure 1, with four component models that are each validated by a model in the Validation package and two base class models in the BaseClasses package.

In the BaseClasses package, the partial model `BaseClasses.PartialRequirement` is a base class for all component models in the library that instantiates `Modelica_Requirements.Verify.Requirement` and `Modelica_Requirements.Interfaces.ObservationID` to consistently allow for specifying and reporting the name and natural language of a requirement in an HTML report generated at the completion of a simulation, propagated for convenient access to users through parameters `name` and `text` respectively. The model `PartialSignalTestWhenActive` extends `PartialRequirement` to add capability to activate (with an optional time delay) or deactivate the requirement with a boolean input signal. When deactivated, the requirement block outputs `undecided`.

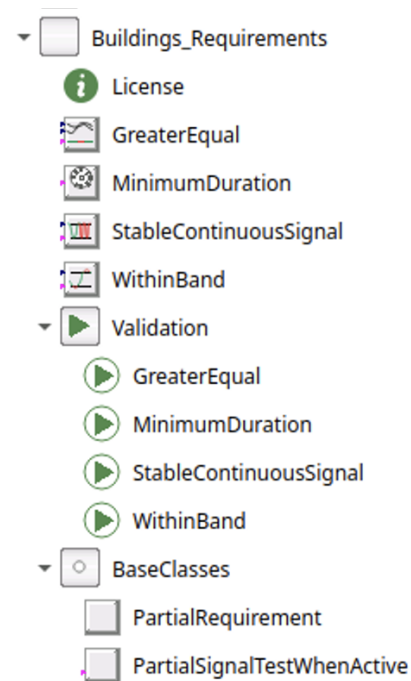


Figure 1. Modelica Buildings Requirements library.

2.2 Components and Illustrative Example

The component model `GreaterEqual` verifies whether an input signal is greater or equal to a second input signal when the test is active, with optional delay time after activation and with optional minimum time of violation to be considered violated, primarily using the threshold block `Buildings.Controls.OBC.CDL.Reals.Less` from the Modelica Buildings Library (Wetter, Zuo, et al. 2014).

The component model `MinimumDuration` verifies whether a signal is true for a minimum time, primarily extending `BaseClasses.PartialRequirement` with use of `Modelica_Requirements.ChecksInFixedWindow.MinDuration`.

The component model `StableContinuousSignal` verifies whether an input signal is stable when the test is active, with optional delay time after activation and with optional minimum time of violation to be considered violated, checking that the signal does not have a high amount of oscillations. It does so by comparing a moving average of the rate of change of a signal within a sliding window to a threshold. The implementation avoids state events that would be introduced with an alternative formulation that detects the change in signal direction instead.

Finally, the component model `WithinBand` verifies whether an input signal is within constant lower and upper bounds when the test is active, with optional delay time after activation and with optional minimum time of violation to be considered violated, primarily extending use of `Modelica_Requirements.LogicalBlocks.WithinBand`.

An example validation model using four instances of `GreaterEqual` is shown in Figure 2. The top three instances are configured the same in terms of verifying the output from the sine source is greater than the output from the constant source, except that the first is always active, the second is always inactive, and the third is sometimes active. The fourth instance is configured so that the requirement is always satisfied, as the constant signal is always equal to itself, but not always active.

Results for simulating the example model for one second are shown in Figure 3. When the sine signal is greater than the constant signal, the requirement is nominally satisfied except for the status of activation. For the top instance of `GreaterEqual`, notice the requirement is reported as satisfied for $t \leq 0.5$, and violated afterwards. Meanwhile, the second instance is reported as undecided always, the third instance oscillates between satisfied ($t \leq 0.5$) or violated ($t > 0.5$) and undecided depending on the value of the activation pulse, and the fourth instance oscillates between satisfied and undecided depending on the activation pulse.

In addition, through functionality provided by the `Modelica_Requirements` Library, a verification report is saved at the end of the simulation in HTML. The report generated for the example model is shown in Figure 4 and gives more details of each requirement, grouping by those violated at some point in time ("Requirements violated"), untested for all time ("Requirements untested"), or satisfied ("Requirements satisfied"). For the requirements that are violated, the time of violation is given. For all the requirement details, the value of the `name` parameter is given in the Observation column, the instance path of the requirement block is given in the Requirement column, and the value of the `text` parameter is given in the Description column.

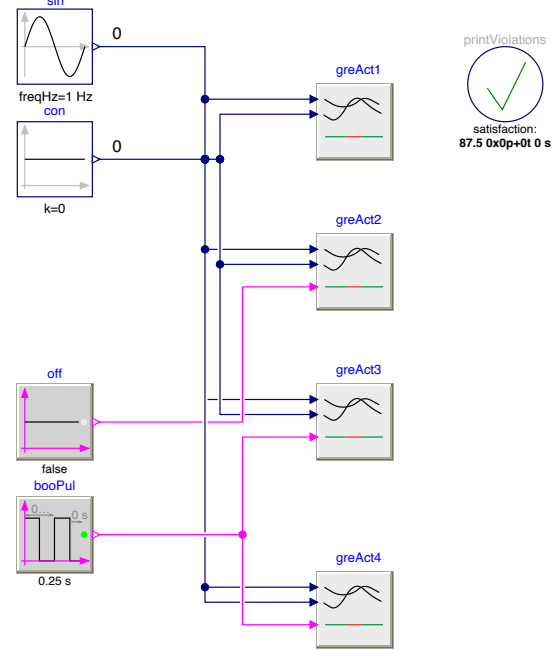


Figure 2. Example model using four instances of `GreaterEqual`.

3 Example Usage: Commercial Building Central Plant

3.1 Context

This section will demonstrate the use of the `Modelica Buildings_Requirements` Library within a project that has the aim of simulating a novel commercial building central plant design known as the Time-Independent Energy Recovery (TIER) plant (Gill 2021), as shown conceptually in Figure 5 and in more detail in Figure 6 as implemented in the `Modelica Buildings Library v12.0.0` as `Buildings.DHC.Plants.Combined.AllElectricCWStorage`. TIER integrates heat recovery chillers, cooling-only chillers, a thermal energy storage tank, cooling towers, and a "trim" heat source, such as air source heat pumps. The integration of heat recovery chillers and storage in this plant design has the potential to reduce energy costs for building owners and help maintain a reliable grid by reusing waste heat and by using storage to shift heating and cooling equipment loads. However, the complexity of this system makes it difficult to control and warrants the use of dynamic simulation to verify reliable control and successful performance across various operating scenarios. Details and results for such simulation studies and scenario sensitivities are presented in Zanetti et al. (2026). In this paper, we demonstrate the use

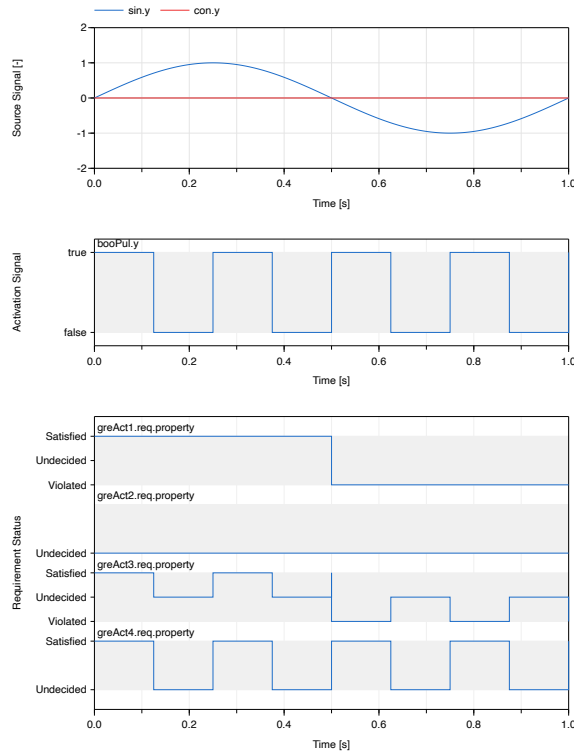


Figure 3. Simulation results for one second of simulation time for example model using `GreaterEqual` from Figure 2, showing sine (blue) and constant (red) signals (top), activation signal pulse (middle), and the requirement status of the four `GreaterEqual` instances (bottom).

Requirements violated (2 of 4)

Observation	Requirement	First violation at	Description
AlwaysActive	greAct1.req	0.5 s	Sine signal is greater or equal to constant signal
SometimesActive	greAct2.req	0.5 s	Sine signal is greater or equal to constant signal subject to activation

Requirements untested (1 of 4)

Observation	Requirement	Description
NeverActive	greAct1.req	Sine signal is greater or equal to constant signal subject to activation

Requirements satisfied (1 of 4)

Observation	Requirement	Description
AlwaysSatisfied	greAct3.req	Sine signal is greater or equal to constant signal subject to activation

Figure 4. Requirements HTML report generated for example model using `GreaterEqual` from Figure 2.

of the Modelica Building_Requirements Library to aid in verifying control performance in the simulations.

The TIER plant is serving a large office building in San Francisco, CA, with a configuration that has two heat recovery chillers, three cooling-only chillers, three cooling towers, two air source heat pumps, and a storage tank sized for three hours. For the simulations, the TIER plant model is `Buildings.DHC.Plants.Combined.AllElectricCWStorage` from the Modelica Buildings Library v12.0.0. Changes between the model in the library and that used in this case study include equipment performance maps for the heat recovery chillers and cooling only chillers being adapted to work at higher temperature

lifts, supplied by a mechanical designer for this study, and the supply heating set point of 51.7°C. The simulation is done by first generating heating and cooling load profiles for the entire year by simulating the Department of Energy prototype ASHRAE 90.1 2022 building for a Large Office (46,000 m², 500,000 ft²) in EnergyPlus. Then, the load profiles are read into the Modelica model and the energy is added or removed from the chilled water and heating water distribution loops, respectively, through heat exchangers. The same weather file based on TMY3 data is used in the Modelica model as in the EnergyPlus model for ambient boundary conditions for equipment like cooling towers and air-source heat pumps. Dymola 2025x with solver Dassl and tolerance 1e-6 was used for the simulation.

3.2 Requirements Usage

The component models presented in the previous section can be built into requirements templates for HVAC equipment and reused throughout a system model. We have prototyped such templates for three types of equipment: heating and cooling production, thermal energy storage, and pumps. Figure 7 shows the Modelica model template for heating and cooling production as an example.

Heating and cooling production equipment are responsible for adding or removing energy from a fluid, such as chillers, boilers, heat pumps, and cooling towers. A template for this type of equipment takes the measured supply temperature leaving the equipment, the measured return temperature entering the equipment, and the equipment's supply temperature set point as inputs. Multiple component blocks are instantiated from the Modelica Buildings_Requirements Library to verify that the:

- Supply temperature is maintained within a tolerance around the set point, using `WithinBand`.
- Supply and return temperatures stay within minimum and maximum bounds, using `WithinBand`.
- Supply temperature remains stable, using `StableContinuousControl`.
- Supply and return temperature difference stays within minimum and maximum bounds, using `WithinBand`.
- Minimum equipment on and off times are respected, using two instances of `MinimumDuration`.

Thermal energy storage is responsible for maintaining a supply temperature while being discharged, subject to timely charging and losses, such as water tanks, ice tanks, or phase change materials. A template for this type of equipment takes the upper and lower storage temperature set point bounds and measured temperatures at different layers within the storage as inputs. Multiple component blocks are instantiated from the Modelica Buildings_Requirements Library to verify that the:

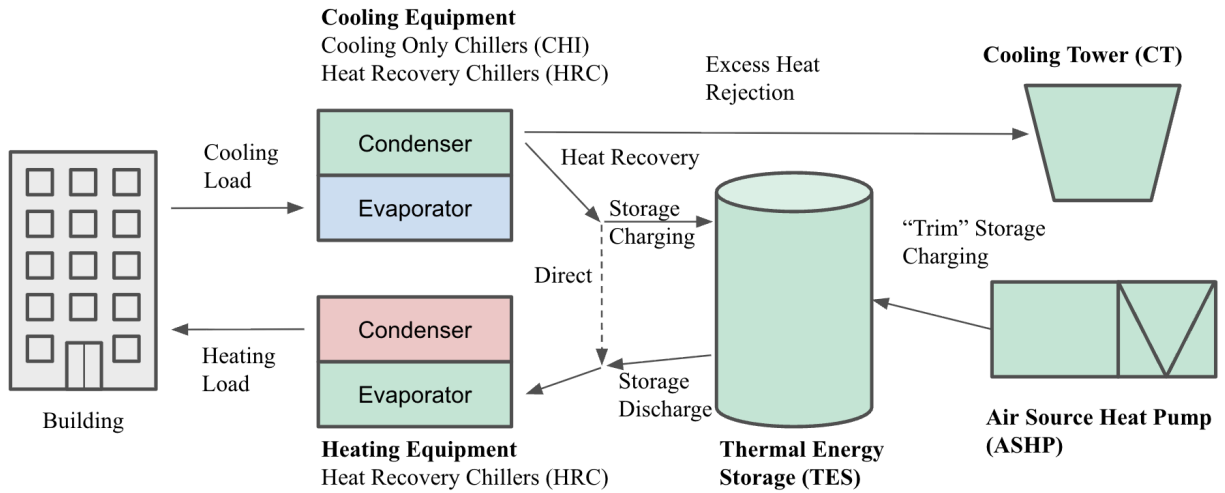


Figure 5. Conceptual diagram of an HVAC central plant known as TIER (Gill 2021).

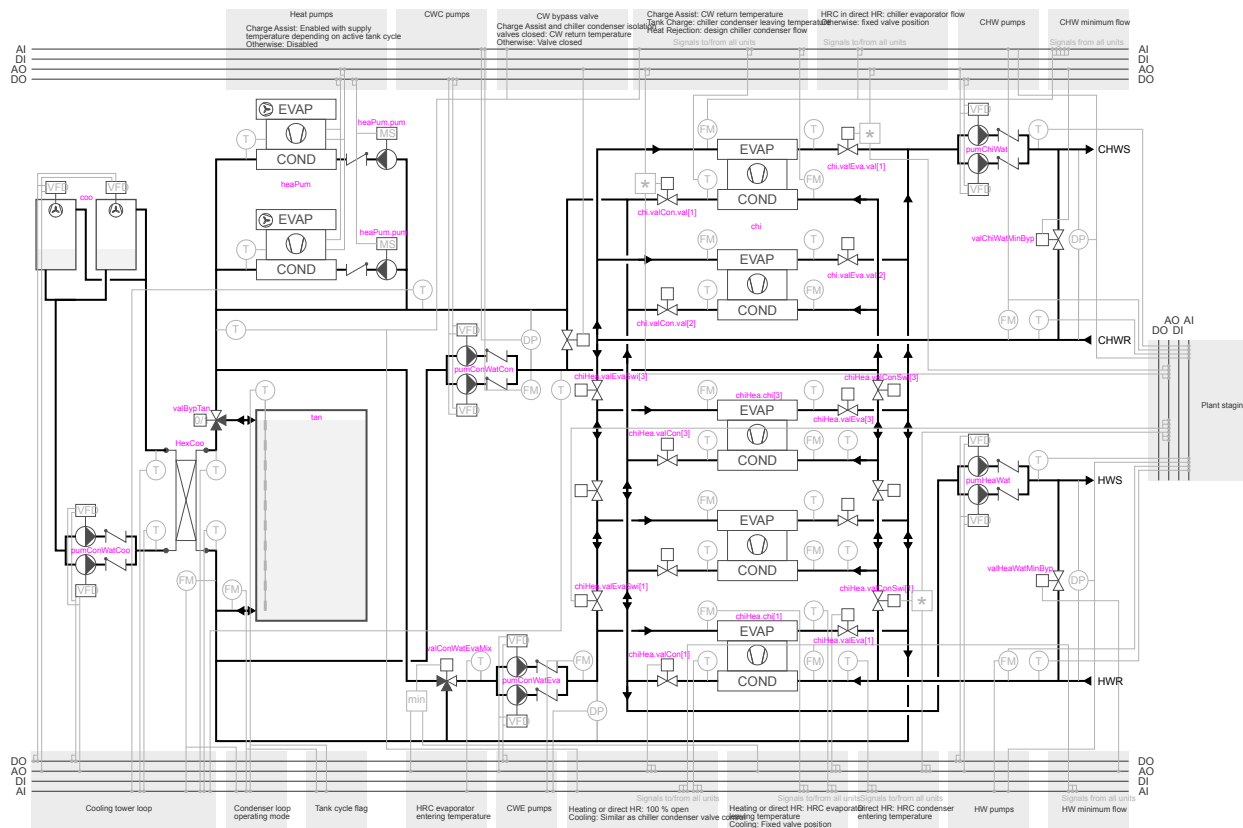


Figure 6. Detailed diagram of the TIER plant as implemented in the Modelica Buildings Library v12.0.0 in `Buildings.DHC.Plants.Combined.AllElectricCWStorage`, with water flow paths and equipment outlined in black and control paths and sensors outlined in gray.

- Average tank temperature is within minimum and maximum bounds, using two instances of `GreaterEqual`.
- Maximum and minimum layer temperatures are within minimum and maximum bounds, using two instances of `GreaterEqual`.

Pumps are responsible for generating pressure to move a liquid. A template for this type of equipment takes the pump mass flow rate, differential pressure, and control signal as inputs. Note that this template could also be used for fans if air is the fluid. With these inputs, multiple component blocks are instantiated from the Modelica Buildings_Requirements Library to verify that the:

- Flow rate is within minimum and maximum bounds, using `WithinBand`.
- Differential pressure is within minimum and maximum bounds, using `WithinBand`.
- Control signal remains stable, using `StableContinuousControl`.
- Minimum mover on and off times are respected, using two instances of `MinimumDuration`.

A final element to consider is judging the severity of the requirement violation. While ideally no requirements are violated, during the iterative design and testing process, it can be helpful to know the severity of violations in order to prioritize addressing them. The Modelica_Requirements Library components report if, when, and where a violation occurs, but not how often, for how long, or by how much. Therefore, for this example case, we added integrator blocks to the violation determination in each of the `GreaterEqual`, `StableContinuousSignal`, and `WithinBand` blocks in order to output for how long a particular requirement was violated.

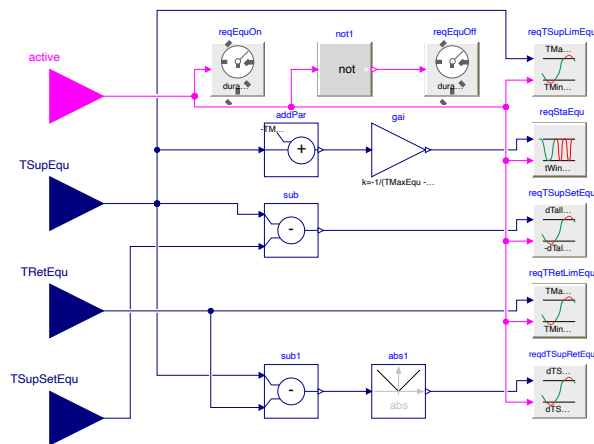


Figure 7. Example requirements template model for heating and cooling production equipment.

3.3 Results

Here we present an example result of using the Modelica Buildings_Requirements Library and HVAC templates to identify issues with simulated operation of the TIER plant. Figure 8 presents the HTML report generated for the simulation. Notice how there are 80 requirements defined and 15 of them were reported to be violated during the simulation. We take a closer look at the reported violation of the plant at $t = 394,654$ seconds to maintain the plant-wide chilled water supply temperature within 0.5 K of the set point. The relevant variables are plotted in Figure 9, where we can see that the requirement is satisfied most of the time, or undecided due to chilled water production being inactive. However, there are actually five distinct violation periods that occur during the year. Focusing on two of them as an example, we gain insight on the issues. The first period at $t = 394,654$ seconds is a violation that occurs the first time chilled water is called for, where it takes extra time for the volume of water in the chilled water loop to cool down to the set point from an initial condition of 20°C. The second period at $t = 1.12859e + 07$ seconds is when the cooling load on the plant reaches its peak, and there is about a 2 hour window when the chilled water temperature set point is not maintained, with a peak excursion of about 2K.

4 Discussion

The observed violations in chilled water supply temperature indicate a few future considerations for defining and using requirements for building and district energy systems. One consideration being whether requirements for starting up from periods of long inactivity, such as outside a cooling season, should be different from more regular start up periods, such as the typical diurnal cycle of building operation during the cooling season.

Another consideration being whether the plant equipment capacity should be increased to prevent set point excursions identified during this simulation, or whether the requirements should be refined to be acceptable of such excursions to avoid the increased capital cost of larger equipment. For example, a second simulation was run where plant sizing was increased by 20% over the initial case. The number of violations in the HTML report reduced from 15 to 10. Notably, the violations that were fixed were the set point tracking violations for the heat recovery chillers and heating plant, the upper limit violation of the average tank temperature, and a minimum runtime violation of the heating plant. The chiller set point tracking violation at $t = 394,654$ seconds discussed in detail earlier was not fixed, but all of the subsequent four violations shown in Figure 9 were fixed. Alternatively, without the capacity increase, set point violations of a central plant may still successfully be accommodated for by increased mass flow of a distribution system, such as airflow through an air handling unit cooling coil. Moreover, in building and district energy systems, it is customary to de-

Requirements violated (15 of 80)

Observation	Requirement	First violation at	Description
chiller	plantRequirement.chiReq[1].reqdTSupRetEqu.req	404246 s	This chiller supply and return temperature difference should be between 4K and 20K once the equipment is active for 10 min.
cooling plant	plantRequirement.cooPlaReq[1].reqTSupSetEqu.req	394654 s	This cooling plant supply temperature set point must be tracked within $\pm 0.5K$ once the equipment is active for 10 min.
cooling plant	plantRequirement.cooPlaReq[1].reqdTSupRetEqu.req	394654 s	This cooling plant supply and return temperature difference should be between 4K and 20K once the equipment is active for 10 min.
heat recovery chiller	plantRequirement.heaChiReq[1].reqTSupSetEqu.req	1.12877e+07 s	This heat recovery chiller supply temperature set point must be tracked within $\pm 0.5K$ once the equipment is active for 10 min.
heat recovery chiller	plantRequirement.heaChiReq[1].reqdTSupRetEqu.req	174000 s	This heat recovery chiller supply and return temperature difference should be between 4K and 20K once the equipment is active for 10 min.
heat recovery chiller	plantRequirement.heaChiReq[2].reqTSupSetEqu.req	1.12865e+07 s	This heat recovery chiller supply temperature set point must be tracked within $\pm 0.5K$ once the equipment is active for 10 min.
heat recovery chiller	plantRequirement.heaChiReq[2].reqdTSupRetEqu.req	290216 s	This heat recovery chiller supply and return temperature difference should be between 4K and 20K once the equipment is active for 10 min.
heating plant	plantRequirement.heaPlaReq[1].reqEquOn.req	3.1536e+07 s	This heating plant must operate at least 10 min when activated.
heating plant	plantRequirement.heaPlaReq[1].reqTSupSetEqu.req	1.12877e+07 s	This heating plant supply temperature set point must be tracked within $\pm 0.5K$ once the equipment is active for 10 min.
heating plant	plantRequirement.heaPlaReq[1].reqdTSupRetEqu.req	174000 s	This heating plant supply and return temperature difference should be between 4K and 20K once the equipment is active for 10 min.
heat pump	plantRequirement.heaPumReq[1].reqEquOn.req	192464 s	This heat pump must operate at least 10 min when activated.
heat pump	plantRequirement.heaPumReq[1].reqdTSupRetEqu.req	394236 s	This heat pump supply and return temperature difference should be between 2K and 20K once the equipment is active for 10 min.
heat pump	plantRequirement.heaPumReq[2].reqEquOn.req	192464 s	This heat pump must operate at least 10 min when activated.
heat pump	plantRequirement.heaPumReq[2].reqdTSupRetEqu.req	394236 s	This heat pump supply and return temperature difference should be between 2K and 20K once the equipment is active for 10 min.
tank	plantRequirement.tanReq[1].greTUp.req	174000 s	This tank average temperature is above the upper bound once the equipment is active for 10 minutes.

Requirements untested (0 of 80)

None

Requirements satisfied (65 of 80)

Figure 8. Requirements HTML report generated for example usage for TIER plant simulation for a large office building in San Francisco, CA. Details for satisfied requirements not shown.

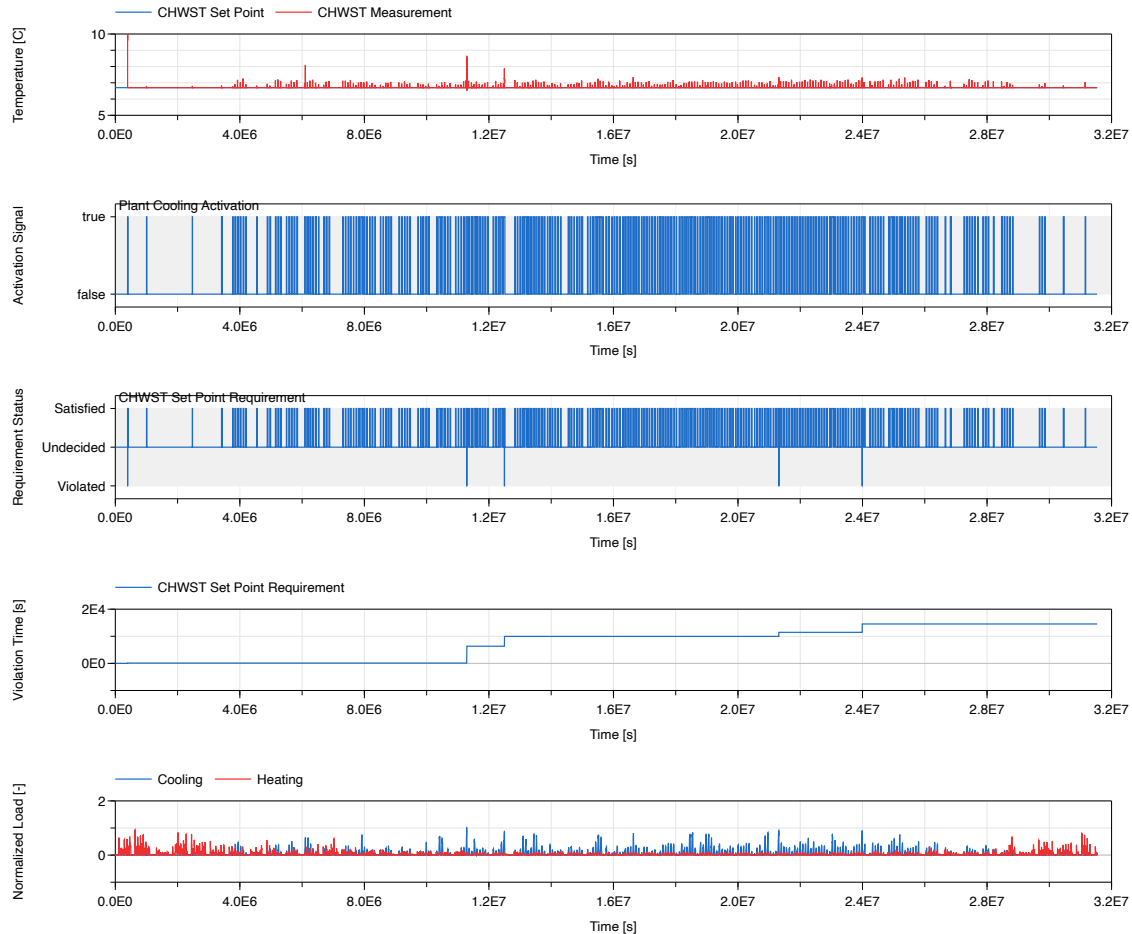


Figure 9. Results for one year of simulation time exploring violation of the TIER plant chilled water supply temperature set point requirement, showing chilled water supply temperature (top) measurement (red) and set point (blue), requirement activation signal (second), requirement status signal (third), requirement violation time integration in seconds (fourth), and normalized plant cooling (blue) and heating (red) loads (bottom).

sign for certain hours of set point violation, as accepting small comfort violations can significantly reduce equipment size, capital costs, and avoids oversizing equipment which may cause low part load performance.

A final note is to acknowledge that the use of the Modelica Buildings_Requirements Library does increase the simulation time of the model. For the simulation in this example, the time increases from 39.9 min without requirements to 51.4 min (1.29x) with requirements. Investigating further, running the validation example from Figure 2 for five seconds of simulation time and with three different solvers in Dymola 2025x shows that the solver affects simulation time, as presented in Figure 10. Notice the difference in trajectories of CPU time between Radau, Dassl, and Cvode. Comparing these simulations with simulations of the same model but without the requirements blocks shows a 7.9x, 3.0x, and 2.1x increase in simulation time when adding the requirements blocks for each solver respectively. Re-running this small test using OpenModelica 1.27.0 results in a 2.7x increase for both Dassl and Cvode (OpenModelica has no Radau). Note that we hypothesize that the example TIER model shows less difference than the small validation model due to the size and complexity of the TIER model, which already introduces numerical overhead due to its number of equations and variables, events, and nonlinear loops. While the change in simulation time may be more or less in each case depending on the system model, solver, and tool, it is something to make note of if running many batches of simulations.

This increase in simulation time also made the authors consider making requirements verification part of a post-processing step that reads the simulation result files. However, for large simulations such as for district energy systems, output files can be several GB large, at an average output interval of around 145 seconds.¹ For control verification, a significantly finer output interval would need to be selected, which also increases simulation time, and it makes post-processing cumbersome due to the need for handling large files.

5 Conclusion

In this paper, we introduced the Modelica Buildings_Requirements Library for formally specifying operational requirements to be verified during simulation for building and district energy and HVAC system applications. We presented the structure and basic components of the library and demonstrated the its use for verifying simulated control performance of a novel central plant HVAC system design. In conclusion, the library proved useful to define and observe operational violations that quickly led to identification of control issues to be addressed. The library eliminated the need for tedious examination of large result files to identify issues and automated this verification process across batches of simulation scenarios.

¹For efficiency, the output is not equidistant.

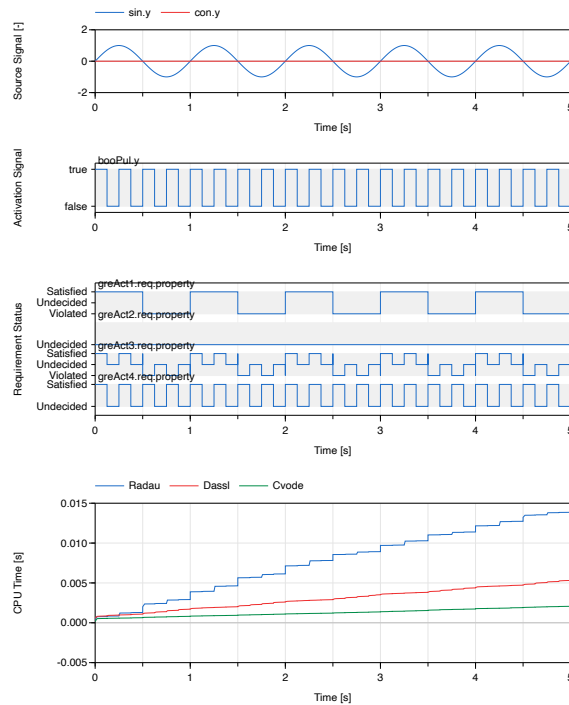


Figure 10. Simulation results for five seconds of simulation time for example model using GreaterEqual from Figure 2, showing sine (blue) and constant (red) signals (top), activation signal pulse (second), the requirement status of the four GreaterEqual instances (third), and CPU time for Radau (blue), Dassl (red), and Cvode (green) solvers (bottom) using Dymola 2025x.

There are additional considerations for future work to develop and use the library. While the library indicates when requirements are violated, it does not indicate the severity of violations, such as how long, how often, nor by how much a violation occurs. While we prototyped a method to determine for how long violations occur, adding capabilities to determine all of this information as a core function of the library and adding it to the output report would be useful to aid users in prioritizing violations to address. Furthermore, using such information in combination with the ability to formally define acceptance criteria that allow for short, small violations would give users an even more accurate assessment of desired performance in various contexts. Lastly, developing and distributing more equipment and system-specific requirements templates, such as for those in the Modelica Buildings Library, would make it easy for users of that library to specify and use requirements in their model-based workflows.

Acknowledgments

This research was developed by Lawrence Berkeley National Laboratory, operated for the U.S. Department of En-

ergy (DOE) under Contract No. DE-AC02-05CH11231. This research was supported by the Environmental Security Technology Certification Program (ESTCP) of the U.S. Department of Defense (DoD), under Contract No. EW23-BB-7686. The California Energy Commission Electric Program Investment Charge Program (EPC-23-023) provided funding for this research.

References

- Batteh, John (2025). “Building a Coffee Cup Thermal Model with Dyad’s Agentic AI”. In: *Online article*. URL: <https://juliahub.com/blog/coffee-cup-thermal-model-dyad-agentic-ai>.
- DOE (2021). *A National Roadmap for Grid-Interactive Efficient Buildings*. Tech. rep. U.S. Department of Energy. URL: <https://gebroadmap.lbl.gov/A%20National%20Roadmap%20for%20GEBs%20-%20Final.pdf>.
- Gill, Brandon (2021). “Solving the Large Building All-Electric Heating Problem”. In: *ASHRAE Journal* 63.10, pp. 12–26.
- Li, Lixiang (2026). “From Intent to Insight: AI-Driven, Physics-Based Modeling with Modelica”. In: *Online article*. URL: <https://modelon.com/blog/from-intent-to-insight-ai-driven-physics-based-modeling-with-modelica/>.
- Otter, Martin et al. (2015). “Formal Requirements Modeling for Simulation-Based Verification”. In: *11th International Modelica Conference*. Versailles, France. DOI: 10.3384/ecp15118625.
- Shehabi, Arman et al. (2024). *2024 United States Data Center Energy Usage Report*. Tech. rep. Lawrence Berkeley National Laboratory. URL: <https://doi.org/10.71468/P1WC7Q>.
- Sommer, Tobias, Matthias Sulzer, Michael Wetter, Artem Sotnikov, Stefan Mennel, and Christoph Stettler (2020). “The reservoir network: A new network topology for district heating and cooling”. In: *Energy* 199, p. 117418. DOI: <https://doi.org/10.1016/j.energy.2020.117418>.
- Sulzer, Matthias, Michael Wetter, Robin Mutschler, and Alberto Sangiovanni-Vincentelli (2023). “Platform-based design for energy systems”. In: *Applied Energy* 352, p. 121955. DOI: <https://doi.org/10.1016/j.apenergy.2023.121955>.
- Wetter, Michael, Massimo Cimmino, et al. (2025). “IBPSA Modelica Working Group: Open-source model development based on open standards to accelerate decarbonization”. In: *Proceedings of Building Simulation 2025: 19th Conference of IBPSA*. Brisbane, Australia. DOI: <https://doi.org/10.26868/25222708.2025.1137>.
- Wetter, Michael, Virak Dee, et al. (2026). *Thermal Reservoir Networks for Modularly Expandable Thermal Microgrids*. Tech. rep. ESTCP Project EW23-7686. URL: <https://escholarship.org/uc/item/6r42g2pt>.
- Wetter, Michael, Paul Ehrlich, et al. (2022). “OpenBuilding-Control: Digitizing the control delivery from building energy modeling to specification, implementation and formal verification”. In: *Energy* 238, p. 121501. DOI: <https://doi.org/10.1016/j.energy.2021.121501>.
- Wetter, Michael, Matthias Sulzer, et al. (2026). “Design, Verify, Deploy: The New Platform-Based Design Paradigm for Large-Scale Energy System Retrofits”. In: *SSRN*. DOI: <http://dx.doi.org/10.2139/ssrn.6557422>.
- Wetter, Michael, Wangda Zuo, Thierry S. Nouidui, and Xiufeng Pang (2014). “Modelica Buildings library”. In: *Journal of Building Performance Simulation* 7.4, pp. 253–270. URL: <https://doi.org/10.1080/19401493.2013.765506>.
- Zanetti, Ettore, David Blum, Antoine Gautier, Paul Raftery, and Hwakong Cheng (2026). “Optimizing Heat Recovery with Storage: Control Validation and Sensitivity Analysis of the Time-Independent Energy Recovery Plant Using Modelica”. In: *Proceedings of 2026 Summer Study on Energy Efficiency in Buildings*. Pacific Grove, CA. URL: <https://www.aceee.org/wp-content/uploads/2026/08/optimizing-heat-recovery-with-storage-control-validation.pdf>.

A Workflow for Transparent and Replicable Modelica Implementations of ASHRAE Guideline 36 Controls for Chiller Plants with Waterside Economizers

Julia Ho¹ Cary Faulkner² Wangda Zuo^{1,3}

¹Department of Architectural Engineering, Penn State University, USA, {j1h7349,wangda.zuo}@psu.edu

²Pacific Northwest National Laboratory, USA, cary.faulkner@pnnl.gov

³National Laboratory of the Rockies, Golden, CO, USA

Abstract

Because of its equation-based, object-oriented nature, Modelica is uniquely positioned to give valuable insights into the performance of complex rule-based controls for heating, ventilating, and air-conditioning (HVAC) systems like ASHRAE Guideline 36 (G36). However, control implementation in Modelica can be arduous, and it has been under-leveraged for G36 control analysis in chiller plants. This work expands on previous research and the authors' experience implementing complex rule-based controls in Modelica by defining and demonstrating a replicable method to implement complex rule-based controls consistent with sequences such as G36 in Modelica. The goals of this control development method is to facilitate complex control implementation in Modelica, reduce model debugging time, and improve control implementation transparency. Implemented G36 controls are shown, and the development and testing of one control block is described in detail using a case study of a chiller plant with a waterside economizer. Then, preliminary full model results are shown as part of the control troubleshooting process. The goal of the workflow is to help other researchers more easily implement complex rule-based controls in Modelica.

Keywords: *rule-based controls, chiller plant, controls, ASHRAE Guideline 36, waterside economizer, modelica*

1 Introduction

ASHRAE Guideline 36 (ASHRAE 2021), or G36, includes a number of heating, ventilating, and air-conditioning (HVAC) control sequences that strive to improve energy efficiency, control stability, reduce engineering and commissioning time, and improve fault detection and diagnostics (TRANE 2021a). It first included chilled water plant applications in its 2021 version (TRANE 2021b) after previously focusing on airside operations. Relatively few analyses to the authors' knowledge have evaluated its chiller plant control sequences, especially for systems with waterside economizers.

Analyses on the airside system control sequences of G36 are much more common. This includes studies quan-

tifying the energy saving potential of G36 for VAV systems (Michael Wetter et al. 2018; Zhang, D. H. Blum, et al. 2020; Zhang, D. Blum, et al. 2022; Lu and Chen 2024; Nassif 2022; Nassif et al. 2022; C. A. Faulkner et al. 2023), investigating fault detection in airside systems (Lu, Fu, et al. 2021; Li et al. 2023; Torabi, B. Gunay, et al. 2023; Torabi, H. B. Gunay, et al. 2024), and investigating other aspects of G36 VAV controls such as uncertainty from component and occupant behavior (Haleem et al. 2020; Elehwany et al. 2024). Many studies (Michael Wetter et al. 2018; Zhang, D. H. Blum, et al. 2020; Zhang, D. Blum, et al. 2022; C. A. Faulkner et al. 2023; Razzano et al. 2025) leverage Modelica to model G36 airside sequences because of its advantages for control-focused modeling.

Analysis of G36-consistent chilled water control sequences began in 2012 when the controls that formed the basis of G36 hydronic controls were developed and analyzed by Taylor (Taylor 2012). Since the publication of the G36-2021, a few more studies have investigated G36 performance in hydronic systems. One study (Lu, Fu, et al. 2021) modeled G36 controls for a building with chilled water and hot water systems to investigate fault detection and diagnostic capabilities and included a waterside economizer in their system analysis. Another study (Roa et al. 2023) used G36's hot water reset control sequences to demonstrate the effectiveness of a method the authors developed. Controls consistent with G36 were developed in Modelica for chiller plants without waterside economizers, and energy use was compared against the operational control sequences for a real-world district cooling system (Ho et al. 2025; Ho 2026). Finally, Grahovac et al (2024) compared G36 controls for chiller plants with waterside economizers to baseline controls using a Modelica model of a theoretical case study in Sacramento, California with a constant load data center and found that G36 could save up to 25% annual energy (Grahovac et al. 2024).

In summary, two previous studies that the authors are aware of model G36-consistent controls for chiller plants with waterside economizers. Lu (2021) focuses on modeling G36 controls for fault detection and diagnostics. Gra-

hovac (2024) models G36 controls for chiller plants with waterside economizers and measures energy efficiency with a constant chilled water load.

This work complements existing work by providing a clear, step-by-step workflow to facilitate the implementation of complex rule-based control sequences like ASHRAE G36 in Modelica. This workflow is used to develop controls that are consistent with ASHRAE G36 and apply them to a Modelica case study of a chiller plant with waterside economizer. This workflow is demonstrated through detailed examples from the authors' implementation of more than fourteen control blocks and fifty control actions related to G36 sequences for chiller plants with waterside economizers in Modelica.

2 Methodology

This section describes the workflow to develop, test, and implement controls consistent with G36 control sequences for chiller plants with waterside economizers, or WSEs. First, a modeling testbed for evaluating G36 controls is defined, and the control blocks implemented using the workflow are described. Then, the control development workflow is described in detail using the waterside economizer enable control block as an example.

2.1 Modeling Testbed

A modeling testbed of a chiller plant with a WSE from (Fan et al. 2021; C. Faulkner et al. 2025) is adopted in this study. No aspects of the physical system have been changed for this study, but the controls have been replaced with the G36-consistent controls developed using the workflow in this paper.

The modeling testbed is comprised of a chiller plant connected to a combined office and data center with a single air handling unit (AHU). EnergyPlus (Crawley et al. 2000) was used to generate dynamic office building loads. The chiller plant supplies chilled water using two parallel chillers and a WSE. The WSE is connected in series with the chillers on the chilled water (CHW) side and in parallel with the chillers on the condenser water (CDW) side. Four headered, parallel, variable frequency drive (VFD) pumps are included: two CHW pumps to supply chilled water to the air handling unit (AHU) and two CDW pumps through the CDW loop that includes two parallel cooling towers, each containing one cell and a VFD fan. The configuration is shown in detail in Figure 1 below, and the Modelica system model is shown in Figure 2.

2.2 Control Development Workflow

The integration of a WSE into existing chiller plant controls affects many control sequences, including cooling tower fan, head pressure, CDW pump, chiller staging, isolation valve, and CHW enabling controls. Therefore, fourteen control blocks describing around 50 control actions are developed in Modelica for chiller plants with water-side economizer controls (Table 1). These control blocks either replace or add to existing, relevant G36-consistent

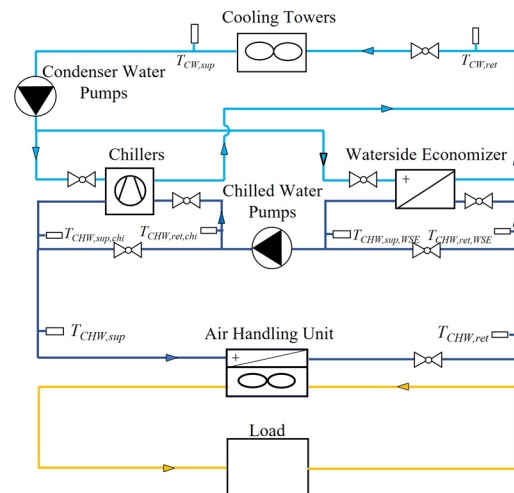


Figure 1. Chiller plant schematic with two parallel chillers and the waterside economizer in parallel on the CDW side, pulled from (C. Faulkner et al. 2025).

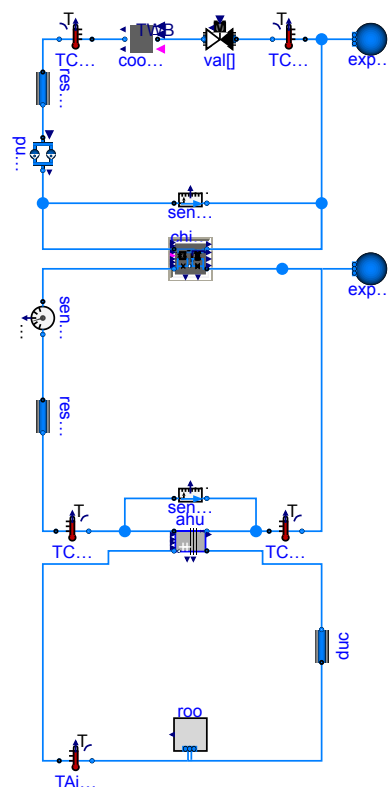


Figure 2. Chiller plant model, from (C. Faulkner et al. 2025).

control blocks that were originally developed for chiller plants without waterside economizers in previous work using the same control development workflow described in this paper (Ho et al. 2025; Ho 2026).

Because G36 control sequences can be complex to implement, a replicable workflow is developed and followed to encourage correct, systematic, and transparent control implementation and reduce model debugging time. The workflow includes six main steps as outlined below and covers the process from the start (written control review and documentation) to the end of the control implementation process (full model debugging). Each step has multiple substeps that are described with examples in this Methodology section. Substeps are labeled using the number of their corresponding step followed by a letter. The workflow's six steps are as follows:

1. G36 is thoroughly reviewed, and a centralized tracking document (CTD) is created for relevant control sequences, associated models, and simulation scripts.
2. Control blocks are developed for subgroups of control actions within each control sequence.
3. Each control block is subject to open loop testing for each control action it executes, and associated testing information is stored in the CTD.
4. Once all control blocks pass open loop testing, they are integrated into subsystem controllers. Subsystem controllers are combined into a partial control block for replicable integration with physical system models.
5. The partial control block is integrated with a physical system model and is verified by inspection through closed loop testing for any control issues that are missed during individual control testing. If control issues are found, the workflow is re-iterated until the model behaves as expected.
6. If unexpected results that are not easily attributable to a control block occur, the controls in the full model are systematically simplified to identify and correct at-fault control blocks.

The following subsections describe each step of the workflow in detail and use the WSE enable control block to demonstrate the process. Regarding wording, the term "control block" is used to describe Modelica components that implement given control sequences. Control sequences are made up of one or more control actions, each of which specifies a single command that the control block executes.

2.2.1 Control Review and Centralized Tracking Documentation

The first step of the workflow is to create a central tracking document (CTD) to be used throughout the development process, and then to thoroughly review and understand the controls.

- 1a. Create a centralized tracking document (CTD) in a software like Excel. Assign CTD columns so that the written control sequences, the control blocks that implement them in Modelica, and associated control testing information can easily be referenced and cross-checked. See Table 3 in Appendix A for the columns used for this case study.
- 1b. Store the CTD in the same repository as the control blocks and models for consistent access.
- 1c. Read the original, written controls and document relevant sequences in the CTD. Careful paraphrasing is encouraged to promote understanding and identify any ambiguities in the original controls.
- 1d. Split each documented control sequence into control actions in the CTD. Ensure that the original control sequence identifiers/labels are included to easily reference back to the original controls if clarification is needed.

As an example, the G36 control sequences dictating waterside economizer enabling and disabling behavior is copied below from the 2021 version of ASHRAE G36 (ASHRAE 2021).

5.20.3.1. Enable waterside economizer (WSE) if it has been disabled for at least 20 minutes and CHWRT upstream of HX is greater than the predicted heat exchanger leaving water temperature (PHXLWT) plus 2 °F. PHXLWT is:

$$\begin{aligned} PHXLWT &= T_{wb} + PA_{HX} + PA_{CT} \\ PA_{HX} &= DA_{HX} * PLR_{HX} \\ PA_{CT} &= m * (DT_{wb} - T_{wb}) + DA_{CT} \end{aligned}$$

where

*T_{wb} = current wetbulb temperature
PA_{HX} = predicted heat exchanger approach
PA_{CT} = predicted cooling tower approach
DA_{HX} = design heat exchanger approach
PLR_{HX} = the lesser of 1 and predicted heat exchanger part load ratio (current chilled water flow rate divided by design HX chilled water flow rate)*

5.20.3.2. Disable WSE when it has run for at least 20 minutes and CHW temp downstream of HX is greater than CHWRT upstream of HX less 1 °F for 2 minutes (i.e., if the HX is not reducing the CHW temp by at least 1 °F).

Table 1. Implemented Control Blocks, Descriptions, and G36 Control Action Locations

Subsystem	Control Blocks	Control Sequence Purpose	G36 Location
CHW	chwPumStaWSE	CHW pump staging and speed	5.20.6.2, 3, 5, 7, 8
	cdwPumSpd_WSE	Condenser water pump speed	5.20.9.7
CDW	cdwPumEna	Condenser water pump enable	5.20.9.2
	headPressureControl_WSE	Chiller head pressure controller; determines CDW pump speed, chiller CDW isolation valve, and fan speed.	5.20.10.5a-6c
	chillerIsolationValveEnable	CDW Isolation Valve Enable	5.20.10.7
	CTFanSpdWSE	CT fan speed control when WSE enabled	5.20.12.2c1i-iii, 5.20.12.2c2i
	fanOffWSE	CT fan enable and disable when WSE enabled	5.20.12.2c2ii
	enaCTCells	Cooling tower Staging	5.20.12.1b-1c, 2f
	coolingTowerHeater	Basin Heater Enable	5.20.15.1a-b
	enaWSE	WSE Enable	5.20.3.1-2
Chiller/WSE	m_WSE	Variable to calculate predicted WSE leaving water temperature on warm water side.	5.20.3.3a-5.20.3.3d
	wseValPos	WSE isolation valve modulation when WSE is controlled with a modulating HX bypass valve	5.20.3.4-6
	chiChwBypValPos	Chiller bypass valve modulation (CHW side)	5.20.3.11
	ChiSta_w_WSE	Chiller staging when WSE enabled	5.20.4.15i, j1-2, k1-k3ii, 11-3

These controls are documented in the case study's CTD and split into discrete control actions. Then, relevant information (such as control sequences, enumerated control sections, applicable systems, and testable control actions) are input into the CTD. This can be seen in Table 3 in Appendix A.

2.2.2 Control Block Development

After all substeps in step 1 are complete, control blocks are implemented in Modelica for closely related groups of control actions (Table 1). The following process is followed for this part of the workflow:

- 2a. Identify groups of closely related control actions.
- 2b. Develop Modelica control blocks from identified control action groups.
- 2c. If needed, create visual representations and/or pseudocode of the written controls related to the control block in question.

In this case study, Modelica Standard Library (MSL) (The Modelica Association and Contributors 2020) and Modelica Buildings Library (MBL) (Wetter et al. 2014) components are used for control block implementation. To facilitate the modeling process, control blocks do not always equate to entire control sequences; multiple control blocks might cover a single sequence, or one block could include control actions in multiple sequences. An example using the WSE enable control is demonstrated below.

Visual representations of the written controls are used when additional clarity is needed. This is demonstrated in Figure 3, which shows a visual representation and associated modeling implementation of the waterside economizer enable control block. Visualizations like the state graph in 3a can provide a helpful bridge between written control comprehension and control block implementation. The state graph demonstrates the two states (on or off) the economizer can be in. Orange arrows show what conditions need to be met to transition from one state to another, and conditions are described in mathematical terms.

Figure 3b shows the Modelica model implementation using state graph components in MSL. Because of the multiple compound statements (and/or conditions) that have different time delay requirements, this control block requires intermediate states (shown as white blocks) and transitions (shown as black rectangles). For example, the "offReady" and "offReady1" states in the model are intermediate states within the overall "enabled" state that allow the waterside economizer to disable only when they are both active.

2.2.3 Open Loop Testing Workflow

After each control block is created, it is combined with controlled inputs in a test model and then simulated to verify that it behaves in accordance with the original control. Each test model tests a single control action by default, although some models test multiple control actions if it is convenient and logical to do so. After successful testing, a Modelica script (.mos file) to simulate and plot relevant timeframes, variables, and parameters is saved with

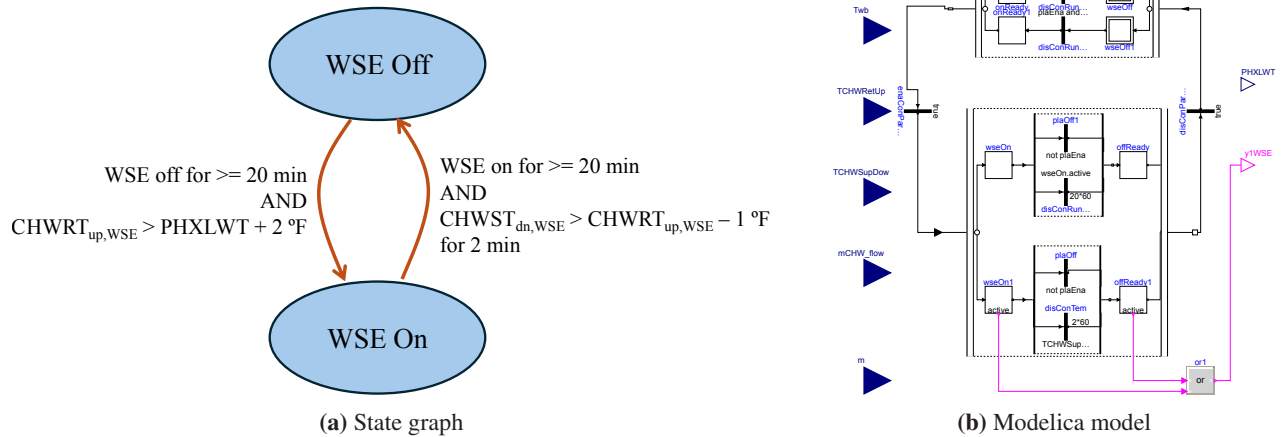


Figure 3. Visualization and modeling implementation of waterside economizer enable control block.

the test model for replicability. The workflow, which is described below, is based on Example model implementation in MBL and MSL.

- 3a. Identify the control action to be tested.
- 3b. Create a simulatable model composed of the relevant control block and input variables that should trigger an expected behavior (open loop testing). Save the model with a name that refers to the original enumerated control sequences.
- 3c. Simulate relevant timeframes and plot the relevant input and output variables.
- 3d. Save a Modelica script with the desired simulation and plotting commands. Use the identifier of the written control action being tested (e.g., 5.20.3.1) as the name of the .mos script. (see Table 3 in Appendix A for an example).
- 3e. Include relevant information (such as the Modelica control block, Modelica simulation and plotting script, and example model) in the CTD.

An example that tests the control action consistent with G36-2021 5.20.3.2 (quoted from G36 above) within the WSE enable control block is shown in Figure 4. Relevant information is included in the CTD in Table 3. To trigger a desired action, the WSE enable control is connected to relevant input variables in a test model. It is simulated, and the output control behavior is then evaluated for consistency with the written control. When all simulatable control actions behaviors within the control block are consistent with the written controls, the control block is considered implemented. Results for this example model are further discussed in the Results section.

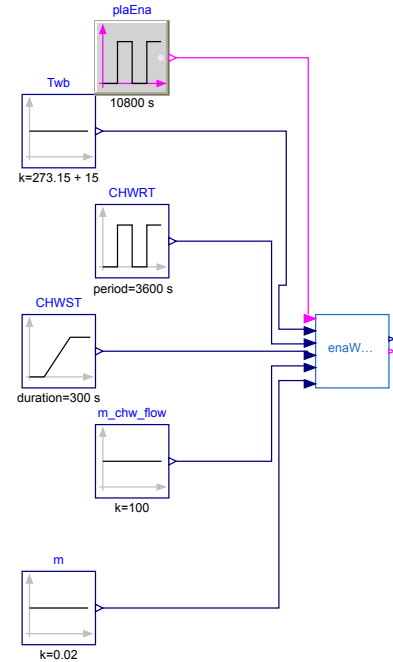


Figure 4. Test model testing behavior consistent with control action 5.20.3.2 of G36 WSE enable control (wseEna).

2.2.4 Integration Into Subsystem Controllers

After testing the control actions of all individual control blocks, the control blocks are integrated into CDW, CHW, and cooling (chiller and WSE) equipment subsystem controllers. Then, they are combined into partial control blocks that are consistent with all relevant G36 control sequences. This partial control block can be integrated with the modeling testbed. The enumerated substeps for this process are described below.

- 4a. Define subsystem controllers.
- 4b. Connect control blocks within each relevant subsystem controller.
- 4c. Propagate necessary parameters.
- 4d. Run syntactical checks on each subsystem controller to ensure no connections or parameters are missing.
- 4e. Create a partial model for the entire control sequence for consistent integration with physical system models.

Figure 5 shows the layered integration of control blocks and subsystem controllers into a single partial model representing all controls consistent with relevant G36 sequences. The WSE enable control block and the cooling equipment subsystem controller are used as examples. Control blocks are iteratively revised during this stage to ensure that they include all necessary inputs and outputs needed by other control blocks and the physical system model. If a control block is revised, it is retested to ensure control actions still comply with G36. In contrast to the individual control blocks, only syntactical checks are conducted for subsystem controllers and partial models. This is because the large number of variables, parameters, and scenarios possible in the combined controllers makes selectively testing control behavior infeasible.

2.2.5 Integration with Physical System Model and Closed Loop Testing

The full chiller plant model with G36 controls is shown in Figure 6. After integrating the partial control model with the modeling testbed, the model is simulated and evaluated for reasonable behavior (closed loop testing). Undergoing closed loop testing by simulating the full model with the implemented controls can give information about control behavior, stability, and coordination that is difficult to measure in individual control testing. After addressing all anomalous behavior in the full model, the full model can be used for control analysis and evaluation. The workflow for integrating with the physical system model is as follows:

- 5a. Integrate the partial control model with the physical system model.
- 5b. Simulate and correct any control issues.

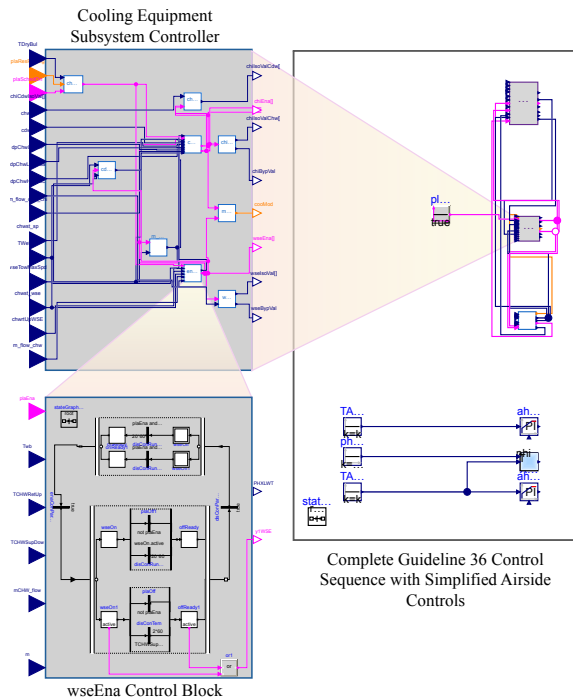


Figure 5. Integration of single control blocks into subsystem controllers, and subsystem controllers into a partial model that can be integrated with a physical system model.

- 5c. If control issues are found that are clearly related to a specific control block, iterate through steps 2-5 of the workflow to correct.
- 5d. If control issues with no clear source are found, use step 6 of the workflow.

2.2.6 Diagnosing Difficult-to-Identify Control Issues

While some unexpected simulation behaviors can be intuitively attributed to a single control, others can cause cascading, system-wide effects that make finding the at-fault

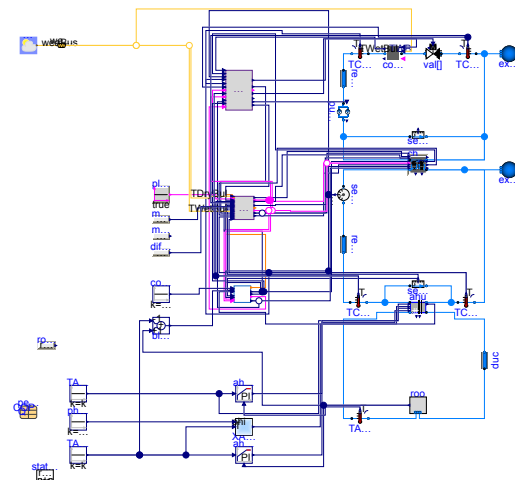


Figure 6. Full physical system and controls model.

control block difficult. In situations where the at-fault control block is unknown, the following steps should be used:

- 6a. Duplicate the full model into a new folder to avoid experimenting with the original copy of the full model.
- 6b. Replace half of the implemented controls with the simplest controls possible (PID controls and constant values) and test if the problem still occurs.
- 6c. If the problem persists, replace another half of the implemented controls that are still in the model (one-fourth of the total) with simple controls and retest.
- 6d. If the problem does not persist, repeat step 6b with the other half of the implemented controls and retest.
- 6e. Continue steps 6b and 6c until you have identified the control block that is consistently present when the problem occurs. This is likely the at-fault control block.

This method leverages the binary search method philosophy (repeatedly splitting your search space in half to find a target value) to quickly identify control blocks that could be causing control issues. This is a highly efficient search method that precludes the need to test each control block individually in the full model or arbitrarily troubleshooting the model.

This method is implemented in the case study to diagnose system instability that permeated throughout the physical system model. To begin to isolate the at-fault control block (and in line with step 6b), Figure 7 below shows a duplicate of the full model during the first iteration of testing where approximately one-half (two of the three) implemented control blocks are replaced with simple controls to isolate the stability issue. This iterative isolation using the binary search philosophy led to the efficient identification of the control issue, the results of which are discussed in the Results section.

2.3 Simulation Setup

All models and control components are developed using MSL 4.0.0 (The Modelica Association and Contributors 2020) and a version of MBL 6.0.0 (Wetter et al. 2014) that is upgraded from MSL 3.2.2 to MSL 4.0.0. The models are simulated using Dymola 2025x (n.a. 2024) with the Dassl solver, a tolerance of 0.0001, and 500 output intervals.

3 Preliminary Results

Results from individual control and full model simulations are presented to demonstrate the control development workflow. The model is simulated using TMY3 weather data from Denver, Colorado in ASHRAE climate zone 5B (cool and dry). Individual control results use the

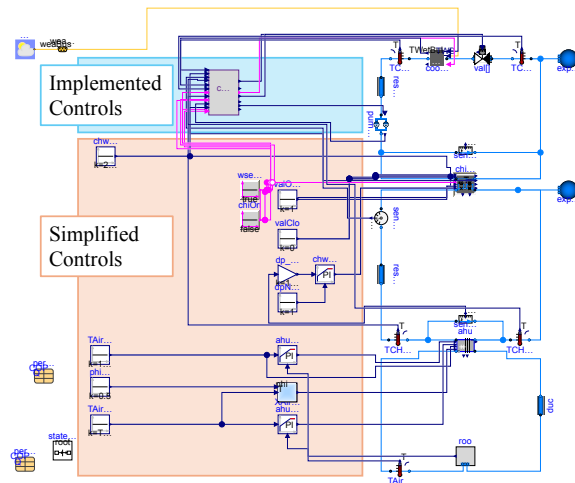


Figure 7. A full system model with approximately half of the implemented control blocks replaced with simple control signals.

same example control (WSE enabling) as in the methodology section. Full model results are preliminary and should not be used to make conclusions about the performance of G36; rather, they are used to demonstrate how control issues can be identified that might have slipped past the individual testing phase.

3.1 Open Loop Control Results

As an example of individual, open loop control testing, the test model results evaluating control block consistency with G36-2021 5.20.3.2 are shown in Figure 8. The model and control block that are used to create these results are shown in Figures 4 and 3b. The exact G36 control action is shown below for ease of understanding, with annotations for the two relevant transition conditions (temperature and runtime) (ASHRAE 2021):

5.20.3.2. Disable WSE when it has run for at least 20 minutes [runtime condition] and CHW temp downstream of HX is greater than CHWRT upstream of HX less 1 °F for 2 minutes (i.e., if the HX is not reducing the CHW temp by at least 1 °F) [temperature condition].

In Figure 8, the top graph shows the status of the WSE and the chiller plant. A true condition indicates that the WSE/chiller plant is enabled. The middle graph shows the CHW return temperatures (CHWRT) and CHW supply temperatures (CHWST). Finally, the bottom graph shows the "enable fire" conditions sent by the transition signals in the staging model. These conditions are true if a condition to transition between two states (such as enabled and disabled) is being met. The condition becomes false if another state has become active, i.e., the transition has successfully occurred, or if the condition stops being true.

As expected, both temperature and runtime conditions must be met in order for the waterside economizer to be

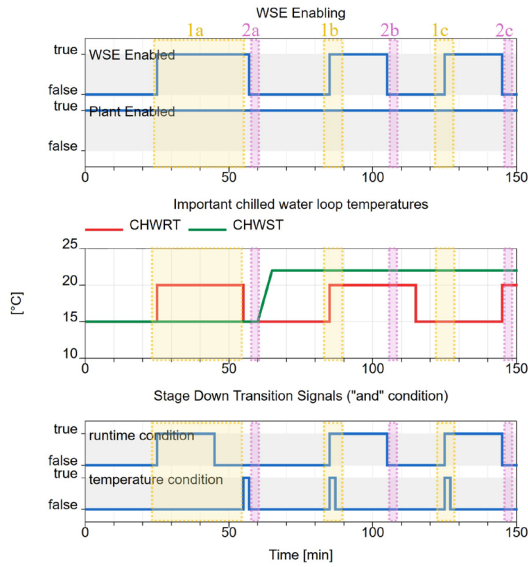


Figure 8. Output results from test model.

disabled. To further explain, Figure 8 is highlighted in six different time frames. In time frame **1a**, the CHWST is more than 1 °F lower than the CHWRT while the WSE is enabled, meaning that the temperature condition is false. This prevents the WSE from disabling even though the runtime condition is met at around $t = 45 \text{ min}$. The WSE is only disabled at around $t = 55 \text{ min}$ when the CHWRT reduces to the same temperature as the CHWST. At this point, both the runtime and temperature conditions have been met. For the rest of the simulation, the CHWRT is consistently below the CHWST. This means that the temperature condition is always met two minutes after the WSE is enabled, as shown in timeframes **1b** and **1c**. Therefore, the only condition preventing the WSE from being disabled is the runtime condition, which is met twenty minutes after being enabled (time frames **2b** and **2c**). As expected, meeting the runtime condition means both conditions have been fulfilled, and the WSE is disabled.

It is important to note that this open loop testing method is not fool-proof. Although the explicit conditional statement that "should" happen is tested, the implicit conditions that "should not" happen are not always tested. These "should not happen" situations can be missed during this phase of testing, which is why behavioral analysis at the full model level is important.

3.2 Diagnosis of Difficult-to-Identify Control Issues Through Closed Loop Testing

In this subsection, a control instability is identified in the full model that propagates through the entire model and therefore makes identifying the at-fault control block difficult. Using step 6 of the replicable control workflow described in this paper, the relevant control block is isolated, identified, and revised, thereby fixing the control instabil-

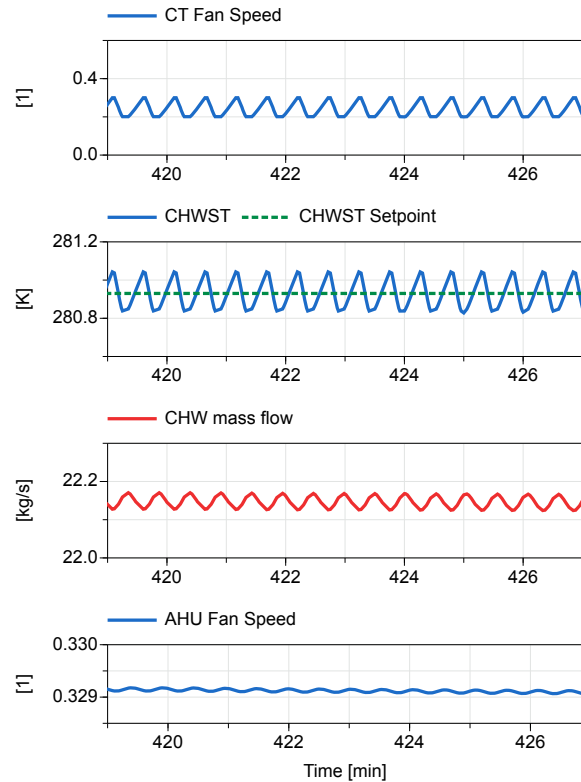


Figure 9. Oscillations propagating throughout the physical system model during simulation.

ity.

As shown in Figure 9, initial simulations of the model experience significant instabilities throughout the entire physical system (CDW, CHW, and airside loops), which makes quickly identifying potential at-fault control blocks difficult. To isolate the problem, individual control blocks in the full model are replaced with simple controllers (PID or constant values) using step 6 of the workflow until a single control block causing the issue is identified. Once isolated, this issue is diagnosed as unstable fan speed PI controls that are exacerbated by large numbers of state events, which are generated from overly complex implementation of fan enable controls.

Two revisions correct the stability issue. First, the proportional gain of the fan speed control is reduced from 0.5 to 0.1, which increases control stability. Second, the fan enable control is re-implemented to reduce the number of state events without changing the control behavior. After the two revisions are implemented, the model simulates more than thirteen times faster than the original case when tested in a representative timeframe of 0 (January 1 at 12am) to 5e5 seconds (around January 6 at 7pm), as seen in Table 2.

3.3 Discussion

Modelica can be a uniquely valuable tool for detailed control evaluation, but control implementation in Modelica can be arduous and error-prone. This workflow therefore

Table 2. Computational times and number of state events for each case.

	Simulation Time (s)	State Events
Original	332	13527
Corrected	25	803

attempts to facilitate complex control implementation in Modelica through systematic documentation, implementation, open and closed loop testing, and troubleshooting methods. The authors use this workflow to implement, test, and troubleshoot fourteen control blocks consistent with G36 in Modelica, as demonstrated in the case study presented in this paper. This workflow is not foolproof for multiple reasons, many of which are due to the highly manual nature of the workflow. For example, the CTD is only highly effective if always updated, and the open loop control testing process is arduous and can require updating the simulatable testing model every time a change is made to the control block. Both of these examples require consistent discipline from the developer.

3.3.1 Future Work

Automating parts of the described workflow would further streamline the process and make it less vulnerable to containing obsolete and confusing models and information. Furthermore, incorporating continuous integration (CI) testing and ensuring control development compliance with standards such as ASHRAE Standard 231 could make control implementation more robust (ASHRAE 2026). Future studies with the models developed in this work can compare control performance with other advanced rule-based control sequences and with multiple climates.

3.3.2 Data Availability

The models used in this study are not publicly available because of licensing and intellectual-property restrictions. Subject to the necessary approvals, the authors plan to publicly release the newly developed Guideline 36-consistent controls, open-loop testing models, Modelica simulation scripts, and centralized-control documentation in the future. Inquiries regarding access to specific materials may be directed to the corresponding author.

4 Conclusion

This work presents a replicable workflow to facilitate the implementation of complex rule-based controls like ASHRAE Guideline 36 in Modelica. It covers planning, implementing, testing, and troubleshooting controls in Modelica and emphasizes making control implementation systematic and transparent. This emphasis reduces model debugging time in the authors' experience. Using this workflow, the authors develop fourteen control blocks and fifty control actions related to G36 sequences for a case study of a chiller plant with a waterside economizer in Modelica. Examples from the case study demonstrate the full scope of the workflow, from documenting written

controls to troubleshooting full model control instability, with the goal of making the workflow understandable and replicable by future researchers. Hopefully, future Modelica researchers will be able to use this workflow to streamline their own control implementation processes.

Acknowledgements

This research was performed under an appointment to the Building Technologies Office (BTO) IBUILD- Graduate Research Fellowship administered by the Oak Ridge Institute for Science and Education (ORISE) and managed by Oak Ridge National Laboratory (ORNL) for the U.S. Department of Energy (DOE). ORISE is managed by Oak Ridge Associated Universities (ORAU), and ORNL is managed by UT-Battelle LLC. All opinions expressed in this paper are the author's and do not necessarily reflect the policies and views of DOE, EERE, BTO, ORISE, ORAU, ORNL or UT-Battelle. ORISE is managed by ORAU under DOE contract number DE-SC0014664, and ORNL is managed by UT-Battelle LLC under DOE contract number DE-AC05-00OR22725. This work was also authored in part by the National Laboratory of the Rockies for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. The views expressed herein do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

- ASHRAE (2021). *ASHRAE Guideline 36-2021: High-Performance Sequences of Operation for HVAC Systems*. Tech. rep. ASHRAE.
- ASHRAE (2026). *ANSI/ASHRAE Standard 231-2026*. American Society of Heating, Refrigerating and Air-Conditioning Engineers, Inc.
- Crawley, Drury, Curtis Pedersen, Linda Lawrie, and Frederick Winkelmann (2000-11). "EnergyPlus: Energy Simulation Program". In: *Ashrae Journal* 42, pp. 49–56.
- Elehwany, Hussein, Burak Gunay, Mohamed Ouf, Nunzio Cotrufo, and Jean-Simon Venne (2024). "Evaluating common supply air temperature setpoint reset strategies with varying occupancy patterns and behaviours". In: *Building and Environment* 266. DOI: 10.1016/j.buildenv.2024.112129.
- Fan, Chengliang et al. (2021). "Open-source Modelica models for the control performance simulation of chiller plants with water-side economizer". In: *Applied Energy* 299, p. 117337.
- Faulkner, Cary, Julia Ho, Chengnan Shi, Chengliang Fan, Nasim Ildiri, and Wangda Zuo (2025). "Development of near-optimal advanced control sequences for chiller plants with water-side economizers in U.S. Climates (ASHRAE RP-1661)". In: *Science and Technology for the Built Environment* 31 (1), pp. 18–35. ISSN: 2374474X. DOI: 10.1080/23744731.2024.2404814.
- Faulkner, Cary A, Robert Lutes, Sen Huang, Wangda Zuo, and Dragana Vrabie (2023-07). "Simulation-based assessment of ASHRAE Guideline 36, considering energy performance, indoor air quality, and control stability". In: *Building and Environment* 240, p. 110371. ISSN: 0360-1323. DOI: 10.1016/j.buildenv.2023.110371. URL: <https://www.sciencedirect.com/science/article/pii/S0360132323003980>.

- Grahovac, Milica, Paul Ehrlich, Jianjun Hu, and Michael Wetter (2024-04). "Model-based data center cooling controls comparative co-design". In: *Science and Technology for the Built Environment* 30 (4), pp. 394–414. ISSN: 2374-4731. DOI: 10.1080/23744731.2023.2276011. URL: <https://doi.org/10.1080/23744731.2023.2276011>.
- Haleem, Shadi M Abdel, Gregory S Pavlak, and William P Bahnfleth (2020-10). "Performance of advanced control sequences in handling uncertainty in energy use and indoor environmental quality using uncertainty and sensitivity analysis for control components". In: *Energy and Buildings* 225, p. 110308. ISSN: 0378-7788. DOI: 10.1016/j.enbuild.2020.110308. URL: <https://www.sciencedirect.com/science/article/pii/S0378778820306964>.
- Ho, Julia (2026). "Equation-based modeling for district cooling system controls evaluation: a case study of a university campus". Accepted by the graduate school; to be published in university's electronic thesis and dissertations website. MA thesis. Penn State.
- Ho, Julia, Hongjun Li, Jianjun Hu, Ardeshtir Moftakhari, and Wangda Zuo (2025-08). "Exploring the potential of ASHRAE Guideline 36 in district cooling systems". In: *Proceedings of the 19th IBPSA Conference*, pp. 0001–0008.
- Li, Yanfei, Piljae Im, Seungjae Lee, Yeonjin Bae, Yeobeom Yoon, and Sangkeun Lee (2023). "Sensor Incipient Fault Impacts on Building Energy Performance: A Case Study on a Multi-Zone Commercial Building". In: *Buildings* 13 (2). DOI: 10.3390/buildings13020520.
- Lu, Xing and Yan Chen (2024). "Energy Saving Estimation of ASHRAE Guideline 36 Supervisory Setpoint Reset Controls in a Commercial Large Office Building". In: *ASHRAE Transactions*. Vol. 130, pp. 148–156.
- Lu, Xing, Yangyang Fu, Zheng O'Neill, and Jin Wen (2021-12). "A holistic fault impact analysis of the high-performance sequences of operation for HVAC systems: Modelica-based case study in a medium-office building". In: *Energy and Buildings* 252. ISSN: 03787788. DOI: 10.1016/j.enbuild.2021.111448.
- n.a. (2024). *Dymola: A Complete Tool for Modeling and Simulation*. URL: <https://www.3ds.com/products/catia/dymola>.
- Nassif, Nabil (2022). "Field and Simulation Testing and Improvement of ASHRAE Guideline 36 Duct Static Pressure Resetting Algorithm". In: *ASHRAE Transactions*. Vol. 128, pp. 253–260. ISBN: 978-195551606-8.
- Nassif, Nabil, Mostafa Tahmasebi, Iffat Ridwana, and Pejman Ebrahimi (2022). "New Optimal Supply Air Temperature and Minimum Zone Air Flow Resetting Strategies for VAV Systems". In: *Buildings* 12 (3). DOI: 10.3390/buildings12030348.
- Razzano, Giuseppe, Silvio Brandi, Marco Savino Piscitelli, and Alfonso Capozzoli (2025-03). "Rule extraction from deep reinforcement learning controller and comparative analysis with ASHRAE control sequences for the optimal management of Heating, Ventilation, and Air Conditioning (HVAC) systems in multizone buildings". In: *Applied Energy* 381. ISSN: 03062619. DOI: 10.1016/j.apenergy.2024.125046.
- Roa, Carlos Duarte, Paul Raftery, and Anand Prakash (2023). *Field Demonstration of the Brick Ontology to Scale up the Deployment of ASHRAE Guideline 36 Control Sequences*. Tech. rep. UC Berkeley: Controls and Information Technology. URL: <https://escholarship.org/uc/item/5zt2d66r>.
- Taylor, Steven (2012-06). "Optimizing Design & Control Of Chilled Water Plants Part 5: Optimized Control Sequences". In: *ASHRAE Journal* 54, pp. 56–74.
- The Modelica Association and Contributors (2020-06). *Modelica Standard Library v4.0.0*. URL: <https://github.com/modelica/ModelicaStandardLibrary/releases/tag/v4.0.0>.
- Torabi, Narges, Burak Gunay, William O'Brien, and Shane Ferreira (2023). "A Building Performance Simulation-Based Investigation on Fault Tolerance of Sequences of Operation for VAV AHU Systems". In: *ASHRAE Transactions*. Vol. 129, pp. 419–427. ISBN: 978-195551664-8.
- Torabi, Narges, H Burak Gunay, and William O'Brien (2024). "A simulation-based investigation of the robustness of sequences of operation to zone-level faults in single-duct multi-zone variable air volume air handling unit systems". In: *Indoor and Built Environment* 33 (9), pp. 1665–1679. DOI: 10.1177/1420326X241265277.
- TRANE (2021a-05). "ASHRAE Guideline 36". In: *TRANE Engineers Newsletter* 50-2.
- TRANE (2021b-09). *Guideline 36-2021: What's New and Why It's Important*.
- Wetter, M, W Zuo, T S Nouidui, and X Pang (2014). "Modelica Buildings Library". In: *Journal of Building Performance Simulation* 7 (4), pp. 253–270.
- Wetter, Michael, Jianjun Hu, Milica Grahovac, Brent Eubanks, and Philip Haves (2018). "OpenBuildingControl: Modeling feedback control as a step towards formal design, specification, deployment and verification of building control sequences". In: *ASHRAE and IBPSA-USA Building Simulation Conference*, pp. 775–782.
- Zhang, Kun, David Blum, Hwakong Cheng, Gwelen Paliaga, Michael Wetter, and Jessica Granderson (2022-03). "Estimating ASHRAE Guideline 36 energy savings for multi-zone variable air volume systems using Spawn of EnergyPlus". In: *Journal of Building Performance Simulation* 15 (2), pp. 215–236. ISSN: 1940-1493. DOI: 10.1080/19401493.2021.2021286. URL: <https://doi.org/10.1080/19401493.2021.2021286>.
- Zhang, Kun, David H. Blum, Milica Grahovac, Jianjun Hu, Jessica Granderson, and Michael Wetter (2020-11). "Development and Verification of Control Sequences for Single-Zone Variable Air Volume System Based on ASHRAEGuideline 36". In: *Proceedings of the American Modelica Conference 2020, Boulder, Colorado, USA, March 23-25, 2020*. Vol. 169. Linköping University Electronic Press, pp. 81–90. DOI: 10.3384/ecp2016981.

Appendix A

Table 3. Example of Centralized Tracking Documentation

Control Sequence	Equivalent Modelica Control Component	G36 Section	Applicable System	Testable Control Action	Simulation Script	Modelica Example Using Simulation Script	Passed Open Loop Test?	Notes
WSE Enable/Disable	WSEControlLogics. Controls. Guideline36. Chiller.enaWSE	5.20.3.1	Plants with wse	Enable WSE if it has been disabled for ≥ 20 min and CHWRT upstream of HX is > predicted HX leaving water temp (PHXLWT*) + 2F.	g36_5.20.3.1.mos	WSEControlLogics. Controls. Guideline36. Validation. WSEStaging.g36_5_20_3_1	y	Calculated PHXLWT by hand - calculations are correct.
WSE Control Enable/Disable	WSEControlLogics. Controls. Guideline36. Chiller.enaWSE	5.20.3.1	Plants with wse and primary-only plants	PLR_HX = min(1, pred HX PLR (m_current / design HX CHW flow rate))	g36_5.20.3.1.mos	WSEControlLogics. Controls. Guideline36. Validation. WSEStaging.g36_5_20_3_1	y	Not explicitly tested, but can be seen in code.
WSE Control Enable/Disable	WSEControlLogics. Controls. Guideline36. Chiller.enaWSE	5.20.3.2	Plants with wse	Disable WSE when it has run for ≥ 20 min and CHW temp downstream of HX > CHWRT upstream HX - 1F for 2 min. "(ie, if the HX is not reducing the CHW temp by at least 1degF)"	g36_5.20.3.2.mos	WSEControlLogics. Controls. Guideline36. Validation. WSEStaging.g36_5_20_3_2	y	None

*Equations for PHXLWT are excluded from Table 3 due to limited space, but are included in the project CTD.

A Unified Multi-Fidelity Modelica Library for Desiccant Wheel Simulation

Hang Guan¹ Mingzhe Liu¹ Zhiyao Yang¹ Zheng O'Neill¹

¹J. Mike Walker '66 Department of Mechanical Engineering, Texas A&M University, USA, {h_guan, mingzhe.liu, z.yang, zoneill}@tamu.edu

Abstract

Desiccant wheels (DWs) are widely used for humidity control in HVAC and industrial drying systems. However, publicly available physics-based models are limited, many implementations lack reusable component-based interfaces, and conventional single-element formulations do not directly provide simultaneous device-level outlet states for both air streams. This paper presents a modular Modelica library that integrates dynamic physics-based models, including single- and multi-element formulations, with steady-state effectiveness-based and performance curve-based models within a unified plug-and-play structure. The physics-based models also support configurable geometries and desiccant materials. An acausal and replaceable architecture enables flexible model selection without modifying system configurations. Model selection and the applicability of the proposed library to energy recovery wheels are also discussed.

Keywords: Desiccant wheel; acausal modeling; physics-based models; reduced-order models; heat and mass transfer

1 Introduction

Dehumidification is essential in many industrial processes and heating, ventilation, and air-conditioning (HVAC) applications, where accurate moisture control affects product quality, occupant comfort, and energy efficiency. Conventional vapor-compression dehumidification systems are widely used but often require high energy input and reheating of over-cooled air, leading to additional energy penalties. Their dehumidification performance also degrades when low dew-point conditions are required (Guan, M. Liu, et al. 2024). The growing emphasis on energy efficiency and the integration of renewable energy sources in building systems has intensified the need for alternative dehumidification technologies that can overcome these limitations.

Desiccant wheels (DWs) have been investigated as an alternative technology for dehumidification, offering high dehumidification efficiency in a compact rotary configuration. The operating principle of a desiccant wheel involves the continuous rotation of a hygroscopic matrix through two distinct air streams: a process stream from which moisture is adsorbed, and a regeneration stream that

desorbs the accumulated moisture using thermal energy (Leva et al. 2014). However, a key challenge limiting the widespread adoption of desiccant wheel technology is the high thermal energy demand required for desiccant regeneration, particularly when conventional materials such as silica gel or zeolites are employed (Speerforck et al. 2017).

Recognizing this challenge, substantial research efforts have focused on improving desiccant wheel performance through multiple pathways. Waste or low-grade heat can be used for regeneration to improve overall system efficiency (Speerforck et al. 2017). In parallel, advanced desiccant materials with lower regeneration temperature requirements, such as metal-organic frameworks (MOFs), composite desiccants, and polymer-based materials, have attracted increasing attention (Abdelgaied et al. 2023; Park et al. 2023; Zeng et al. 2021). Although these material and system innovations show considerable promise, experimental evaluation of new materials and system configurations remains costly and time-consuming.

Modeling and simulation provide essential tools to accelerate the development and optimization of desiccant wheels. Existing studies (Guan, M. Liu, et al. 2024) generally classify DW models into several categories, including physics-based, empirical models (effectiveness-based and performance curve-based models), and data-driven models. Physics-based models employ fundamental heat and mass transfer equations to capture the transient sorption dynamics, temperature profiles, and humidity gradients within the desiccant matrix (Zhang et al. 2003; M. Liu et al. 2026). These detailed models are particularly valuable for fundamental research, material evaluation, and geometry optimization (Guan, M. Liu, et al. 2025), but impose significant computational demands that limit their use in system-level simulations. Effectiveness-based models relate outlet conditions to inlet states through effectiveness parameters, offering a balance between physical fidelity and computational efficiency suitable for system integration (Habib et al. 2020). Performance curve-based models employ empirical correlations derived from manufacturer data or detailed simulations, providing rapid predictions for annual energy simulations and optimization studies at the cost of reduced physical insight (Ma et al. 2017). Data-driven methods have also been explored, but they typically require extensive training data and may have limited extrapolation capability (Gherzi et al. 2026).

Despite this extensive body of modeling research, several critical limitations persist in the current state of the art. First, to the authors' knowledge, there is a notable lack of publicly available physics-based desiccant wheel models, with most implementations remaining proprietary or inaccessible to the broader research community, thereby limiting reproducibility and collaborative advancement. Second, many existing models are implemented in MATLAB or Engineering Equation Solver (EES) (H. Liu et al. 2025). While EES is equation-based and acausal, these implementations lack the component-oriented, connector-based architecture of Modelica for hierarchical model reuse and plug-and-play system integration (Wetter et al. 2014). Third, conventional single-element formulations represent one wheel sector that alternates between process and regeneration modes. At any given time, such formulations provide the outlet state only for the air stream associated with the active mode and therefore do not directly represent the simultaneous operation of both air streams at the full-wheel level (Zhang et al. 2003).

To address these gaps, this paper develops a modular Modelica library that provides a common interface for dynamic physics-based and steady-state reduced-order desiccant wheel models. The physics-based models capture cyclic sorption behavior, support configurable geometries and desiccant materials, and include a full-wheel multi-element implementation that simultaneously calculates the process- and regeneration-side outlet states. Effectiveness-based and performance curve-based models provide computationally efficient alternatives for system-level simulation. By leveraging the acausal and modular nature of Modelica, the proposed framework enables plug-and-play model interchangeability and seamless integration into dehumidification systems.

2 Library Architecture

The desiccant wheel modeling library is developed based on Modelica Buildings Library 12.1.0 (Wetter et al. 2014) and follows its modular, hierarchical, and acausal modeling philosophy. The equation-based formulation of Modelica enables flexible integration into system-level simulations without requiring predefined input–output causality, which is particularly advantageous for HVAC systems in building and industrial applications where components must be interconnected in a consistent and reusable manner.

As shown in Figure 1, the library is organized into three main packages: *PhysicsBased*, *ReducedOrder*, and *Examples*, together with a *UsersGuide* for documentation. The *PhysicsBased* package contains dynamic high-fidelity models, with subpackages covering shared interfaces and heat and mass transfer formulations (*BaseClasses*), element-level building blocks (*Components*), equilibrium sorption models for different desiccant materials (*Materials*), complete device-level assemblies (*Equipment*), and verification cases (*Validation*). The *ReducedOrder* pack-

age provides computationally efficient steady-state alternatives, including effectiveness-based and curve-based formulations. The *Examples* package demonstrates the integration of models with different fidelity levels within a unified simulation framework.

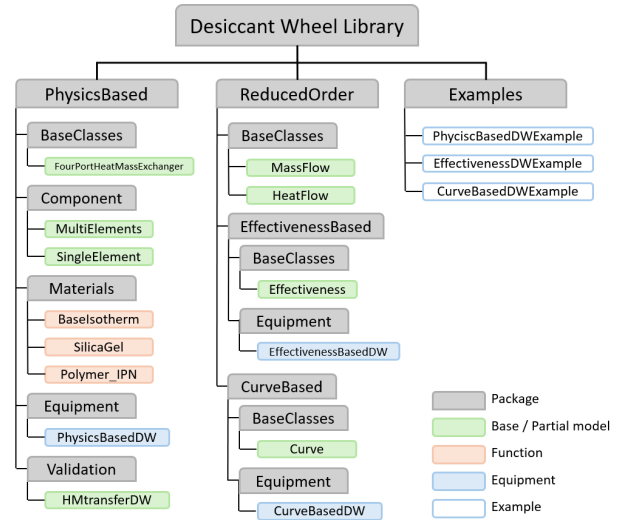


Figure 1. Structure of the desiccant wheel modeling library.

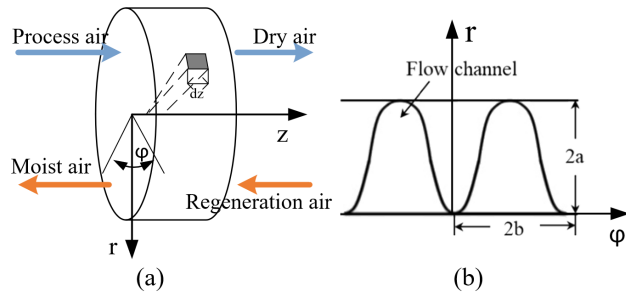


Figure 2. Desiccant-wheel geometry and honeycomb channel structure: (a) cylindrical coordinate system and differential axial control volume; (b) cross-section of the flow channels.

3 Dynamic Physics-Based Models

3.1 Governing Equations and Assumptions

A cylindrical coordinate system with axial (z), radial (r), and circumferential (ϕ) coordinates is used to describe the wheel geometry, as shown in Figure 2a.

The physics-based desiccant wheel models are formulated based on the coupled heat and mass transfer processes between the air streams and the desiccant material. A honeycomb channel structure (as shown in Figure 2b) is adopted to represent the desiccant matrix, as it is commonly used in practical desiccant wheels (Guan, M. Liu, et al. 2024). To ensure computational efficiency while preserving the dominant physical behavior, the following assumptions are adopted (Ge et al. 2008):

- Heat conduction and mass diffusion in the desiccant along the axial direction are negligible.

- No temperature or moisture gradients exist in the radial direction.
- Axial heat conduction and mass diffusion in the air stream are negligible.
- The inlet air conditions are spatially uniform.
- All channels are identical, and the channel walls are assumed to be adiabatic and impermeable.
- Air carry-over and pressure-driven leakage between the process and regeneration air streams are neglected for the current implementation.
- Thermophysical properties of dry air, water vapor, and desiccant are assumed constant, except for the air-side mass diffusion coefficient.
- Pressure drop along the air flow direction is neglected.

Under the assumptions described above, the governing equations are derived from energy and moisture conservation principles for both the air phase and the desiccant phase, following established formulations in the literature (Ge et al. 2008; Zhang et al. 2003).

The moisture conservation in the air is expressed as

$$A\rho_a \left(\frac{\partial Y_a}{\partial t} + u \frac{\partial Y_a}{\partial z} \right) = K_y P (Y_d - Y_a) \quad (1)$$

For Equation 1, the first and second terms on the left-hand side (LHS) represent moisture storage in the air and axial moisture transport by the airflow, respectively. The term on the right-hand side (RHS) represents convective mass transfer between the desiccant surface and air.

And the energy conservation in the air is given by

$$A\rho_a (c_{pa} + Y_a c_{pv}) \left(\frac{\partial T_a}{\partial t} + u \frac{\partial T_a}{\partial z} \right) = hP(T_d - T_a) + K_y P c_{pv} (Y_d - Y_a)(T_d - T_a) \quad (2)$$

In Equation 2, The first and second terms on the LHS represent energy storage in the moist air and axial energy transport by the airflow, respectively. The first term on the RHS represents sensible convective heat transfer between the air and solid desiccant, while the second represents the sensible heat transfer associated with moisture transfer between the solid desiccant and air.

The moisture conservation in the desiccant is written as

$$\frac{1}{2} f_d \frac{\partial W}{\partial t} = K_y P (Y_a - Y_d) \quad (3)$$

The term on the LHS of Equation 3 represents moisture storage in the desiccant, while the term on the RHS represents convective mass transfer between the air and desiccant.

And the energy conservation in the desiccant is expressed as

$$\left[\frac{1}{2} f_d (c_{pd} + W c_{pl}) + f_{sp} c_{psp} \right] \frac{\partial T_d}{\partial t} = hP(T_a - T_d) + K_y P (Y_a - Y_d) q_{st} + K_y P c_{pv} (Y_a - Y_d)(T_a - T_d) \quad (4)$$

The term on the LHS of Equation 4 represents thermal energy storage in the desiccant matrix. The three terms on the RHS represent sensible convective heat transfer, the heat associated with adsorption or desorption, and the sensible heat transfer associated with moisture transfer between the solid desiccant and air, respectively.

In these equations, Y_a and Y_d denote the humidity ratio of the air and the equilibrium humidity ratio at the desiccant surface, respectively, expressed in kg/kg_d, where the subscript d denotes dry air; W represents the moisture content of the desiccant, expressed in kg water/kg dry desiccant. T_a and T_d are the temperatures of the air and desiccant, and u is the air velocity. ρ_a is the air density, while c_{pa} , c_{pv} , c_{pd} , c_{pl} , and c_{psp} are the specific heat capacities of dry air, water vapor, desiccant, liquid water, and supporting material, respectively. q_{st} is the heat of adsorption, expressed in J/kg water. A and P represent the cross-sectional area and perimeter of the channel, respectively, while f_d and f_{sp} denote the specific masses per unit channel length of the desiccant and supporting material, respectively, expressed in kg/m. K_y and h are the mass and heat transfer coefficients. These values are used to calculate the heat transfer coefficient h and the mass transfer coefficient K_y according to Equation 5 and Equation 6.

$$h = \frac{\text{Nu} k_a P}{4A} \quad (5)$$

$$K_y = \frac{\rho_a \text{Sh} D_a P}{4A} \quad (6)$$

Following (Zhang et al. 2003), a heat and mass transfer analogy with $\text{Le} \approx 1$ is adopted, such that $\text{Sh} = \text{Nu}$. Here, k_a and D_a denote the thermal conductivity and mass diffusion coefficient in air, respectively, with D_a expressed in m²/s. D_a is calculated as (Zhang et al. 2003)

$$D_a = 2.302 \times 10^{-5} \frac{P_0}{P_{atm}} \left(\frac{T_a}{T_0} \right)^{1.81} \quad (7)$$

where $P_0 = 0.98 \times 10^5$ Pa and $T_0 = 256$ K are reference pressure and temperature, respectively.

Under fully developed laminar-flow conditions (De Antonellis et al. 2015), the Nusselt and Sherwood numbers depend only on the channel aspect ratio $\tau = a/b$ and are independent of air velocity, where a and b denote the half channel height and half channel width in the honeycomb structure, respectively (Figure 2b). Consequently, h remains constant for a fixed channel geometry, whereas K_y varies with the local air temperature through D_a ; neither coefficient depends explicitly on air velocity. The values adopted in this work are summarized in Table 1. These geometric parameters are directly defined in the Modelica implementation, allowing the transfer coefficients to be

consistently updated when the channel geometry is modified.

Table 1. Nusselt (Nu) and Sherwood (Sh) numbers as a function of channel aspect ratio $\tau = a/b$.

$\tau = a/b$	Nu = Sh
2.0	2.658
1.5	2.614
1.0	2.463
0.75	2.317
0.5	2.135

The equilibrium relationship between the desiccant moisture content and the air state is described by an adsorption isotherm, which can be generally expressed as

$$RH = f(W, T_d) \quad (8)$$

where RH denotes the relative humidity at the desiccant surface.

The humidity ratio corresponding to the equilibrium state is calculated as

$$Y_d = \frac{0.62198 P_v}{P_{atm} - P_v} = \frac{0.62198 RH P_{ws}}{P_{atm} - RH P_{ws}} \quad (9)$$

where $P_v = RH \cdot P_{ws}$ is the partial pressure of water vapor. P_{atm} is the atmospheric pressure.

The saturation vapor pressure is determined using the Antoine equation (Zhang et al. 2003):

$$P_{ws} = \exp \left(23.196 - \frac{3816.44}{T_d - 46.13} \right) \quad (10)$$

The governing equations are solved subject to inlet boundary conditions and initial conditions for both the process and regeneration air streams. For the process air stream, the inlet boundary at $z = 0$ is given by

$$T_a(0, t) = T_{pro,in}, \quad Y_a(0, t) = Y_{pro,in} \quad (11)$$

For the regeneration air stream, the inlet boundary is defined at the opposite end of the channel due to the counter-flow configuration, i.e.,

$$T_a(L, t) = T_{reg,in}, \quad Y_a(L, t) = Y_{reg,in} \quad (12)$$

The initial conditions are assumed to be spatially uniform for both the air and desiccant phases:

$$T_a(z, 0) = T_{init}, \quad T_d(z, 0) = T_{init}, \quad W(z, 0) = W_{init} \quad (13)$$

3.2 Model Implementation

3.2.1 Single-Element Implementation

The wheel matrix consists of a large number of identical honeycomb channels, as shown in Figure 2b. A representative channel is selected as the basic modeling unit, and the differential control volume dz shown in Figure 2a represents one axial segment of this channel. Under the assumptions introduced in Section 3.1, the governing equations are resolved only along the axial direction z . The channel of length L is therefore divided into n uniform finite volumes of length $\Delta z = L/n$, with all state variables defined at the cell centers, as shown in Figure 3.

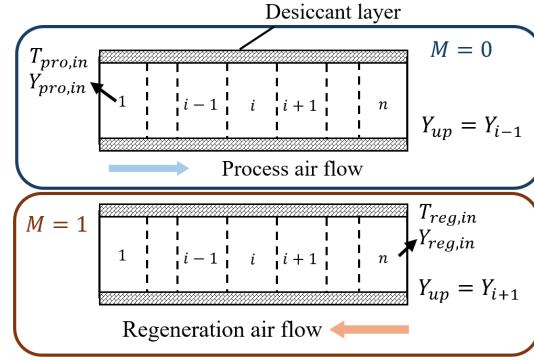


Figure 3. One-dimensional axial discretization of a representative honeycomb channel with flow-dependent upwind selection.

The convective terms are approximated using a first-order upwind scheme, where the upstream value for each cell i is dynamically selected based on the mode signal $M \in \{0, 1\}$, where $M = 0$ denotes process mode and $M = 1$ denotes regeneration mode:

$$Y_{a,up,i} = \begin{cases} Y_{a,0} & M = 0, i = 1 \\ Y_{a,i-1} & M = 0, i > 1 \\ Y_{a,n+1} & M = 1, i = n \\ Y_{a,i+1} & M = 1, i < n \end{cases} \quad (14)$$

and analogously for $T_{a,up,i}$. The effective flow velocity is likewise switched: $u_{eff} = u_p$ in process mode and $u_{eff} = u_r$ in regeneration mode. Figure 3 illustrates the discretization and the flow-direction-dependent upstream selection for both operating modes.

This unified formulation handles both flow directions within a single model structure, with M as the sole switching variable.

3.2.2 Multi-Element Implementation

The single-element model represents one channel element that operates in either process or regeneration mode at a given time and therefore provides only the outlet state associated with the active air stream. To represent the simultaneous operation of both air streams at the full-wheel level, multiple single-element instances are assembled along the circumferential direction, with elements present in the process and regeneration zones at the same time. The full desiccant wheel is modeled as an assembly of n_{Ele} channel elements, each implemented as a `DesiccantWheel.PhysicsBased.Component.SingleElement` instance sharing identical geometry and boundary inputs. The rotation of the wheel is represented not by continuous motion but by a discrete mode-switching mechanism governed by the wheel speed N (in revolutions per hour).

One full wheel revolution takes $t_{cycle} = 3600/N$ seconds, which is subdivided into n_{Ele} equal switching intervals of duration:

$$\Delta t_{sw} = \frac{t_{cycle}}{n_{Ele}} = \frac{3600}{N \cdot n_{Ele}} \quad (15)$$

At each switching event, a period counter $p \in$

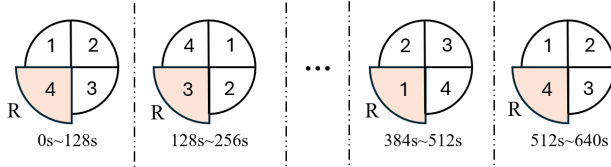


Figure 4. Mode assignment across switching steps for $n_{Ele} = 4$, $n_{PEle} = 3$. At each step, the elements advance by one circumferential position through the fixed process and regeneration zones, approximating continuous wheel rotation.

$\{0, 1, \dots, n_{Ele} - 1\}$ increments by one (wrapping to zero after $n_{Ele} - 1$). The logical position of element i is then:

$$pos_i = 1 + ((i - 1 + p) \bmod n_{Ele}) \quad (16)$$

Here, the modulo operation (mod) ensures cyclic indexing by constraining the position index to the range from 1 to n_{Ele} . Consequently, after reaching the last element (n_{Ele}), the index wraps around and returns to 1. Elements with $pos_i \leq n_{PEle}$ are assigned to process mode ($M = 0$), and the remainder to regeneration mode ($M = 1$):

$$M_i = \begin{cases} 0 & pos_i \leq n_{PEle} \\ 1 & pos_i > n_{PEle} \end{cases} \quad (17)$$

where n_{PEle} is the number of process-side elements. This shifting of logical positions at each interval approximates the continuous rotation of the wheel sectors through the process and regeneration zones. Figure 4 illustrates the mode assignment for an example case of $n_{Ele} = 4$ and $n_{PEle} = 3$. In this case, $N = 7$ rev/h is assumed. The process-to-regeneration area ratio $R_A = A_{pro}/A_{reg} = n_{PEle}/(n_{Ele} - n_{PEle})$ is determined entirely by the integer parameters n_{PEle} and n_{Ele} . Since both are integers, only discrete area ratios are achievable.

The process and regeneration outlet conditions are computed as weighted averages over the elements currently in each mode:

$$T_{out,pro} = \frac{\sum_{i=1}^{n_{PEle}} w_{pro,i} T_{out,i}}{n_{PEle}}, \quad T_{out,reg} = \frac{\sum_{i=1}^{n_{Ele} - n_{PEle}} w_{reg,i} T_{out,i}}{n_{Ele} - n_{PEle}} \quad (18)$$

where $w_{pro,i} = 1$ if element i is in process mode and 0 otherwise, and $w_{reg,i} = 1 - w_{pro,i}$. The humidity-ratio outputs $Y_{out,pro}$ and $Y_{out,reg}$ are computed analogously.

The wheel motor power consumption is modelled as a polynomial function of rotational speed (Ali Mandegari et al. 2014):

$$P_{mot} = \left(10^{-5} N^3 + 3 \times 10^{-4} N^2 + 2 \times 10^{-3} N + 8.2 \times 10^{-3} \right) \times 10^3 \text{ [W]} \quad (19)$$

where P_{mot} is the motor power consumption and N is in rev/h. This empirical fit captures the cubic dependence of motor power on rotational speed and is valid over the operating range $5 \leq N \leq 25$ rev/h (Ali Mandegari et al. 2014). Users may define their own power calculation method.

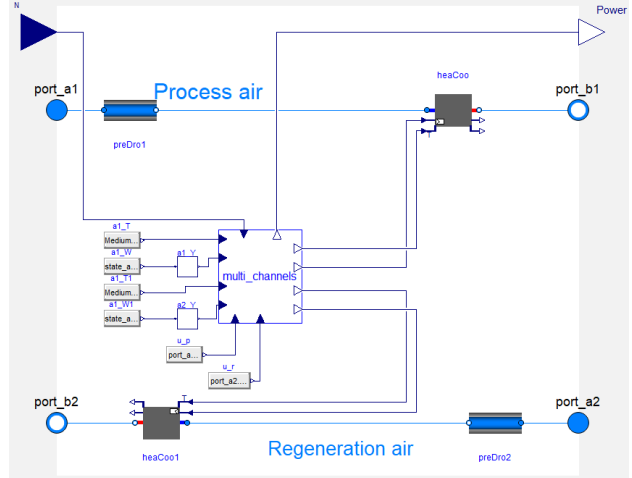


Figure 5. Diagram of physics-based DW model in the Dymola environment.

Table 2. Connectors of the physics-based DW model.

Type	Name	Description
FluidPort_a	port_a1	Fluid connector for the inlet of the process air stream
FluidPort_b	port_b1	Fluid connector for the outlet of the process air stream
FluidPort_a	port_a2	Fluid connector for the inlet of the regeneration air stream
FluidPort_b	port_b2	Fluid connector for the outlet of the regeneration air stream
RealInput	N	Rotational speed of the wheel [rev/h]
RealOutput	Power	Motor power consumption [W]

3.2.3 Model interface and parameters

Figure 5 shows the diagram of the developed physics-based DW model in the Dymola environment.

The desiccant wheel component extends `Buildings.Fluid.Interfaces.FourPortHeatMassExchanger`, which provides a four-port fluid interface for two separate air streams. The internal mixing volumes of the base class are replaced by `Buildings.Fluid.Interfaces.PrescribedOutlet` blocks on both the process and regeneration sides. The outlet temperature and humidity ratio computed by the `MultiElements` model are passed directly to the respective `PrescribedOutlet`, which imposes these values on the downstream fluid port without solving an additional energy or mass balance equation. Two pressure drop components account for the flow resistance on each side. Reviewer1: The model connectors are summarized in Table 2. The wheel rotational speed N is exposed as a `RealInput` and can be continuously adjusted during simulation rather than being limited to on/off control. The adjustable parameters, including the desiccant material isotherm and wheel and channel geometries, are shown in Figure 6 and listed in Table 3.

Parameters		
isothermType	DesiccantWheel.PhysicsBased.Materials.SilicaGel	Desiccant isotherm
a	0.00075	Channel half height
b	0.0015	Channel half width
L	0.2	Wheel thickness
D	0.5	Wheel diameter
nEle	6	Number of elements
nPEle	3	Number of process side elements

Figure 6. Adjustable parameters of the desiccant wheel model as shown in the Modelica parameter dialogue.

Table 3. Adjustable parameters of the desiccant wheel model.

Name	Type	Default	Description
isothermTypeFunction	SilicaGel		Desiccant isotherm
a	SI.Length	0.0015 m	Channel half height
b	SI.Length	0.0015 m	Channel half width
L	SI.Length	0.2 m	Wheel thickness
D	SI.Length	0.5 m	Wheel diameter
nEle	Integer	6	Number of elements
nPEle	Integer	3	Number of process-side elements

4 Reduced-Order Models

Two reduced-order models are implemented as alternatives to the physics-based model, targeting applications where computational efficiency is prioritized over spatial resolution.

4.1 Effectiveness-based model

The effectiveness-based model follows the analogy method proposed by Jurinak (Jurinak 1982), also known as the effectiveness method, which has been adopted in TRNSYS (Jani et al. 2020) and other studies (Panaras et al. 2010). Two characteristic potentials are defined as:

$$F_{1,i} = \frac{-2865}{T_i^{1.49}} + 4.344 Y_i^{0.8624}, F_{2,i} = \frac{T_i^{1.49}}{6360} - 1.127 Y_i^{0.07969} \quad (20)$$

where $i = 1, 2, 3, 4$ denotes the four state points of the wheel (process inlet, process outlet, regeneration inlet, regeneration outlet, respectively). The effectiveness factors are defined as:

$$\varepsilon_{F_1} = \frac{F_{1,2} - F_{1,1}}{F_{1,3} - F_{1,1}}, \quad \varepsilon_{F_2} = \frac{F_{2,2} - F_{2,1}}{F_{2,3} - F_{2,1}} \quad (21)$$

Given the inlet conditions at points 1 (process inlet) and 3 (regeneration inlet), and user-specified effectiveness factors ε_{F_1} and ε_{F_2} , the process outlet state (T_{PO} , Y_{PO}) is solved implicitly.

The effectiveness factors are constant and typically calibrated from experimental data. Reference values for different performance levels can be found in existing literature (Guan, M. Liu, et al. 2024).

4.2 Curve-based model

The curve-based model follows the formulation in EnergyPlus (U.S. Department of Energy 2024) for the air-to-air balanced-flow desiccant heat exchanger. The regeneration outlet temperature and humidity ratio are computed directly from the inlet conditions of both air streams via polynomial regressions:

$$T_{r,out} = B_1 + B_2 Y_{RI} + B_3 T_{RI} + B_4 \frac{Y_{RI}}{T_{RI}} + B_5 Y_{PI} + B_6 T_{PI} + B_7 \frac{Y_{PI}}{T_{PI}} + B_8 V_R \quad (22)$$

$$Y_{r,out} = C_1 + C_2 Y_{RI} + C_3 T_{RI} + C_4 \frac{Y_{RI}}{T_{RI}} + C_5 Y_{PI} + C_6 T_{PI} + C_7 \frac{Y_{PI}}{T_{PI}} + C_8 V_R \quad (23)$$

where T_{RI} and Y_{RI} are the regeneration inlet temperature and humidity ratio; T_{PI} and Y_{PI} are the process inlet temperature and humidity ratio; and V_R is the regeneration face velocity. All inputs are clamped to their valid operating ranges prior to evaluations, and the outputs are similarly bounded. The regression coefficients are available from the EnergyPlus dataset (U.S. Department of Energy 2024).

4.3 Model Interface

Both models share a common base-class structure: the predicted outlet air state is converted into heat and moisture source terms by `HeatFlow` and `MassFlow`, which are then injected into the fluid volumes on each side.

MassFlow. Given the inlet and outlet humidity ratios and the total mass flow rate, the moisture exchange rate transferred to the fluid volume is given by the following equation:

$$\dot{m}_w = \dot{m} \cdot \frac{1}{1 + Y_{in}} (Y_{out} - Y_{in}) \quad (24)$$

The opposite side receives $-\dot{m}_w$ by mass conservation between the process and regeneration sides. Depending on the model formulation, the effectiveness model computes the process side while the regeneration side acts as the opposite side, whereas the curve-based model computes the regeneration side with the process side acting as the opposite side. A Boolean signal activates or deactivates the exchange.

HeatFlow. The specific enthalpy of the outlet air is computed at the inlet pressure from the predicted outlet temperature T_{out} and the corresponding water-vapor mass fraction X_{out} , converted from the humidity ratio Y_{out} , using `Medium.specificEnthalpy_pTX`. The heat flow rate injected into the fluid volume is calculated using the following equation:

$$\dot{Q} = (h_{in} - h_{out}) \dot{m} \quad (25)$$

and the reverse side receives $-\dot{Q}$. The same Boolean activation signal as in `MassFlow` is used.

Both reduced-order models expose the same four fluid connectors as the physics-based model (`port_a1`, `port_b1`, `port_a2`, `port_b2`). An additional Boolean input `u` is provided to represent the operational state of the wheel. When `u = false`, the heat and moisture source terms are set to zero.

5 Validation and Demonstration

5.1 Validation of Physics-Based Model

To validate the implementation of the single-element model, the simulation results are compared with the reference results reported in (Zhang et al. 2003). Figure 7 presents the transient temperature and humidity ratio profiles over one complete rotation cycle, including both regeneration and dehumidification phases. The simulation inputs and operating conditions used for this validation case are summarized in Table 4, including the inlet air conditions, wheel rotational speed, and sector configuration. The model used for validation corresponds to the single-element implementation in the library (i.e., `DesiccantWheel.PhysicsBased.Validation.HMtransferDW`).

As shown in Figure 7, the implemented model is able to reproduce the key dynamic behaviors of the desiccant wheel, including the rapid temperature rise at the sector transition and the corresponding variation in humidity ratio. The overall trends and magnitudes are in good agreement with the reference results. Quantitatively, the root mean square error (RMSE) of temperature is approximately 0.76°C with a coefficient of determination (R^2) of 0.991, while the RMSE of humidity ratio is about 0.0016 kg/kg_d with an R^2 value of 0.972. These results indicate that the implementation accurately captures both thermal and moisture transport processes.

This comparison verifies the correctness of the single-element implementation rather than evaluating the performance of different models. The full-wheel multi-element model has been validated separately in (Guan, Yang, et al. 2026). To avoid duplicating previously reported results, the corresponding validation details are not repeated here.

5.2 Demonstration of Physics-Based Model

The capabilities of the physics-based models are demonstrated through a series of representative cases, as summarized in Table 5. These cases are designed to illus-

Table 4. Simulation conditions for validation of the single-element model.

Parameter	Symbol	Value
Half channel height	a	0.0015 m
Half channel width	b	0.0015 m
Desiccant wheel thickness	L	0.2 m
Process-to-regeneration area ratio	R_A	3/1
Regeneration air velocity	$V_{r,in}$	1.5 m/s
Process air velocity	$V_{p,in}$	2 m/s
Inlet process air temperature	$T_{p,in}$	34.3°C
Inlet process air humidity ratio	$Y_{p,in}$	0.019332 kg/kg_d
Inlet regeneration air temperature	$T_{r,in}$	100°C
Inlet regeneration air humidity ratio	$Y_{r,in}$	0.019332 kg/kg_d
Cycle time	t_{cyc}	512 s

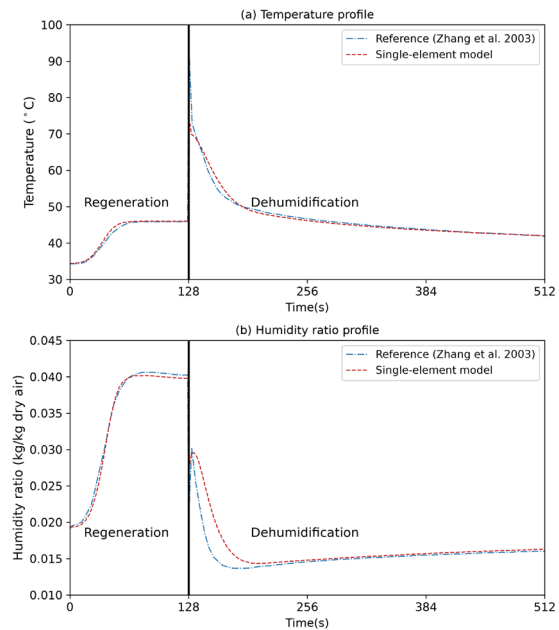


Figure 7. Validation of the single-element model implementation against the reference results of Zhang et al. (2003).

trate different aspects of the modeling framework, including implementation correctness, discretization resolution, flexibility in material and geometry configuration, and dynamic response under time-varying operating conditions. All simulation cases are run for 3600 seconds.

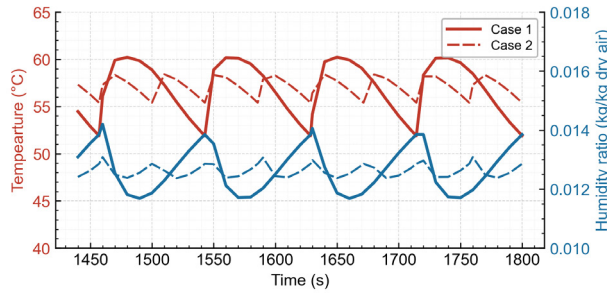
Case 1 serves as a benchmark to verify the baseline behavior of the implemented model under fixed operating conditions, and the results provide a baseline for comparisons with subsequent cases.

5.2.1 Full-Wheel Representation

The multi-element model represents the complete desiccant wheel by distributing multiple channel elements across the process and regeneration zones. Cases 1 and 2 use 6 and 12 elements, respectively. Both configurations represent the complete wheel, while their different element numbers provide different circumferential discretization resolutions. Figure 8 compares their pre-

Table 5. Summary of model parameters for different demonstration cases.

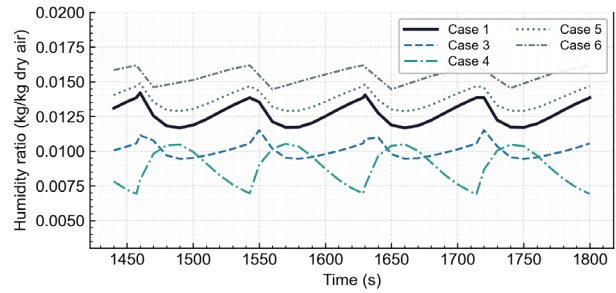
Parameter	Symbol	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7
Number of elements (-)	n_{Ele}	6	12	6	6	6	6	6
Half channel height (m)	a	0.0015	0.0015	0.00075	0.0015	0.0015	0.0015	0.0015
Half channel width (m)	b	0.0015	0.0015	0.0015	0.0015	0.0015	0.0015	0.0015
Desiccant wheel thickness (m)	L	0.2	0.2	0.2	0.5	0.2	0.2	0.2
Desiccant wheel diameter (m)	D	0.5	0.5	0.5	0.7	0.5	0.5	0.5
Process-to-regeneration area ratio (-)	R_A	1	1	1	1	2	1	1
Material (-)	–	Silica gel	Silica gel	Silica gel	Silica gel	Silica gel	Polymer (Zeng et al. 2021)	Silica gel
Rotational speed (rev/h)	N	7	7	7	7	7	7	7 → 10 (time-varying)
Regeneration air flow rate (kg/s)	$m_{r,\text{in}}$	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Process air flow rate (kg/s)	$m_{p,\text{in}}$	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Inlet process air temperature (°C)	$T_{p,\text{in}}$	34.3	34.3	34.3	34.3	34.3	34.3	34.3
Inlet process air humidity ratio (kg/kg _d)	$Y_{p,\text{in}}$	0.019332	0.019332	0.019332	0.019332	0.019332	0.019332	0.019332
Inlet regeneration air temperature (°C)	$T_{r,\text{in}}$	100	100	100	100	100	100	100 → 80 (time-varying)
Inlet regeneration air humidity ratio (kg/kg _d)	$Y_{r,\text{in}}$	0.019332	0.019332	0.019332	0.019332	0.019332	0.019332	0.019332

**Figure 8.** Outlet temperature and humidity ratio of Case 1 and Case 2 during steady-state operation.

dicted outlet temperature (red lines) and humidity ratio (blue lines) over a representative steady-state period. The 12-element configuration resolves the switching behavior with finer angular resolution, whereas the 6-element configuration produces smoother periodic profiles. This comparison illustrates the influence of element number on the predicted device-level outlet states.

5.2.2 Material and Geometry Flexibility

Cases 3–6 are designed to demonstrate the flexibility of current models in handling different material properties and geometric configurations. Figure 9 presents the outlet humidity ratio for Cases 1 and 3–6, where key geometric and material parameters are varied. The results show a clear spread in humidity ratio across cases, confirming that geometric parameters such as channel dimensions and wheel size directly affect moisture removal capacity. Case 6, which uses a different desiccant material, exhibits a notably different humidity ratio profile, consistent with

**Figure 9.** Outlet humidity ratio of Cases 1 and 3–6 during steady-state operation.

its distinct adsorption equilibrium characteristics. Importantly, all cases are generated by parameter modifications alone, without structural changes to the model, demonstrating the flexibility of the model.

5.2.3 Dynamic Operation

Case 7 demonstrates the capability of the model to simulate dynamic operation under time-varying conditions, where the wheel rotational speed and regeneration inlet air temperature are varied during the simulation.

Figure 10 illustrates the transient response of the outlet temperature and humidity ratio for Case 1 and Case 7. The dashed vertical line at $t = 1536$ s marks the point at which the wheel rotational speed and regeneration inlet air temperature are varied. Prior to this point, both cases exhibit similar periodic behavior under identical conditions. Following the condition change, Case 7 shows a clear shift in both the amplitude and mean level of the temperature and humidity ratio oscillations, demonstrat-

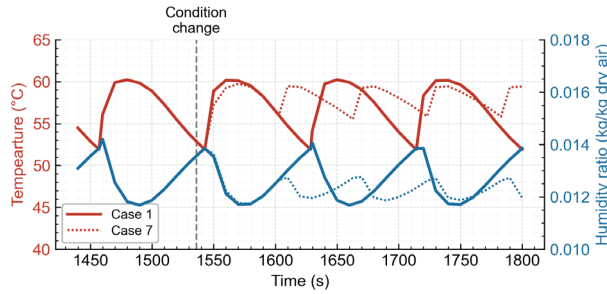


Figure 10. Transient outlet temperature and humidity ratio of Case 1 and Case 7. The dashed vertical line indicates the point of operating condition change at $t = 1536$ s.

ing the model's capability to capture the coupled transient dynamics of heat and mass transfer under time-varying inputs. This dynamic responsiveness is essential for control strategy development and system-level simulation.

6 Discussion

Model selection. The choice of model type involves a trade-off between computational efficiency and physical fidelity. As shown in Table 6, the physics-based model requires significantly longer CPU time than the reduced-order models, with runtime increasing with the number of elements. All simulations were performed on a system equipped with a 12th Gen Intel Core i7-12700 CPU and 32 GB RAM. Despite this, it provides detailed internal state information and supports flexible geometry and material configurations, making it suitable for component-level design and analysis. For computationally demanding cases, high-performance computing can be employed to reduce simulation time. The reduced-order models are orders of magnitude faster, making them well-suited for long-time system-level simulations. However, their applicability is limited to the conditions under which they were fitted, and cannot be directly transferred to different application domains (Guan, M. Liu, et al. 2024). The choice between model types should therefore be guided by the specific requirements of the application in terms of fidelity, computational budget, and operating conditions.

Applicability to Energy Recovery Wheels. Although the proposed library is developed specifically for desiccant wheels, the underlying modeling method is directly applicable to energy recovery wheels (ERWs), which are widely used in HVAC systems. For total energy recovery wheels, the governing heat and mass transfer mechanisms are essentially identical between the two devices; the primary distinctions are that ERWs operate at higher rotation speeds and do not require an external regeneration heat source (Nóbrega and Brum 2009). These differences can be accommodated through parameter adjustment without any structural modification to the model. Sensible heat recovery wheels, in contrast, require a heat-transfer-only formulation. The physics-based model can be adapted by preserving its energy balance, circumferential discretization, and rotation mechanism while removing the mois-

ture balance and sorption equations. The reduced-order models can use the existing interface and architecture after latent transfer is disabled and their sensible effectiveness or performance curves are recalibrated using data for the target wheel. This extends the applicability of the proposed library beyond desiccant dehumidification, offering a potentially useful tool for the broader class of rotary heat and mass exchanger modeling in building energy simulation and HVAC system design.

Table 6. CPU time for different model types.

Model Type	Case	CPU time (s)
Physics-based model	1	1.689
	2	16.487
Effectiveness-based model	–	0.003733731
Curve-based model	–	0.003032741

7 Conclusion

This paper presents a modular desiccant wheel modeling library developed in Modelica that integrates dynamic physics-based models and steady-state reduced-order models within a unified framework. The proposed approach addresses key limitations of existing models by providing open-source implementations, enabling acausal system integration, and incorporating a multi-element formulation that simultaneously calculates the process- and regeneration-side outlet states.

The physics-based models support configurable geometry and desiccant materials, making them suitable for detailed analysis, while the reduced-order models offer computational efficiency for system-level simulations. By organizing all models within a plug-and-play architecture, the library allows flexible selection of model fidelity without modifying system configurations.

Comparative results highlight the trade-offs between accuracy and computational cost and demonstrate the applicability of different models across use cases. The proposed library provides a scalable and extensible platform for desiccant wheel modeling in both building and industrial applications.

A limitation of the present work is that the current validation is limited to comparisons with reference results. Future work will focus on validation against experimental data, as well as further validation of the model for systems involving multiple materials.

Software availability. The Modelica library developed in this study is publicly available as open-source software at <https://github.com/BE-HVACR/DesiccantWheel> under the MIT License.

Acknowledgements

This work was partially supported by the U.S. Department of Energy under award number DE-EE0010850.

References

- Abdelgaied, Mohamed, Mohamed A. Saber, Mohamed Mahgoub Bassuoni, and Ahmed M. Khaira (2023-01-18). "Adsorption Air Conditioning: A Comprehensive Review in Desiccant Materials, System Progress, and Recent Studies on Different Configurations of Hybrid Solid Desiccant Air Conditioning Systems". In: *Environmental Science and Pollution Research* 30.11, pp. 28344–28372. ISSN: 1614-7499. DOI: 10.1007/s11356-023-25209-z.
- Ali Mandegari, Mohsen, Hassan Pahlavanzadeh, and Somayeh Farzad (2014-09). "Energy Approach Analysis of Desiccant Wheel Operation". In: *Energy Systems* 5.3, pp. 551–569. ISSN: 1868-3967, 1868-3975. DOI: 10.1007/s12667-013-0115-z.
- De Antonellis, Stefano, Manuel Intini, and Cesare Maria Joppolo (2015-09). "Desiccant Wheels Effectiveness Parameters: Correlations Based on Experimental Data". In: *Energy and Buildings* 103, pp. 296–306. ISSN: 03787788. DOI: 10.1016/j.enbuild.2015.06.041.
- Ge, T.S., Y. Li, R.Z. Wang, and Y.J. Dai (2008-08). "A Review of the Mathematical Models for Predicting Rotary Desiccant Wheel". In: *Renewable and Sustainable Energy Reviews* 12.6, pp. 1485–1528. ISSN: 13640321. DOI: 10.1016/j.rser.2007.01.012.
- Gherzi, Djamel Eddine, Nour Elislam Mougari, Khaled Loubar, Meriem Amoura, and Umberto Desideri (2026-01). "A Machine Learning and Particle Swarm Optimization Approach for Desiccant Wheel Modeling and Performance Prediction". In: *Applied Thermal Engineering* 284, p. 129084. ISSN: 13594311. DOI: 10.1016/j.applthermaleng.2025.129084.
- Guan, Hang, Mingzhe Liu, Zhiyao Yang, and Zheng O'Neill (2024). "Comparison of Different Desiccant Wheel Modeling Methods: A Review and Case Study". In: *Proceedings of the ASHRAE Annual Conference*. DOI: 10.63044/s24gua23.
- Guan, Hang, Mingzhe Liu, Zhiyao Yang, and Zheng O'Neill (2025). "Optimizing the Geometric Design of Desiccant Wheels for Drying Controllability and Desiccant Use". In: *Proceedings of the ASHRAE Annual Conference*. DOI: 10.63044/w25gua67.
- Guan, Hang, Zhiyao Yang, Mingzhe Liu, and Zheng O'Neill (2026-09). "Application-Specific Performance Assessment of Desiccant Wheel Design and Operational Parameters for Industrial Drying and HVAC Dehumidification". In: *Applied Thermal Engineering* 304, p. 132767. ISSN: 1359-4311. DOI: 10.1016/j.applthermaleng.2026.132767.
- Habib, Muhammad Farhan, Muzaffar Ali, Nadeem Ahmed Sheikh, Abdul Waheed Badar, and Sajid Mehmood (2020-07). "Building Thermal Load Management through Integration of Solar Assisted Absorption and Desiccant Air Conditioning Systems: A Model-Based Simulation-Optimization Approach". In: *Journal of Building Engineering* 30, p. 101279. ISSN: 23527102. DOI: 10.1016/j.jobee.2020.101279.
- Jani, D. B., Kiran Bhabhor, Mohsin Dadi, Sachindra Doshi, P. V. Jotaniya, Harish Ravat, and Kumar Bhatt (2020-06). "A Review on Use of TRNSYS as Simulation Tool in Performance Prediction of Desiccant Cooling Cycle". In: *Journal of Thermal Analysis and Calorimetry* 140.5, pp. 2011–2031. ISSN: 1388-6150, 1588-2926. DOI: 10.1007/s10973-019-08968-1.
- Jurinak, Jeff Jerome (1982). "Open Cycle Solid Desiccant Cooling—Component Models and System Simulations". PhD thesis. The University of Wisconsin-Madison.
- Leva, Alberto, Erica Zavaglio, and Marco Bonvini (2014). "Dynamic Modelling of Desiccant Wheels for the Design of Energy-Efficient Air Handling Units". In: *IFAC Proceedings Volumes* 47.3, pp. 11629–11634. ISSN: 14746670. DOI: 10.3182/20140824-6-ZA-1003.01409.
- Liu, Haoran, Yixiang Huang, Shaochen Tian, Lei Huang, Shangao Li, Qinbao Wang, and Xing Su (2025-05). "Operational Performance of Heat Pump Desiccant Wheel System in Low Humidity Industrial Environment: On-site Measurements and Model Based Optimization". In: *Energy* 322, p. 135543. ISSN: 03605442. DOI: 10.1016/j.energy.2025.135543.
- Liu, Mingzhe et al. (2026-06). "Multidisciplinary Modeling and Evaluation of Hybrid Hydrogel Desiccant Coated Heat Exchanger". In: *Applied Thermal Engineering* 298, p. 130927. ISSN: 1359-4311. DOI: 10.1016/j.applthermaleng.2026.130927.
- Ma, Yunlong, Suvash C. Saha, Wendy Miller, and Lisa Guan (2017-06-25). "Parametric Analysis of Design Parameter Effects on the Performance of a Solar Desiccant Evaporative Cooling System in Brisbane, Australia". In: *Energies* 10.7, p. 849. ISSN: 1996-1073. DOI: 10.3390/en10070849.
- Nóbrega, C. E. L. and N. C. L. Brum (2009-12). "Modeling and Simulation of Heat and Enthalpy Recovery Wheels". In: *Energy*. ECOS 2007 34.12, pp. 2063–2068. ISSN: 0360-5442. DOI: 10.1016/j.energy.2008.08.016.
- Panaras, G., E. Mathioulakis, V. Belessiotis, and N. Kyriakis (2010-10). "Experimental Validation of a Simplified Approach for a Desiccant Wheel Model". In: *Energy and Buildings* 42.10, pp. 1719–1725. ISSN: 03787788. DOI: 10.1016/j.enbuild.2010.05.006.
- Park, Myeong Hyeon, Jun Yeob Chung, Seong Ho Hong, Hyun Ho Shin, Dongchan Lee, and Yongchan Kim (2023-12). "Optimized Geometric Designs of Desiccant Wheels with Metal-Organic Frameworks Considering Dehumidification Capacity and Energy". In: *Energy* 284, p. 129369. ISSN: 03605442. DOI: 10.1016/j.energy.2023.129369.
- Speerforck, Arne, Jiazhen Ling, Vikrant Aute, Reinhard Radermacher, and Gerhard Schmitz (2017-12). "Modeling and Simulation of a Desiccant Assisted Solar and Geothermal Air Conditioning System". In: *Energy* 141, pp. 2321–2336. ISSN: 03605442. DOI: 10.1016/j.energy.2017.11.151.
- U.S. Department of Energy (2024). *EnergyPlus Engineering Reference*. URL: <https://energyplus.net/> (visited on 2024-03-31).
- Wetter, Michael, Wangda Zuo, Thierry S. Noudui, and Xiufeng Pang (2014). "Modelica Buildings library". In: *Journal of Building Performance Simulation* 7.4, pp. 253–270. DOI: 10.1080/19401493.2013.765506. URL: <https://doi.org/10.1080/19401493.2013.765506>.
- Zeng, Yi, Jason Woods, and Shuang Cui (2021-09). "The Energy Saving Potential of Thermo-Responsive Desiccants for Air Dehumidification". In: *Energy Conversion and Management* 244, p. 114520. ISSN: 01968904. DOI: 10.1016/j.enconman.2021.114520.
- Zhang, X.J., Y.J. Dai, and R.Z. Wang (2003-06). "A Simulation Study of Heat and Mass Transfer in a Honeycombed Rotary Desiccant Dehumidifier". In: *Applied Thermal Engineering* 23.8, pp. 989–1003. ISSN: 13594311. DOI: 10.1016/S1359-4311(03)00047-4.

A Templated Modelica Model for Decoupling Physical System and Control in Resilient Community Energy Systems

Juan Diego Flores Garcia¹ Tahiya Hossain¹ Viswanathan Ganesh¹ Wangda Zuo^{1,2}

¹Architectural Engineering, Penn State University, United States of America,
{jdf5775,tzh5647,vmg5298,wangda.zuo}@psu.edu

²National Laboratory of the Rockies, Golden, CO, USA

Abstract

As natural disasters increasingly impact the electrical infrastructure in the United States and cause widespread, prolonged power loss, microgrids capable of operating in island mode have emerged as a critical solution for sustaining community resilience. While Modelica models can accurately predict high-fidelity dynamic system behavior, the manual effort required to build and configure multiple distinct architectures from scratch is too time-consuming for practical design exploration. To address this gap, this paper presents a set of high-fidelity, pre-defined Modelica system templates for islanded capable Net Zero Energy Communities (NSEC) microgrids. The proposed model uses a parameterized 1D array to seamlessly switch among 12 distinct system topologies, automatically updating the corresponding baseline controller. Within these templates, the physical system is structurally decoupled from the control logic, allowing researchers to easily swap and test different control algorithms for any given system topology. To demonstrate the use of this modular framework, we conducted a case study that performed an exhaustive search of equipment-sizing for a simulated community in Tampa, Florida. The study evaluated each system topology against key performance metrics: capital cost, emissions, and fuel consumption. The results proved that the templating mechanism functions as intended, successfully instantiating the correct system topology and control logic for each iteration. Rather than isolating a singular best topology, the rapid evaluation efficiently mapped the techno-economic design space to identify a Pareto-optimal frontier, ultimately presenting seven highly viable configurations for user selection. Ultimately, the successful generation and evaluation of these multiple topologies establishes a time-efficient, high-fidelity foundation for future design workflows. This enables researchers to seamlessly design optimal equipment sizing and test advanced control algorithms for an islanded NSEC.

Keywords: microgrid, resilience, template models, design

1 Introduction

Much of the United States' electrical grid relies on a centralized infrastructure primarily constructed in the 1950s

and 1960s. With an average age of roughly 40 years, the system suffers from a significant resilience gap due to aging components such as transmission lines and transformers that are nearing or exceeding their intended lifespans (Idima et al. 2023). A lack of comprehensive upgrades to the aging electrical grid has created a severe resilience gap, leaving the system fundamentally unequipped to withstand extreme weather events like hurricanes, wildfires, and severe winter storms. Because of this structural fragility, weather-related failures are the primary driver of large-scale outages, resulting in an average of 520 million customer-hours without power annually across the United States (Do et al. 2023). The real-world consequences of this aging infrastructure are severe: Hurricane Maria (2017) devastated Puerto Rico's fragile grid, resulting in a grueling 11-month restoration effort to reconnect residents to power (Campbell 2018). Similarly, extreme weather threats such as widespread wildfires in California frequently force utility providers to disconnect power to millions of residents for consecutive days to prevent grid-sparked ignitions (Zuo et al. 2025). Moreover, the Texas grid failure during the 2021 Winter Storm left 4.5 million Texans freezing without electricity and triggered unprecedented spikes in residential utility bills (Cramton 2022). Furthermore, grid vulnerabilities extend beyond extreme weather events. Under normal operating conditions, aging transmission infrastructure often lacks the capacity to meet the increasing peak energy demands of growing communities or geographically isolated areas. In such scenarios, microgrids can disconnect from the bulk grid to self-generate power and shave peak loads, thereby preventing localized blackouts (Ganesh et al. 2022). Ultimately, this undeniable susceptibility to both extreme weather and everyday capacity limits highlights a growing national trend of prolonged outages and underscores the urgent need for more resilient, localized energy systems.

Microgrid energy systems capable of autonomous islanding have emerged as a vital solution to sustain critical power during prolonged outages, ensuring community resilience while minimizing severe economic impacts (Balducci et al. 2020; Energy 2017). Traditionally, emergency backup power has relied exclusively on inefficient, high-emission diesel generators (DG). However, modern com-

munity microgrids leverage a hybrid architecture of renewable distributed energy resources (DERs), such as solar photovoltaics (PV) coupled with battery energy storage systems (BESS). This hybrid approach not only provides robust, localized power during catastrophic grid failures but also aligns with broader decarbonization goals by minimizing fossil fuel consumption during islanded operations.

However, designing a resilient microgrid with multiple DERs is a highly complex engineering challenge due to the vast number of possible equipment combinations, sizing and control strategies (Yan et al. 2017). While various hardware configurations can successfully sustain the critical load during an outage, they often yield drastically different capital costs, fuel consumption rates, and emission profiles. Furthermore, generic controllers are insufficient for these diverse topologies; to guarantee system stability and prevent secondary blackouts, each unique physical configuration requires a dedicated, rigorously tailored control strategy. Therefore, evaluating and optimizing these systems requires a comprehensive approach that accounts for both the physical equipment mix and the supervisory logic simultaneously.

Existing microgrid optimization tools, such as the Hybrid Optimization of Multiple Energy Resources (HOMER) and the Renewable Energy Integration and Optimization (REopt) model, are widely utilized for baseline techno-economic planning but present distinct limitations for advanced system development. Primarily, their reliance on Mixed-Integer Linear Programming (MILP) with piecewise-linear approximations effectively captures discrete equipment coupling and steady-state dispatch, resolving fast transient dynamics requires fine time-discretization that can become computationally demanding. Additionally, these platforms operate with simplified, static control logic. This closed architecture restricts engineers from implementing or evaluating the highly rigorous supervisory controllers necessary for optimizing power dispatch under dynamic, high-stress conditions. To capture the non-linear dynamics that standard planning tools miss, recent literature has used high-fidelity simulation environments, such as Modelica, to model microgrid performance. For instance, Castro et al. (2023) demonstrated the efficacy of the Modelica OpenIPSL library in capturing the electromechanical transients of a specific university campus microgrid (Fachini et al. 2023). Similarly, Zuo et al. (2025) used Modelica-based frameworks to perform high-fidelity, multi-timescale dynamic simulations of an offshore energy island, demonstrating the necessity of detailed component-level modeling for integrated renewable systems. While these platforms provide exceptional transient resolution, they impose a significant architectural constraint: the supervisory control logic in custom microgrid models is typically rigidly coupled to the physical configuration of the equipment. This inherent lack of modularity dictates that any attempt to alter the equipment configuration or substitute an advanced control al-

gorithm requires major manual reengineering and remapping (Dosa et al. 2025).

To address these critical gaps, this paper proposes a reproducible, unified Modelica modeling template specifically engineered for net-zero energy communities (NSEC) with islanded capability. Evaluating diverse microgrid dispatch strategies traditionally requires manually rewiring the model's graphical layer. To overcome this, the proposed framework implements a parameterized 1D array switching mechanism that completely decouples the physical distributed energy assets from the supervisory control logic. This advanced parameterization enables users to automatically instantiate and simulate 12 distinct system topologies without manual topological reconfiguration. The remainder of this paper is organized as follows: the modular modeling methodology and software implementation, followed by a community resilience case study, an analysis of the simulation results, and conclusion.

2 Methodology

To facilitate the rapid and reproducible evaluation of community energy systems, this work introduces a modular modeling framework developed using the Modelica Buildings Library (Wetter et al. 2014). The core methodological approach involves constructing hierarchical templates that allow users to seamlessly transition between different system topologies and operational strategies. This flexibility was achieved by dividing the architecture into two interoperable sub-models. First, a dedicated controller model was designed to govern the dispatch logic, specifically managing the power flow of the DG and the BESS. Second, a comprehensive Test Bed model was developed to encompass the physical system of the dispatchable units, renewable energy generation, and the community load. This modular template work is shown in Figure 1

Communication between these sub-models is facilitated by standardized information buses. The physical Test Bed transmits real-time state data to the control model, specifically the instantaneous power outputs from the renewable generation, the DG, and the BESS. In response, the controller evaluates the system state and transmits three primary dispatch signals back to the Test Bed: a binary on/off operational command for the DG, a requested power output setpoint for the DG, and a directional power command dictating the charge or discharge rate of the BESS.

2.1 System Description

The physical system model structurally decouples the control logic from the energy assets, which include PV, wind turbines, BESS, and DG. This architecture defines 12 distinct community energy systems topologies, as detailed in Table 1. To guarantee system resilience during islanded events, every configuration is designed to include at least one dispatchable unit (either a DG or BESS).

The physical system is driven by a set of boundary conditions that simulate a realistic operational environment. The primary input is a community electrical load pro-

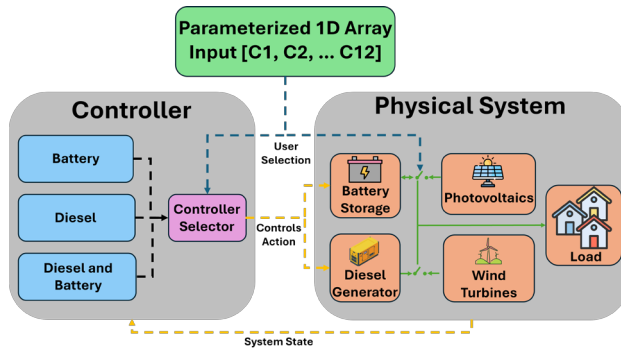


Figure 1. Template diagram

Table 1. Microgrid Configurations and Active Control Strategies

Config ID	Included Components	Controlled Asset(s)
1	DG only	DG
2	BESS only	BESS
3	DG + Wind	DG
4	DG + BESS	DG + BESS
5	DG + PV	DG
6	Wind + BESS	BESS
7	BESS + PV	BESS
8	DG + Wind + BESS	DG + BESS
9	DG + Wind + PV	DG
10	DG + BESS + PV	DG + BESS
11	Wind + BESS + PV	BESS
12	DG + Wind + BESS + PV	DG + BESS

file, which establishes the strict power demand required to maintain resilience. Furthermore, site-specific meteorological data such as solar irradiance and ambient wind speed are utilized to dictate the transient, intermittent generation of the renewable assets. This integration of real-world weather and load data ensures the system model accurately captures complex transient behaviors, requiring the control logic to continuously balance a fluctuating renewable supply against the community's critical demand.

2.2 Controls

To ensure seamless operation without necessitating manual model reconfiguration, a parameterized 1D array switching mechanism automatically links the active physical system to its corresponding controller. This mechanism dynamically selects among BESS-only, DG-only, or combined DG-BESS control strategies, depending on the active equipment configuration and the available dispatchable units.

The DG-only controller employs a load-following threshold strategy, activating whenever the building load exceeds 80% of the active generation capacity. To mitigate mechanical wear from rapid cycling, the generator deactivates only when total generation exceeds 130% of the load, provided this condition is sustained for a mandatory 30-minute waiting period. The state logic for this controller is illustrated in Figure 2.

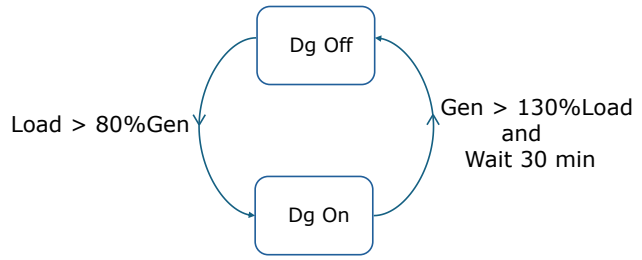


Figure 2. DG-only controller

The BESS-only controller manages transitions between idle, charging, and discharging states to strictly track the net electrical load. By continuously evaluating whether PV generation exceeds the building demand, this controller prioritizes maximizing renewable self-consumption while maintaining the battery's state of charge (SOC) within predefined safety limits. This behavior is shown in Figure 3.

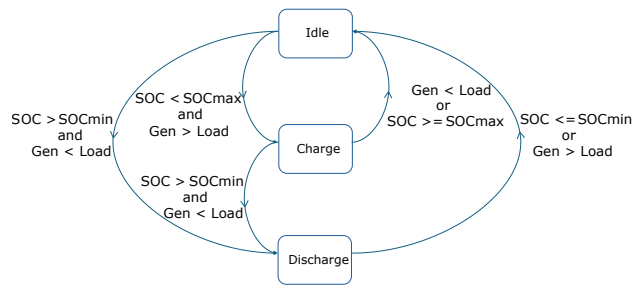


Figure 3. Battery-only controller

For configurations featuring both dispatchable units (DG and BESS), the controller employs a four-state logic architecture that prioritizes PV utilization and battery dispatch, introducing specific generator activation thresholds to ensure system resilience. The diesel unit is activated strictly to recharge the system when the battery SOC falls below a critical lower threshold (SOC_{DgOn}), and it returns to standby once the battery recovers to a designated upper threshold (SOC_{DgOff}). The control flow for this combined strategy is depicted in Figure 4.

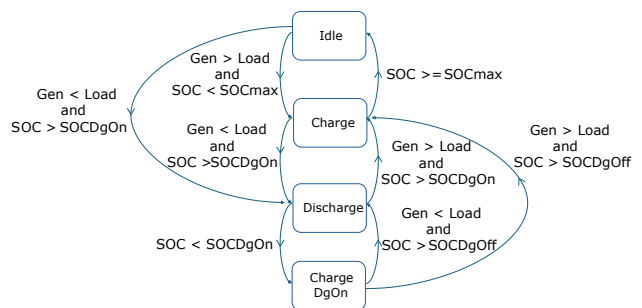


Figure 4. Battery and DG controller

2.3 Performance Metrics

To comprehensively evaluate the viability of each islanded microgrid topology, the system's performance is quantified across key techno-economic and environmental indicators. Specifically, the evaluation framework assesses the total capital cost of the resilient system, the aggregate fuel consumption, and the resulting CO₂ emissions.

Total fuel consumption is dynamically calculated by integrating empirical, part-load ratio (PLR)-dependent performance curves for the active diesel generators over the simulated outage period. Utilizing operational data from (Generator Source 2024), the instantaneous volumetric fuel flow rate is determined as a function of the PLR and subsequently integrated over the duration of the event to obtain the total fuel consumed. The total fuel consumed, $Fuel_{total}$, is calculated as:

$$Fuel_{total} = \int_0^t \dot{V}_{fuel}(PLR(\tau)) d\tau, \quad (1)$$

where $\dot{V}_{fuel}$ is the volumetric fuel flow rate determined by the part-load ratio (PLR) at time τ , and t is the total duration of the simulated outage.

The total carbon emissions, $Emissions_{total}$ (in kg of CO₂), are determined by applying a standard conversion factor (e.g., 2.68 kg of CO₂ per liter of diesel combusted) to the integrated fuel consumption:

$$Emissions_{total} = \int_0^t (\dot{V}_{fuel}(PLR(\tau)) \times 2.68) d\tau. \quad (2)$$

Finally, the total capital cost, $Cost_{total}$, aggregates the installed capacities of the selected DERs multiplied by their respective baseline unit prices (C). These economic parameters were sourced from the National Laboratories of the Rockies' REopt (Simpkins et al. 2014):

$$Cost_{total} = (C_{DG} \cdot P_{DG}) + (C_{PV} \cdot P_{PV}) + (C_{Wind} \cdot P_{Wind}) + (C_{Bat} \cdot E_{Bat}) + (C_{Inv} \cdot P_{Inv}) + C_{Bat, const}, \quad (3)$$

where P denotes the rated power capacity and E denotes the energy capacity. The specific baseline unit costs applied to the system components are $C_{DG} = \$880/\text{kW}$ for the diesel generator, $C_{PV} = \$2,208/\text{kW}$ for solar photovoltaics, $C_{Wind} = \$5,776/\text{kW}$ for wind turbines, $C_{Bat} = \$452/\text{kWh}$ for battery energy capacity, $C_{Inv} = \$489/\text{kW}$ for the battery inverter capacity, and $C_{Bat, const} = \$5,926$ representing the fixed, capacity-independent installation and balance-of-system cost for the battery system.

3 Implementation

The proposed microgrid templates were developed using the Modelica language, leveraging its equation-based, hierarchical structure to effectively model the complex, multi-domain physics of the community energy systems. The top-level model serves as the primary integration interface, linking the user-defined configuration array to

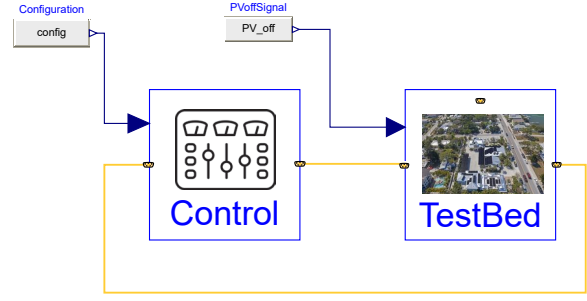


Figure 5. Top-level Modelica model illustrating the integration of the dynamic parameter array, control logic, and physical system (testbed) for the community energy system.

both the physical system and the control logic, as illustrated in Figure 5.

To facilitate rapid evaluation, optimization, and seamless integration with advanced co-simulation environments, both the physical system and the control logic models were structured to be exported as Functional Mock-up Units (FMUs). To automate the simulation workflow across the 12 distinct topologies, an external Python script is utilized to orchestrate the FMU execution. When a specific Configuration ID (referencing Table 1) is selected, the Python algorithm dynamically updates the system parameters prior to initialization. It assigns the targeted sizing capacities to the active equipment while strictly setting the capacities of any omitted components to zero, effectively removing them from the simulation dynamics.

The physical architecture of these interconnected DERs and the community load is encapsulated within the physical system model (Figure 6).

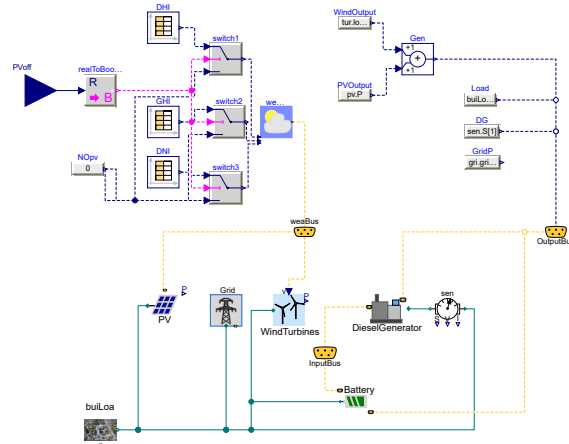


Figure 6. Hierarchical view of the physical system model, highlighting the interconnected distributed energy resources and community load

Structurally decoupled from the physical assets is the controller model (Figure 7). This subsystem encapsulates the dynamic switching mechanism, housing the distinct

baseline operational strategies such as battery-only, diesel-only, and combined dispatch to actively govern the energy routing based on the selected microgrid configuration.

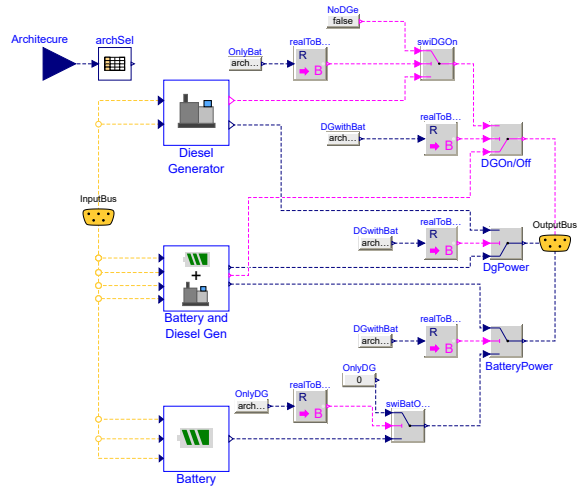


Figure 7. Internal structure of the control model, detailing the routing mechanism for the battery-only, diesel-only, and combined baseline operational strategies.

Signal routing between the physical plant and the supervisory controller is managed through a centralized information bus. The physical system continuously outputs key state variables specifically, aggregated renewable power generation, the DG's instantaneous power output, and the BESS SOC. These metrics are fed as inputs to the controller. Based on the active microgrid topology and control strategy, the controller evaluates these states and dispatches command signals back to the physical assets. These control outputs dictate the discrete on/off status of the DG, the continuous active power setpoint requested from the DG, and the required charge or discharge power for the BESS.

4 Case Study

To demonstrate the practical application and efficacy of the proposed modeling templates, a case study was conducted to determine the optimal equipment sizing for multiple microgrid configurations designed to sustain a one-week power outage. The selected site is the Historic Green Village, a 2,776 m^2 mixed-use community located on Anna Maria Island near Tampa, Florida (Huang et al. 2021). The community comprises one commercial office, two retail stores, and two residential buildings. Geographically classified as ASHRAE Climate Zone 2A (hot and humid) (Edition et al. 2010), the site experiences high solar irradiance, making PV generation a highly viable energy resource. Conversely, due to suboptimal historical wind velocities in this specific region, wind generation was excluded from the optimization scope. Nonetheless, all 12 templated topologies listed in Table 1, including those incorporating wind turbines, were independently

instantiated and executed to confirm that the parameterized switching mechanism correctly assembled the corresponding physical assets and baseline controller for each configuration. This validation, discussed further in Section 5, confirms the general applicability of the templating framework beyond the specific equipment mix used in the Anna Maria Island case study, where wind resources were determined to be unviable for this site.

The baseline electrical load profile for the community was derived from an established open-source library (Huang et al. 2021), while the solar resource availability was determined using empirically measured on-site irradiance data. For this resilience assessment, standard HVAC systems were excluded from the building models to isolate and simulate the community's critical electrical load profile during an emergency event. To capture a worst-case scenario representative of hurricane season in coastal Florida, a deterministic grid outage was simulated over a continuous 168-hour (7-day) period, commencing at midnight on August 20th and concluding at midnight on August 27th.

Based on this operational profile, the islanded network must sustain a peak electrical demand of 31.5 kW. With an average daily energy consumption of 524.2 kWh, the optimal equipment configurations must generate and manage a total of 3,669.4 kWh of critical energy over the simulated week.

5 Results and Discussion

To evaluate the proposed modular templates, the system was sized to sustain all critical loads during a simulated one-week outage. For the baseline DG configuration, capacity was fixed at 40 kW to cover the absolute peak load alongside a 20% engineering safety margin, assuming unconstrained fuel availability. Conversely, for the BESS-only topology, the required capacity to reliably meet the weekly energy demand with a 10% safety margin was determined to be 4,748 kWh. Following these baselines, a parametric exhaustive search was conducted across the hybrid topologies specifically combining PV with a DG, a BESS, or both to identify optimal equipment sizing. The search space was constrained as follows: PV capacity ranged from 10 kW to 100 kW in 10 kW increments, while BESS capacity ranged from 50 kWh to 500 kWh in 50 kWh increments, with its power capacity held constant at 40 kW. The DG capacity remained fixed at 40 kW across all simulations. The results of this exhaustive search are illustrated in Figure 8. It is important to note that the exhaustive search approach used here is a demonstration of the templating framework's evaluation capability rather than a scalable design-optimization method. With only two free sizing variables (PV and BESS capacity), the search space was limited to 100 combinations; introducing additional free variables (e.g., DG or wind capacity, or dispatch thresholds) would cause the number of required simulations to grow combinatorially.

For each simulated configuration, the Python orchestration script assigns the corresponding equipment capacities to the physical-system FMU, executes a co-simulation of the physical system and controller FMUs over the full 168-hour outage period, and records whether the critical load was sustained without interruption, along with the resulting capital cost, fuel consumption, and CO₂ emissions. Configurations that fail to sustain the critical load throughout the simulated outage are excluded from further analysis. Among the surviving configurations, the Pareto-optimal frontier is identified via a non-dominance filter applied to the capital cost and emissions results in Python: a configuration is retained on the Pareto front if no other simulated configuration in the search space achieves both a lower capital cost and lower emissions. This filtering step, rather than a formal multi-objective optimization algorithm, identifies the Pareto-optimal set from the exhaustively simulated design space described above.

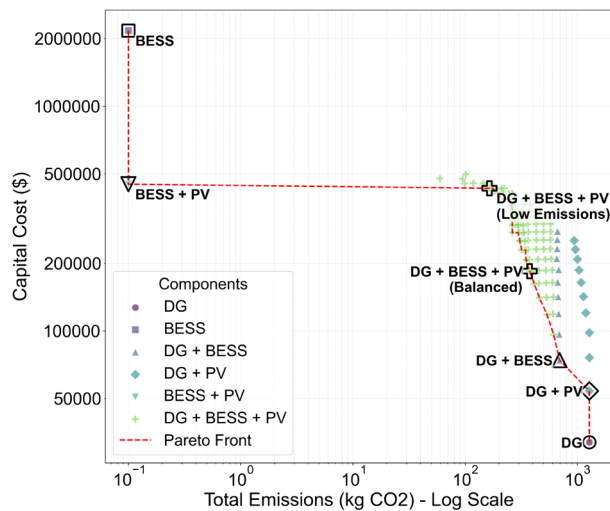


Figure 8. Exhaustive search for component sizing

Figure 8 illustrates only the configurations that successfully sustained the total critical load for the entire one-week duration; any configuration that experienced a loss of load was classified as a failure and excluded from the plot. The survival rates varied significantly across the evaluated topologies. For the PV+BESS architecture, only two specific sizing combinations successfully met the load. In contrast, all configurations incorporating a DG (DG+PV, DG+BESS, and the fully hybrid DG+BESS+PV) successfully survived the outage. To narrow the viable options, the analysis focused exclusively on configurations on the Pareto front, which represent optimal trade-offs between capital cost and carbon emissions.

The ultimate selection of an optimal microgrid architecture depends heavily on stakeholder priorities regarding capital expenditure and environmental impact. For instance, while a BESS+PV system achieves zero emissions, it incurs prohibitive capital costs, and a BESS-only configuration remains highly unrealistic due to its mas-

sive physical footprint and the need to be fully charged precisely when a multi-day outage occurs. Conversely, a DG-only system offers the lowest capital cost but generates the highest emissions and requires substantial on-site fuel storage. To conduct a balanced comparative analysis, representative optimal configurations were strategically selected from the Pareto front. Specifically, the DG+PV configuration with the largest PV capacity was selected to maximize renewable penetration, while the single most optimal point on the Pareto front was chosen for the DG+BESS setup. Finally, to represent divergent stakeholder goals within the fully hybrid DG+BESS+PV architecture, two distinct configurations were selected: one prioritizing the lowest possible emissions, and a balanced configuration that effectively mitigates both capital cost and carbon output. The specific equipment capacities, required capital investments, and total carbon emissions for the seven selected configurations are detailed in Table 2.

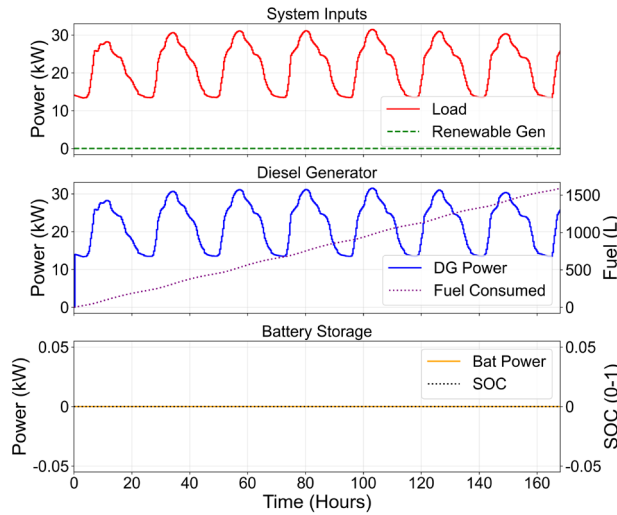
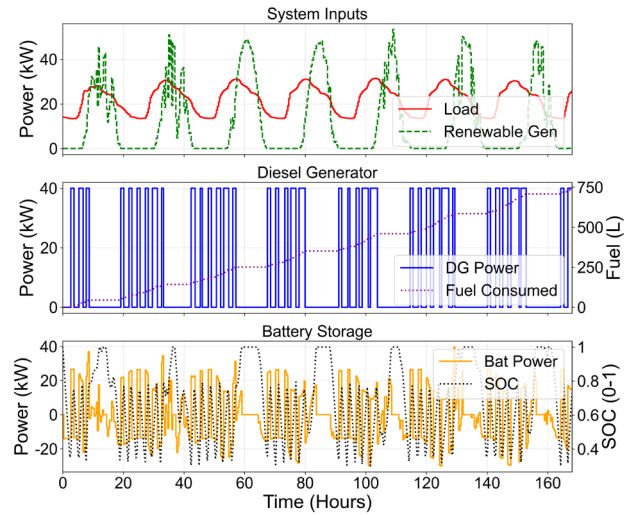
To validate the dynamic operational stability and the efficacy of the applied control logic, high-fidelity time-series dispatch simulations were analyzed for four representative architectures: the DG-only system, the purely renewable PV+BESS system, and the two fully hybrid (DG+BESS+PV) configurations representing the "Balanced" and "Low Emissions" Pareto extremes. These four configurations were selected because they span the extremes and a representative midpoint of the Pareto front — from lowest capital cost/highest emissions to highest capital cost/zero emissions — providing the most informative test of the controller's dynamic behavior across the design space. All 12 templated topologies listed in Table 1 were independently instantiated and simulated to confirm correct automatic assembly of the corresponding physical system and controller; detailed transient dispatch analysis was limited to these four representative cases for clarity, as the Anna Maria Island case study itself does not include wind generation as an active component. For each selected configuration, the transient behaviors of the critical load, renewable generation, DG power output, cumulative fuel consumption, BESS power, and SOC were plotted.

Serving as the baseline with the lowest capital cost and maximum emissions, the DG-only configuration (Figure 9) demonstrates strict load-following behavior. The diesel generator continuously modulates its power output to perfectly match the community demand without any energy storage buffer. Consequently, the cumulative fuel consumption during the one-week outage is approximately 1,600 liters, underscoring the absolute necessity of a large on-site fuel storage tank to ensure community resilience in this topology.

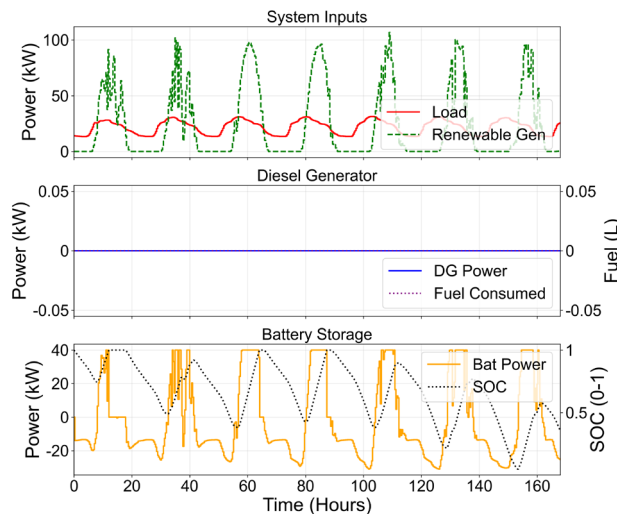
Conversely, the purely renewable PV+BESS architecture (Figure 10) relies entirely on solar generation and energy storage to maintain critical power. As depicted, the BESS absorbs excess PV generation during peak daylight hours and discharges to meet the community load when renewable generation is insufficient, primarily overnight. Because the PV array was optimally sized to simultane-

Table 2. Selected Pareto Front Configurations and Operational Metrics

Configuration	PV (kW)	BESS (kWh)	DG (kW)	Capital Cost (\$)	Emissions (kgCO ₂)
DG	0	0	40	31,926	1,281
BESS	0	4,748	0	2,171,853	0
DG + BESS	0	50	40	74,086	695
DG + PV	10	0	40	54,006	1,281
BESS + PV	100	450	0	449,686	0
DG + BESS + PV (Balanced)	50	50	40	184,486	375
DG + BESS + PV (Low Emissions)	80	400	40	431,526	164

**Figure 9.** Dynamic operational behavior of the DG-only system.**Figure 11.** Dynamic operational behavior of the DG + BESS + PV (Balanced) hybrid system.

ously supply the transient load and fully recharge the battery, this configuration successfully survives the extreme weather event with absolute zero fuel consumption.

**Figure 10.** Dynamic operational behavior of the purely renewable PV + BESS system.

For the hybrid configurations, the supervisory control

logic utilizes a robust cycle-charging strategy. As illustrated in the "Balanced" hybrid system (Figure 11), the controller dynamically balances the assets by activating the diesel unit at maximum capacity only when the battery SOC drops below a predefined 30% minimum threshold. The generator then remains active until the battery is recharged to an 80% upper limit. Because this specific topology minimizes capital expenditure by utilizing a smaller battery, the BESS depletes rapidly, forcing the DG to cycle on multiple times to satisfy the load. While this reduces upfront capital costs, it noticeably increases total fuel consumption and imposes more micro-cycles on the battery. If deployed in a standard grid-connected mode, this elevated cycling frequency would likely accelerate battery degradation and reduce its operational lifespan.

In contrast, the "Low Emissions" hybrid architecture (Figure 12) achieves a superior operational balance through increased BESS and PV capacities. As shown, the larger battery acts as a substantial buffer, significantly reducing the frequency of DG activation. When combined with the larger PV array, the battery is often able to reach a full SOC solely from renewable surplus. Ultimately, this structural design dramatically reduces cumulative diesel

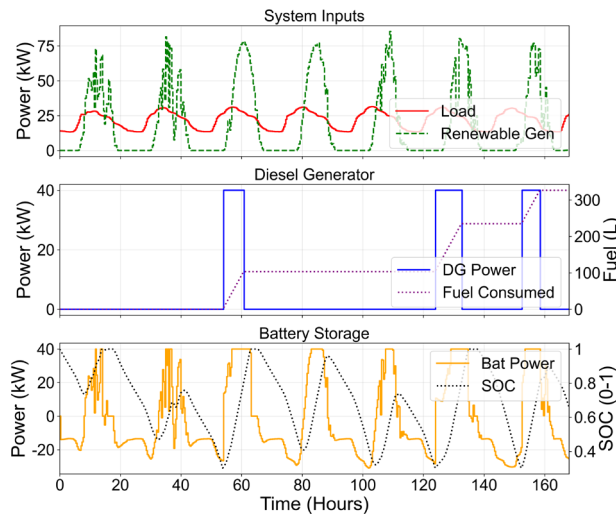


Figure 12. Dynamic operational behavior of the DG + BESS + PV (Low Emissions) hybrid system.

consumption, minimizes carbon emissions, and decreases the cyclical stress placed on the BESS, thereby preserving battery health over the system's operational lifespan.

6 Conclusion and Future Work

As extreme weather events increasingly expose the fragility of centralized electrical grids, resilient community microgrids have emerged as critical infrastructure for sustaining power during prolonged outages. However, designing these systems is historically constrained by the rigid, hardcoded nature of high-fidelity simulation tools. To address this, this study successfully developed a highly modular Modelica template framework that enables the seamless generation and execution of diverse microgrid architectures. By structurally decoupling the physical plant models from the supervisory control logic, this standardized approach simplifies efficient evaluation of the techno-economic and environmental trade-offs among distinct physical configurations without requiring manual model remapping.

Applying this framework to a deterministic community outage case study successfully demonstrated its analytical capabilities. The simulation mapped the Pareto-optimal frontier, quantifying the specific capital-cost and carbon-emission trade-offs required to keep the community powered. Furthermore, the framework validated the dynamic dispatch behavior and operational stability of purely renewable, diesel-only, and fully integrated hybrid operations, explicitly highlighting how equipment sizing and control thresholds directly impact fuel consumption, capital expenditure, and battery cycling.

Future research will build directly upon this foundational model. Because the physical plant is completely decoupled from the operational brain, researchers can now rapidly implement, test, and benchmark novel supervisory control algorithms against the baseline strategies es-

tablished in this work. Furthermore, advanced mathematical optimization algorithms will be integrated to explore more complex, multi-objective sizing strategies. Ultimately, this modular template will serve as the critical foundation for a comprehensive co-design framework one that evaluates both optimal asset sizing and dynamic control logic simultaneously to maximize the resilience of future net-zero energy communities.

Acknowledgements

This research was supported by the National Science Foundation, United States under Award Number ECCS-2433426. This work was also authored in part by the National Laboratory of the Rockies for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. The views expressed in the article do not necessarily represent the views of the DOE or the U.S. Government. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for U.S. Government purposes.

References

- Balducci, Patrick, Kendall Mongird, Di Wu, Dexin Wang, Vanishika Fotedar, and Robert Dahowski (2020). "An evaluation of the economic and resilience benefits of a microgrid in Northampton, Massachusetts". In: *Energies* 13.18, p. 4802.
- Campbell, A. F. (2018). *It Took 11 Months to Restore Power to Puerto Rico After Hurricane Maria. A Similar Crisis Could Happen Again*. Vox [Online]. Accessed January 12, 2022. URL: <https://www.vox.com/identities/2018/8/15/17692414/puerto-rico-power-electricity-restored-hurricane-maria>.
- Castro, Marcelo de, Dietmar Winkler, Giuseppe Laera, Luigi Vanfretti, Sergio A. Dorado-Rojas, Tin Rabuzin, Biswarup Mukherjee, and Manuel Navarro (2023). "Version OpenIPSL 2.0.0 - iTesla Power Systems Library (iPSL): A Modelica library for phasor time-domain simulations". In: *SoftwareX* 21. ISSN: 2352-7110. DOI: 10.1016/j.softx.2022.101277. URL: <https://www.sciencedirect.com/science/article/pii/S2352711022001959>.
- Cramton, Peter (2022). "Fostering resiliency with good market design: Lessons from Texas University of Maryland Working Paper". In.
- Do, Vivian, Heather McBrien, Nina M Flores, Alexander J Northrop, Jeffrey Schlegelmilch, Mathew V Kiang, and Joan A Casey (2023). "Spatiotemporal distribution of power outages with climate events and social vulnerability in the USA". In: *Nature communications* 14.1, p. 2470.
- Dosa, Adebayo, Oludolapo Akanni Olanrewaju, and Felix Mora-Camino (2025). "A Comprehensive Review of Hybrid Renewable Microgrids: Key Design Parameters, Optimization Techniques, and the Role of Demand Response in Enhancing System Flexibility". In: *Energies* 18.19, p. 5154.
- Edition, SI, DH Erbe, MD Lane, SI Anderson, PA Baselici, S Hanson, R Heinisch, J Humble, S Taylor, and R Kurtz (2010). "Energy standard for buildings except low-rise residential

- buildings”. In: *ASHRAE, Atlanta, GA, USA, Tech. Rep. IP Edition*.
- Energy, Celtic (2017). “Resilient microgrids for Rhode Island critical services”. In: *Celtic Energy: Caerphilly, UK*.
- Fachini, Fernando, Srijita Bhattacharjee, Miguel Aguilera, Luigi Vanfretti, Giuseppe Laera, Tetiana Bogodorova, Ardesir Moftakhari, Michael Huylo, and Atila Novoselac (2023). “Exploiting Modelica and the OpenIPSL for University Campus Microgrid Model Development”. In: *Modelica Conferences*, pp. 285–292.
- Ganesh, Viswanathan, S Senthilmurugan, and Rathinam Ananthanarayanan (2022). “Implementation of Peak Shaving Algorithm in an Islanded Microgrid for Economic Power Consumption”. In: *Smart Energy and Advancement in Power Technologies: Select Proceedings of ICSEAPT 2021 Volume 1*. Springer, pp. 49–60.
- Generator Source (2024-12-10). *Fuel Consumption Charts*. URL: <https://generatorsource.com/tools-info/fuel-consumption-charts/> (visited on 2026-03-31).
- Huang, Sen, Jing Wang, Yangyang Fu, Wangda Zuo, Kathryn Hinkelman, Raymond M Kaiser, Dong He, and Draguna Vrabie (2021). “An open-source virtual testbed for a real net-zero energy community”. In: *Sustainable Cities and Society* 75, p. 103255.
- Idima, Smart, Chineme Edger Nwatu, Ekene Micheal Adim, and Ifeanyichukwu Jeffrey Okwesa (2023). “Predictive analytics for aging US electrical infrastructure: Leveraging machine learning to enhance grid resilience and reliability”. In: *World Journal of Advanced Research and Reviews* 19.2, pp. 1595–1622.
- Simpkins, Travis, Dylan Cutler, Kate Anderson, Dan Olis, Emma Elgqvist, Michael Callahan, and Andy Walker (2014). *REopt: a platform for energy system integration and optimization*. Vol. 45875. American Society of Mechanical Engineers.
- Wetter, Michael, Wangda Zuo, Thierry S Nouidui, and Xiufeng Pang (2014). “Modelica buildings library”. In: *Journal of Building Performance Simulation* 7.4, pp. 253–270.
- Yan, Bing, Peter B Luh, Guy Warner, and Peng Zhang (2017). “Operation and design optimization of microgrids with renewables”. In: *IEEE Transactions on automation science and engineering* 14.2, pp. 573–585.
- Zuo, Li, Yuanhui Dong, Shubin Zhang, Yuxin Li, Haiming Zhang, Ji Ding, Fanli Zhou, Qi Liu, and Liping Chen (2025). “Dynamic Simulation of Off-Grid Energy Island with Wind-PV-Storage Hydrogen Production”. In: *Modelica Conferences*, pp. 909–920.

Rumoca: Modelica as a Universal Algebraic Frontend via a Rust-Native Compiler

Micah K. Condie¹ Abigaile Woodbury¹ Thomas Meschede² Joel A. E. Andersson³ James M. Goppert¹

¹School of Aeronautics and Astronautics, Purdue University, USA, {condiem, awoodbu, jgoppert}@purdue.edu

²Xyntopia LLC, USA, thomasm@xyntopia.com

³FMIOPT AS, Norway, joel@fmiopt.com

Abstract

Modelica is a well established cyber-physical modeling language, but many modern engineering workflows, such as optimization, differentiable simulation, scientific machine learning, and system analysis, make use of other environments such as CasADi, JAX, and Julia. Existing infrastructure for Modelica toolchains does not target these languages directly, meaning models must typically be rewritten, or lose important information in the interface. This paper presents Rumoca, a Rust-native Modelica compiler that turns Modelica into a universal algebraic frontend for a variety of tools. Rumoca is organized as a sequence of explicit phase boundaries from parsing through Differential-Algebraic Equation (DAE) construction and template-driven code generation, with a native Rust simulation backend that also supports real-time, software-in-the-loop execution. We report quantitative coverage of the Modelica Standard Library across pipeline phases, together with compile-time and simulation-time benchmarks against an open-source reference. The full compiler ships as a VS Code extension and runs in the browser via WebAssembly, enabling zero-install playgrounds and self-contained HTML simulators. End-to-end case studies demonstrate realtime software-in-the-loop control of a quadrotor model and deployment of a single Modelica source across multiple algebraic backends.

Keywords: *Modelica compiler, code generation, Rust, WebAssembly, FMI, CasADi, differentiable simulation, MSL benchmarking*

1 Introduction

The Modelica language is widely used for declarative, equation-based modeling of cyber-physical systems. The mature Modelica toolchain, led on the open-source side by OpenModelica (Fritzson et al. 2020) and complemented by industrial environments such as Dymola, is highly effective at simulation, FMU export, and graphical authoring. However, a growing set of engineering workflows live downstream of simulation. Optimization, differentiable simulation, scientific machine learning, and system analysis are all workflows typically built on symbolic and numerical frameworks such as CasADi (Andersson, Gillis,

et al. 2019), JAX, and SymPy. Existing common Modelica tools do not target these languages directly. In current practice, an engineer working across these ecosystems may end up writing the same model several times. For example, once in Modelica for simulation, again in CasADi for optimization, and a third time in SymPy for analysis, with no inherent guarantee that the three implementations of the model are the same. Moreover, these frameworks' modeling syntax is often harder to learn than Modelica's, since the model must be assembled imperatively in a general-purpose host language rather than declared as equations in a form close to textbook notation. Engineers who would benefit from Modelica's modeling discipline are often forced to use general-purpose languages or closed environments such as Simulink, where the tooling ecosystem is broader even when the modeling abstractions are weaker.

This paper presents Rumoca, a Rust-native Modelica compiler that turns Modelica into a universal algebraic frontend for a variety of tools. Rumoca preserves Modelica as a dedicated declarative source language and exposes its intermediate representation to template-driven backends that emit CasADi, SymPy, FMI, JavaScript, JAX, Julia, ONNX, or embedded C, as well as to a native-Rust simulation runtime with exact automatic-differentiation Jacobians and software-in-the-loop (SIL) support.

The remainder of this paper is organized as follows:

- Section 2 reviews background on Modelica, FMI, and the Rumoca project.
- Section 3 describes the Rumoca compiler architecture: a pipeline of explicit phase boundaries with typed intermediate representations, template-driven code generation to multiple algebraic backends, and a native Rust simulation backend with real-time SIL support, all exposed through unified command-line interface (CLI), language service protocol (LSP), WebAssembly, and Python frontends.
- Section 4 presents a quantitative Modelica Standard Library (MSL) parity benchmark across pipeline phases, together with compile- and simulation-time

microbenchmarks against a reference open-source compiler.

- Section 5 describes Rumoca’s WebAssembly-native deployment story, including self-contained HTML simulators that compile a Modelica model into a single shareable artifact runnable in any modern browser.
- Section 6 presents end-to-end case studies in which a single Modelica model is used as a realtime SIL plant for an external autopilot and as a source for multiple algebraic deployments.

2 Background

Rumoca builds on two existing standards and extends a tool we described previously. This section reviews the Modelica language and the properties that make it a suitable source language for algebraic backends, the FMI standard that Rumoca targets as an export format, the surrounding landscape of Modelica and Modelica-adjacent tooling, and the evolution of Rumoca from a translator into a compiler.

2.1 Modelica

Modelica (Modelica Association 2017) is an equation-based, object-oriented modeling language for cyber-physical systems. Unlike procedural modeling approaches, Modelica models are expressed declaratively, with computational causality determined automatically by the compiler. This enables reusable component libraries and acausal multi-domain modeling. In procedural and signal-flow environments, computational causality is specified explicitly: block-diagram tools such as Simulink define directed computations between inputs and outputs, and hand-written Ordinary Differential Equations (ODE) implementations similarly require the modeler to choose which variables are solved for directly. In contrast, a Modelica equation such as $m\dot{v} = -kx - cv$ specifies a physical relationship without prescribing a computational direction. The compiler determines the required causality during transition.

The modeler can still fix a direction when one is wanted. Top-level signals can be marked as `input` or `output` for FMI export, for control design, or for connecting to a causal environment. The equations inside a component do not have to commit to a direction. As a result, the same resistor, mass, or pipe model can be reused in different contexts. Component libraries follow the same discipline. Typed `connector` and `connect` equations let a library author publish reusable parts. The language enforces the interconnection rules so each user does not have to reimplement them. The MSL ships several thousand such components across electrical, mechanical, thermal, fluid, and control domains.

Modelica is also a plain-text language. A model is a file. It can be read, diffed, version-controlled, and gen-

erated by a script in the same way as any other source code. This matters for large language models. An LLM can read a Modelica file directly, reason about its structure, and propose edits in the same medium. A visual block diagram cannot be inspected this way. It has to be serialized into some intermediate form first, if such a form exists. As LLMs become a standard tool in engineering workflows, the readability of the source language matters as much as its expressive power. These properties make Modelica attractive even when the downstream consumer is not a traditional simulator, such as when the model feeds an optimizer, a learned controller, or a symbolic analysis.

Listing 1 shows a complete Modelica model of a bouncing ball. The model has a continuous dynamics ($\ddot{h} = -g$), an event condition ($h \leq 0$ moving downward), and a discrete reset that flips the velocity and applies a coefficient of restitution. The compiler extracts the state variables, detects the zero-crossing, and re-initializes the integrator at each event.

Listing 1. A Modelica model of a bouncing ball. The `equation` section declares the continuous dynamics; the `when` clause declares an event-triggered discrete reset.

```
model BouncingBall
  parameter Real e = 0.8 "coefficient of
    restitution";
  parameter Real g = 9.81 "gravity";
  Real h(start = 1.0) "height";
  Real v(start = 0.0) "velocity";
equation
  der(h) = v;
  der(v) = -g;
  when h <= 0 and v < 0 then
    reinit(v, -e * pre(v));
  end when;
end BouncingBall;
```

2.2 FMI

The Functional Mock-up Interface (FMI) (Modelica Association 2022) is a tool-independent standard for packaging simulation models as Functional Mock-up Units (FMUs): zip archives containing an XML interface description and a compiled implementation of the model’s dynamics. FMI defines two interaction modes. In *Model Exchange* the host provides the integrator and the FMU exposes residuals, event indicators, and reset logic; in *Co-Simulation* the FMU embeds its own solver and is advanced by the host over a communication step. The standard does not specify a modeling language, but was developed in the Modelica community and the two are commonly encountered together. Recent work has extended FMI handling in CasADi to support symbolic import, event dynamics, and analytic sensitivities, narrowing the gap between FMI’s traditional black-box framing and the symbolic workflows targeted by Rumoca (Andersson 2023; Andersson and Goppert 2024). Rumoca’s FMI backend (section 3) emits dual-mode FMUs that declare both interfaces from a single Modelica source.

2.3 Related Work

The first commercial implementation of a Modelica compiler is Dymola (Dassault Systèmes 2025), developed by Dynasim and now owned by Dassault Systèmes. It remains the reference commercial tool for simulation and FMU export. On the open-source side, OpenModelica (Fritzson et al. 2020) is the most widely used compiler. It provides simulation, FMU export, and graphical authoring through OMEdit. A more recent open-source effort is MARCO (Agosta et al. 2023), an LLVM/MLIR-based C++ compiler aimed at high-performance simulation of large-scale models. These tools all treat simulation as the primary deliverable. Rumoca instead targets the symbolic and algebraic workflows that sit downstream of simulation: optimization, differentiable simulation, scientific machine learning, and symbolic analysis.

Not all of this work has treated simulation as the deliverable. Optimica and JModelica.org (Åkesson et al. 2010) extend Modelica with constructs for the optimization interval, the cost function, and the constraints, and transcribe the resulting problem into a nonlinear program rather than a simulation run. Rumoca shares the premise that a Modelica compiler is useful for more than transient simulation, but differs in mechanism: rather than extending the language and implementing the analysis inside the compiler, it exposes the DAE through templates so the model can be consumed in the native idiom of whichever tool the analysis calls for.

Another line of work worth mentioning is Julia-hosted modeling tools. ModelingToolkit.jl (Ma et al. 2021) embeds an acausal, equation-based modeling layer inside Julia. It exposes SciML solvers and Julia’s broader numerical ecosystem directly to the modeler. Dyad (JuliaHub 2025), released by JuliaHub in 2025 is a direct comparison to Modelica. It is a declarative, equation-based modeling language with object-oriented composition, typed connectors, GUI authoring, a standard component library, and FMI export. It covers essentially the same design space as Modelica. The differences are in the surrounding ecosystem and licensing. Dyad is released under a source-available license rather than an open one. Its compiler lowers Dyad source to ModelingToolkit.jl code, so models ultimately run only inside the Julia/SciML stack. Its standard component libraries are BSD3-licensed and are explicitly intended to replace the Modelica Standard Library over time.

Rumoca takes a different stance. Rather than introducing a new language tied to one runtime, it preserves Modelica itself as an open, declarative frontend and emits the model into many host languages, including ModelingToolkit.jl itself. The two approaches trade off against each other: a clean-slate language like Dyad is free to introduce new modeling features directly, whereas Rumoca, building on the existing Modelica language, would need such features added to Modelica itself. In exchange, Rumoca inherits Modelica’s 25-year history and large public cor-

pus of open-source models, including the MSL. This is an advantage that also benefits LLM workflows, since an LLM can be trained on or grounded against that corpus.

2.4 Evolution of Rumoca

In our previous paper (Condie et al. 2025) we introduced Rumoca as a Rust-based translator from Modelica to algebraic modeling languages, with initial CasADi and SymPy backends and a Jinja-driven template mechanism for additional targets. That paper positioned Rumoca as a translator, demonstrated it on three example models, and outlined a path in which complex Modelica models would first be lowered to flattened Modelica by an external compiler before being handed to Rumoca.

Since that publication, Rumoca has expanded into a Modelica compiler. This change in scope was made possible by AI co-development (see subsection 3.5). Even so, implementing full Modelica support takes time, and using flattened Modelica as a bridge between more mature compilers and Rumoca is still necessary in some cases. Rumoca now implements parsing, name resolution, type checking, instantiation, flattening, DAE construction, structural analysis with initialization-condition planning, and template-driven code generation across explicit phase boundaries. A native Rust simulation backend provides exact automatic-differentiation Jacobians and solver fallbacks, the full pipeline runs in the browser via WebAssembly, and the project maintains a continuous-integration parity gate against MSL. Rumoca is distributed through CLI, language-server, Python, and VS Code interfaces. The compiler is developed in the open at <https://github.com/CogniPilot/rumoca> and released under the Apache-2.0 license. The remainder of this paper describes the resulting system.

3 Architecture, Features, and Development

Rumoca is designed such that each phase of the compiler has a clear scope and produces an accessible artifact. Solvers, code generators, and simulators all consume clear intermediate representations, but none of these consumers is permitted to influence the IRs. Rather, each IR is either transformed into the next IR by a compiler phase, or consumed terminally by a downstream tool. This clear separation of representations is what aids the design of Rumoca as a Modelica frontend for many ecosystems rather than a simulator alone. Other than code generation, features of Rumoca include SIL and native Rust simulation with automatic differentiation. Rumoca can be used through the CLI, through a VS Code extension, and imported as a Python package (a pip-installable wheel) or npm package.

3.1 Pipeline Overview

Rumoca consists of two main peer activities, *Compile* and *Simulation*. Compile is responsible for parsing and building the various internal representations of the model,

while Simulation takes the IR representations produced by the Compile API and applies the relevant solvers and IO tools required for simulation. Internally, every activity in Rumoca is implemented as a sequence of phases, each phase living in its own Rust crate and communicating only through explicit transitions. This organization, of clearly defining crate scope, mirrors that of the Rust compiler itself (The Rust Project Developers 2025), whose source tree is split across dozens of crates with strict phase boundaries. Compile and Simulation activities are exposed by the four entry points of Rumoca: the command-line binary, an LSP server (used in the VS Code extension), Python bindings, and WebAssembly bindings. These will be described in more detail later.

Figure 1 shows an overview of the Rumoca architecture, though in reality the implementation consists of 45 Rust crates. Within the compile activity, there are four stages of processing a Modelica model goes through, each represented by a unique IR. The first pass on the model is done by the Parol (Singer 2025) parser. This produces the Parsed IR that still contains the native object-oriented structure of the model. The second phase then flattens this model, producing the Flattened IR, which has the object-oriented structure removed and the connect equations resolved. This bears the closest resemblance to traditional flattened Modelica (similar to that used in OpenModelica). The next pass produces the DAE IR, which arranges the equations of the model in a hybrid differential-algebraic form as specified by the Modelica Language Specification Appendix B. This includes variables being tagged by role (state, algebraic, discrete, parameter) and equations tagged by category (continuous, discrete update, event condition).

Between the DAE IR and the Solver IR, Rumoca performs the standard structural preparation. Exact scalar incidence is taken from the checked DAE expressions; equations are assigned to unknowns by maximum bipartite matching over augmenting paths, sorted into block-lower-triangular form from the strongly connected components of the dependency graph, and coupled blocks are torn by a greedy heuristic so that a small nonlinear core is iterated while the remainder is evaluated explicitly. Higher-index systems are reduced by the dummy-derivative method over a Pantelides-style differentiation of unmatched constraints: the index-1 lift is complete, higher-index lifts cover an explicit subset, and constraints retained by the reduction are enforced at runtime by a minimum-norm manifold projection. Event support covers time events, state events from relational zero crossings, when and elsewhere equations, when statements in algorithms, and the discrete and temporal operators (pre, previous, edge, change, reinit, delay, noEvent, smooth). Synchronous support covers periodic clocks, declared as `Clock(interval)` or `Clock(intervalCounter, resolution)`, together with the clock conversions and clocked partitions exercised by the `Clocked` results of Table 1; event and

solver clocks, `spatialDistribution`, and Modelica state machines (MLS §17) are not yet lowered to a DAE and are rejected with a diagnostic rather than silently mistranslated. The per-phase rates of section 4 localize the remaining gap: attrition is concentrated at DAE construction for connection-heavy libraries and at initialization for the machine and magnetic libraries, which is where the index-reduction and initialization work is currently directed.

A final pass produces the Solver IR, which represents the model lowered to a flat, register-based opcode stream with Jacobian-vector products precomputed by forward-mode automatic differentiation, ready for direct numerical evaluation by an interpreter or a JIT backend. This multi-pass discipline, where each phase is distinct and crosses a typed IR boundary, simplifies testing and AI-assisted contribution. It also opens the door to advanced diagnostics: when a simulation or optimization fails, the user can inspect each IR in turn—Parsed, Flattened, DAE, or Solver—to localize the problem to a specific compiler phase.

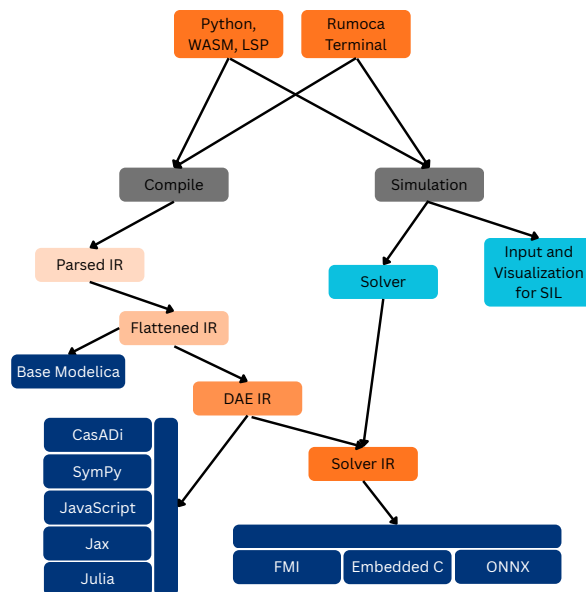


Figure 1. Rumoca’s architecture. Compile and Simulation are peer activities, each callable from any frontend. The IR’s are used to generate the various backends. The Solver IR used for Rumoca’s native simulation

3.2 Code Generation

Code generation is template-driven and can operate on any of the IR phases. Templates are processed by the Jinja2 template engine (Pallets 2025), a widely used text-templating system originating in the Python web-development ecosystem, where it is commonly used to generate HTML pages from dynamic data. The same mechanism applies cleanly to code generation: Jinja2 walks the prepared DAE and emits the model in the syn-

tax specified by a particular template. Each backend is a separate template; Jinja2’s control-flow constructs (loops, conditionals) handle the per-model variation within a template. New backends can be added by writing additional templates, using existing ones as examples. The templates currently shipped with Rumoca, in approximate order of maturity, are:

- **CasADi** in both MX and SX expression flavours — MX for matrix-valued DAGs of arbitrary CasADi nodes, SX for dense scalar-valued trees.
- **SymPy** for symbolic analysis and verification.
- **FMI 2.0 and FMI 3.0**, exported as dual-mode FMUs that declare both Model Exchange and Co-Simulation; Model Exchange is the accurate path, with Co-Simulation provided as a convenience wrapper for tools that only consume CoSim FMUs.
- **JavaScript** exported as self-contained js code, including solvers for inclusion in web-based frontends.
- **JAX** for differentiable simulation and machine learning.
- **Julia / ModelingToolkit** that works with SciML solvers. Simple examples have been shown to work with this target, but more comprehensive models and testing are in progress.
- **ONNX** for export of model components into machine-learning runtimes. Simple examples have been shown to work with this target, but is not yet fully implemented.
- **Embedded C**, currently at an early stage: the template generates a self-contained C model suitable for integration into bare-metal or RTOS targets, and has been flown experimentally on a quadrotor through the CogniPilot Cerebri autopilot, but is not yet recommended for production use.

In addition to these consumer backends, Rumoca ships two other templates. The first produces flat Modelica, adhering to the proposed Base Modelica standard (Modelica Association 2023). The second re-emits the compiler’s own DAE and flat-model IRs as Modelica source. These are not translation targets in the conventional sense but inspection and traceability surfaces: they let a user see exactly what the flattening or DAE-formation phase has produced. The DAE re-emission is particularly useful as an entry point for custom tools and analyses, since it exposes Rumoca’s canonical hybrid DAE in a human- and machine-readable form that downstream consumers can parse without re-implementing any of the upstream compiler phases. It also provides a natural handoff to other tools expecting Base Modelica-style input.

3.3 Simulation and SIL

Rumoca’s simulation runtime consumes the Solver IR and ships with two solver backends: a pure-Rust RK45 explicit integrator for non-stiff problems, and an integration with DiffSol (Robinson and Allmont 2026), an open-source Rust ODE and semi-explicit DAE solver from the University of Oxford, which provides BDF and ESDIRK methods for stiff and implicit-DAE problems. Initialization-condition planning runs ahead of integration, and residual evaluation can be optionally JIT-compiled through Cranelift (Bytecode Alliance 2025) on native targets and through WebAssembly in the browser. Initialization solves the initial equations together with the MLS §8.6 equations induced by `fixed = true` start attributes over a combined state-parameter vector, so `fixed = false` parameters enter as initialization unknowns; the solve is a scaled damped Newton iteration with line search and a per-iteration step limit that keeps the iterate on its starting branch, and `homotopy(actual, simplified)` is implemented as a true λ -continuation rather than the trivial reading the specification permits. Jacobians and mass matrices are obtained by exact forward-mode automatic differentiation of the DAE IR rather than finite differences, giving the solver analytic derivatives at every step. The runtime supports both batch execution and a deterministic, step-locked realtime mode suitable for SIL work, with named-signal I/O serialized through a FlatBuffers codec (Google 2025) and transported over UDP for desktop clients or WebSockets for browser viewers and remote operators. Live operator input is supplied through gamepad and keyboard backends. For visualization, simulations can drive a Three.js scene, giving the operator a live 3D view of the plant alongside the numeric feed.

3.4 Additional Features

The language server, exposed through the *Rumoca Modelica* extension on the VS Code marketplace, makes the compiler’s internal information available at edit time. Beyond standard LSP features — realtime diagnostics, autocomplete, hover information, go-to-definition, find-references, document symbols, signature help, and formatting — the extension integrates directly with the compiler’s simulation surface. A user can edit a `.mo` file, trigger a one-click simulation run, and receive a self-contained HTML report with embedded plots, all without leaving the editor. The extension also exposes a configurable visualization view system: in addition to standard time- and X/Y-plot views, a 3D view generates a Three.js scene script for the model that the user can edit directly to reflect the model’s physical state, giving each Modelica model a customizable runtime visualization without leaving the source-tree workflow.

3.5 AI-Assisted Compiler Development

Rumoca has been developed with substantial assistance from frontier large-language-model based coding agents.

This has only worked under close human direction and with project-specific guardrails. For semantic changes, we require an explicit lookup of the relevant Modelica Language Specification rule. We also require the agent to respect Rumoca’s phase boundaries, identify the owning compiler phase, and avoid fixing downstream consumers when the error belongs in the compiler pipeline.

In practice, the useful workflow is to define the compiler contract first and let the agent implement inside that boundary. For compiler bugs, we usually ask the agent to reproduce one concrete failure, trace the affected symbol or equation through the pipeline, inspect intermediate artifacts such as flattened models or DAEs, and compare phase-level MSL outcomes. This helps distinguish a real regression from a change that improves an earlier phase while exposing a pre-existing limitation in a later one.

Despite these measures, the main limitation remains unresolved: agents still struggle with root-cause analysis and frequently repair the visible symptom. A simulation failure may lead to a solver-side workaround. A diagnostic failure may lead to relaxed validation. A backend mismatch may lead to a template patch. In each case, the actual bug may have been introduced earlier during name resolution, instantiation, flattening, or DAE construction.

As a result, substantial manual analysis is still required. Human developers must guide the investigation, inspect compiler artifacts, decide which phase first introduced the wrong symbol, equation, or structural form, and reject plausible but shallow fixes. We therefore treat AI output as a hypothesis, not a fix. In our experience, AI accelerates implementation once the semantic target is clear, but it does not replace human root-cause analysis, specification judgment, or architectural review.

4 Benchmarking and MSL Parity

To quantify how much of the Modelica language Rumoca handles in practice, we benchmark the compiler against the Modelica Standard Library (MSL) version 4.1.0 (Modelica Association 2025), the canonical reference suite for Modelica tools. Rather than report a single pass/fail number, we measure two things: how far each model advances through the pipeline described in section 3 (a reachability funnel that separates front-end coverage from DAE construction, solver lowering, initialization, and simulation), and, for the models that simulate to completion, how closely their trajectories agree with the OpenModelica compiler (OMC).

4.1 Methodology

The benchmark set is the collection of root example models in MSL—every non-partial class of the form `Modelica.<library>.Examples.<...>` with no unbound top-level inputs. This yields 566 standalone-simulatable models spanning the MSL sublibraries. All OMC references were produced with OpenModelica 1.26.7.

Reachability funnel. Each model is run through the full Rumoca pipeline, and we record the last phase it reaches: *Parse* (Parsed IR), *Flatten* (Flattened IR), *DAE* (DAE IR), *Solve* (Solver IR), *IC* (consistent initialization), and *Sim* (integration to the model’s stop time through the embedded Diffsol suite in its `auto` mode, which tries BDF first and falls back through RK-family methods on stiff or discontinuous problems). A model counts as passing a phase if the pipeline gets past that phase without failing; the phases are nested by construction, so the reported rates are non-increasing from left to right across all packages. It should be noted that passing the flattening or DAE phase does not necessarily mean it is correct. Future work will formally verify the semantics of the compiler to ensure the correct construction of the IR and the semantic lowering of it. *Sim* here measures only that a model integrates to completion, not that it agrees with OMC; agreement is measured separately below. Rumoca integrates at solver tolerances $\text{rtol} = \text{atol} = 10^{-6}$.

Matched stop time. Both tools integrate each model to its own `experiment(StopTime=)` annotation: Rumoca reads the annotation directly, and OMC is driven with `--use-experiment-stop-time` so it uses the same horizon. This makes the integration spans match per model, which is a precondition for a meaningful simulation comparison.

Trace agreement. For the models that both tools simulate to completion, parity is measured per output channel with a normalized, bounded error. Channels are variables matched by exact name; each channel is compared on the union of both tools’ sample times over the overlap window, with linear interpolation for continuous channels and zero-order hold for discrete/clocked channels. For a channel, let $\bar{e}$ be the time-averaged absolute deviation between the Rumoca and OMC traces (trapezoidal rule), normalized by a robust scale s equal to the 5th–95th percentile span of that variable’s OMC trace; the channel score $e = \bar{e}/s$ is mapped into $[0, 1]$ by $\hat{e} = e/(1 + e)$, so a single diverging or near-flat channel saturates near 1 rather than dominating the aggregate. A channel is in *high* agreement if $\hat{e} \leq 0.05$, *near* agreement if $\hat{e} \leq 0.20$, and *deviating* otherwise. A model is *agreeing* if its channel distribution falls in either a high-agreement band (at least 80% of channels high and at most 1% deviating) or a near-agreement band (at least 90% high-or-near and at most 10% deviating). Of the 151 models that both tools simulate and that produce comparable traces, 102 ($\approx 67\%$) are agreeing; the remaining third deviate. The agreeing set is the basis for the timing comparison in subsection 4.3.

4.2 Coverage Results

Table 1 reports per-package and overall pass rates at each phase. All 566 models parse cleanly (100% for every sublibrary), so we omit the Parse column and begin at Flatten. Flattening succeeds for 99% of the suite; the only sublibraries below 100% are

Electrical.Polyphase, Electrical.Spice3, Magnetic.FluxTubes, and Utilities.

DAE construction succeeds for 73% of the suite. The losses are concentrated in connection- and constraint-heavy libraries: the quasi-static electrical machines and polyphase libraries (0%), the magnetic wave libraries (FundamentalWave 4%, QuasiStatic.FundamentalWave 10%), Fluid (35%), and the large coupled-constraint systems of Mechanics.MultiBody (55%); StateGraph (0%) is gated by state-machine semantics not yet lowered to a DAE. Media now constructs a DAE for 68% of its examples.

Solver lowering (60%) and initialization (41%) are the next two attrition steps. Electrical.Machines, for example, lowers a Solver IR for 53% of its models but loses all of them at initialization (53% Solve $\rightarrow$ 0% IC), and Magnetic.FluxTubes drops from 85% Solve to 15% IC.

End-to-end, 28% of the suite simulates to its stop time, and—as noted in subsection 4.1—about two-thirds of the simulated models then agree with OMC. There are thus two distinct gaps: a *reachability* gap (only 28% reach a completed simulation) and an *agreement* gap (of those, $\approx$ 33% deviate from OMC). Several libraries are clean end-to-end: ComplexBlocks and the Math examples reach 100%, Electrical.Digital 96%, and Clocked 88%. By contrast, several switched-electronics and thermal libraries reach high DAE and solver coverage but simulate few or no models (Electrical.PowerConverters: 76% Solve, 66% IC, but 0% Sim), indicating gaps in event handling and stiff-solver coverage rather than missing language features. Compile-side phases are fast (parse, flatten, and DAE construction average well under a second per model); wall-clock time is dominated by the solve and simulation phases.

4.3 Compile and Simulation Time

We complement the coverage results with a wall-clock comparison against OpenModelica (OMC), reporting *compilation* and *simulation* time separately rather than as a single combined number. The two tools have opposite performance profiles: Rumoca's advantage is concentrated in compilation, while its interpreted integration loop is slower than OMC's native code on large systems, so a combined figure would hide the crossover. All timings were collected on a single workstation running Ubuntu 24.04 with an AMD Ryzen 9 5950X (16 cores, 32 threads) and 64 GB of RAM.

The harness runs a pool of 14 worker processes, each pinned to a dedicated physical core, that draw models from the 100-model agreeing set (the models that both tools simulate to their `experiment` stop time and on which the traces agree within the bands of subsection 4.1, and for which both tools record valid timings) until the set is exhausted. For each model we measure the time

to build and simulate it from scratch; a model exceeding a 10s timeout is killed and its worker restarted. Each Rumoca worker retains a persistent in-process cache, so intermediate representations of shared MSL components computed for one model may be reused by later models in the same worker. The OMC side is driven through its persistent ZMQ session. We separate three Rumoca stages—front-end through DAE, Solver-IR lowering plus Cranelift JIT build, and the integration loop—and group the first two as *compilation* (build-to-runnable) and the third as *simulation*. For OMC we use the self-reported `timeTotal - timeSimulation` as compilation (frontend, backend, SimCode, C-code generation, and clang) and `timeSimulation` as simulation. We report median per-model time within scalar-equation-count groups, and a throughput (sum/sum) ratio across the whole set.

Table 2 reports the results in three panels. *Compilation* (panel b): Rumoca is faster in every bin ($2.6\text{--}5.2\times$; throughput $4.18\times$, front-end-only median $5.25\times$), because it emits native code through a Cranelift JIT (Bytecode Alliance 2025) rather than generating C and invoking clang. *Simulation* (panel c) crosses over with system size: Rumoca leads on small models ($6.7\times$ at 1–9 equations), reaches parity near 25–49 equations, and then falls behind sharply ($0.29\times$, $0.05\times$, $0.01\times$ in the top three bins; throughput $0.22\times$). OMC's integration is nearly flat ($\approx 0.02\text{--}0.04$ s) because it runs native compiled C, while Rumoca's interpreted Solver-IR runtime grows steeply with size. *Total* (panel a): end to end, Rumoca is faster on all but the very largest systems—median per-model speedup $4.79\times$, throughput $3.01\times$ —and only the 250+ bin (two models) tips to OMC, at $0.54\times$. The breakdown localizes the gain: the speedup is a compile-time effect, and Rumoca's numerical integration is in fact slower than OMC's on large systems.

These numbers should be read as an efficiency comparison, not a like-for-like one. The comparison is restricted to the subset of models that both tools simulate successfully and on which they agree, which by construction excludes the harder cases where OMC's index reduction, tearing, and broader stiff- and event-handling succeed and Rumoca does not; this easy-and-agreeing subset plausibly favors Rumoca, and OMC remains the more robust and capable simulator. Several further issues temper the absolute numbers. The warm, core-pinned worker pool lets Rumoca reuse cached IR for shared MSL components across models, and although OMC is driven through its persistent ZMQ session, we have not verified that it caches MSL components equivalently, so part of the measured compile gap may reflect caching asymmetry. The agreeing population skews small (62 of the 100 models have ≤ 24 equations), so per-model median speedups are small-model-weighted; we report the throughput ratio alongside every median for this reason, and the largest (250+) bin contains only two models. Finally, each model is timed once on a single host with no variance bars; a production

Table 1. MSL 4.1.0 pass rates by package and pipeline phase. Scope: $N = 566$ root `Modelica.*.Examples.*` models. All models pass Parse at 100%, so that column is omitted. A model passes a phase (Flatten, DAE, Solve, IC, Sim) if the pipeline reached past it; *Sim* means the model integrated to its `experiment` stop time, not that it agrees with OMC (trace agreement is reported separately in subsection 4.1). Reported rates are non-increasing left-to-right per row.

MSL Package	n	Flatten	DAE	Solve	IC	Sim
Blocks	32	100%	78%	69%	63%	63%
Clocked	76	100%	96%	96%	88%	88%
ComplexBlocks	2	100%	100%	100%	100%	100%
Electrical.Analog	65	100%	95%	83%	58%	32%
Electrical.Batteries	8	100%	50%	13%	13%	0%
Electrical.Digital	23	100%	96%	96%	96%	96%
Electrical.Machines	43	100%	63%	53%	0%	0%
Electrical.Polyphase	5	80%	80%	80%	20%	20%
Electrical.PowerConverters	59	100%	98%	76%	66%	0%
Electrical.QuasiStatic.Machines	1	100%	0%	0%	0%	0%
Electrical.QuasiStatic.Polyphase	4	100%	0%	0%	0%	0%
Electrical.QuasiStatic.SinglePhase	6	100%	100%	83%	0%	0%
Electrical.Spice3	14	93%	79%	79%	7%	0%
Fluid	23	100%	35%	0%	0%	0%
Magnetic.FluxTubes	20	90%	90%	85%	15%	0%
Magnetic.FundamentalWave	27	100%	4%	4%	0%	0%
Magnetic.QuasiStatic.FluxTubes	9	100%	100%	100%	0%	0%
Magnetic.QuasiStatic.FundamentalWave	21	100%	10%	0%	0%	0%
Math.FastFourierTransform	2	100%	100%	100%	100%	100%
Math.Nonlinear	1	100%	100%	100%	100%	100%
Math.Random	1	100%	100%	100%	100%	100%
Mechanics.MultiBody	42	100%	55%	0%	0%	0%
Mechanics.Rotational	17	100%	100%	100%	41%	18%
Mechanics.Translational	16	100%	100%	100%	81%	69%
Media	22	100%	68%	36%	27%	27%
Media.Incompressible	1	100%	100%	0%	0%	0%
StateGraph	7	100%	0%	0%	0%	0%
Thermal.FluidHeatFlow	12	100%	17%	17%	8%	0%
Thermal.HeatTransfer	4	100%	100%	100%	75%	25%
Utilities	3	67%	67%	67%	67%	67%
Overall	566	99%	73%	60%	41%	28%

claim should repeat the run and report variance, as sub-second compile times are sensitive to scheduling and thermal noise. We therefore present these results as evidence that the Rumoca pipeline is fast enough to be practical, not as a claim of general superiority over a production Modelica tool.

5 Browser-Native Compilation via WebAssembly

A central goal of Rumoca is to make Modelica modeling easy to use. The compiler infrastructure described in section 3 is built so that the WASM bindings crate exposes the compiler’s API as a JavaScript module. Two distinct deployment patterns fall out of this design.

5.1 Compiler-in-Browser

The first pattern runs the compiler itself client-side. A hosted playground at <https://rumoca.cognipilot.org/> (Figure 2) loads the WASM build of `rumoca-bind-wasm` in the user’s browser and uses it to drive an editor with realtime parsing, diagnostics, and DAE introspection on every keystroke. The implementation stack matches the desktop language server described

in subsection 3.4: the same `Session`, the same diagnostic types, the same backend templates.

This is unusual for Modelica tooling. Browser-based Modelica environments typically run the compiler on a remote server and ship results to the client; the user’s models leave their machine, the server bears the operating cost, and editor latency is bounded by network round-trips. By contrast, the Rumoca playground requires no backend, no authentication, no per-session quota. Models stay on the user’s machine. The operator of the playground pays only for static asset hosting. This is particularly valuable to researchers, students, and engineers evaluating Modelica without committing to a desktop installation, lowering the barrier to first contact.

5.2 Standalone HTML Simulators

Rumoca includes JavaScript templates that compile Modelica models into pure, self-contained JavaScript code, which can be embedded directly in an HTML file (Figure 3). This enables lightweight, browser-based simulators and makes Rumoca suitable for applications such as digital twins, product model exchange, and interactive simulation demos on web pages.

The js-template-based approach exposes the solver and

Table 2. Wall-clock time, Rumoca vs. OpenModelica (OMC), on the 100 models that both tools simulate to their experiment stop time and agree within the parity thresholds of subsection 4.1, grouped by scalar-equation count. *Compilation* is build-to-runable (timeTotal – timeSimulation for OMC, including clang); *Simulation* is the integration loop only. Per-bin Rumoca/OMC columns are median seconds and Speedup is the per-bin median of per-model ratios; the *All* row reports the throughput (sum/sum) ratio over all 100 models, which need not equal any median.

Scalar eqns	Models	Rumoca (s)	OMC (s)	Speedup (×)
<i>(a) Total: model → simulated result</i>				
1–9	26	0.46	2.40	5.20
10–24	36	0.49	2.44	5.03
25–49	11	0.54	2.48	4.55
50–99	14	0.80	2.68	3.37
100–249	11	1.33	2.75	2.07
250+	2	6.05	3.29	0.54
All	100	84.9	255.9	3.01
<i>(b) Compilation: build-to-runable</i>				
1–9	26	0.46	2.38	5.18
10–24	36	0.48	2.42	5.10
25–49	11	0.50	2.45	4.88
50–99	14	0.55	2.58	4.72
100–249	11	0.97	2.73	2.81
250+	2	1.24	3.26	2.64
All	100	59.8	250.3	4.18
<i>(c) Simulation: integration only</i>				
1–9	26	0.0028	0.0190	6.71
10–24	36	0.0100	0.0205	2.06
25–49	11	0.0255	0.0252	0.99
50–99	14	0.1376	0.0399	0.29
100–249	11	0.4889	0.0240	0.05
250+	2	4.8140	0.0292	0.01
All	100	25.1	5.6	0.22

code-generation logic in a form that is easy for AI coding agents to inspect, modify, and extend. Existing examples (Meschede 2026) demonstrate how compilation targets can be switched quickly between different architectures and programming languages or build rich interfaces on top of detailed simulation models.

6 Case Studies

This section presents two end-to-end case studies, each taking a single Modelica source through the Rumoca toolchain. 6.1 drives a 21-state quadrotor as a realtime SIL plant for an external autopilot, exercising the native Rust runtime. 6.2 takes a small planar quadrotor and emits it into four algebraic ecosystems from one source, exercising the code generators. Together they support the paper’s thesis: Modelica as a universal algebraic frontend, authored once and consumed by downstream optimization, symbolic analysis, differentiable simulation, and tool-independent deployment in their native idioms.

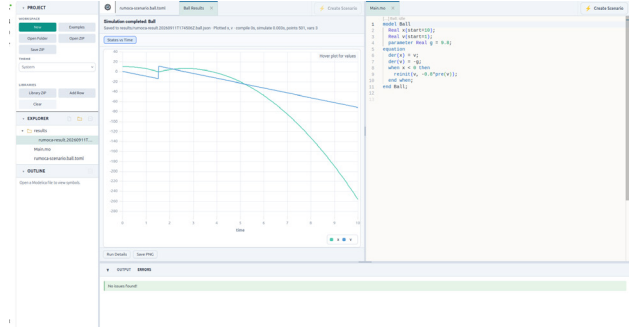


Figure 2. The Rumoca in-browser playground. The compiler runs entirely client-side via WebAssembly, with parsing, diagnostics, and DAE inspection driven by the same Session API used by the desktop CLI and language server.

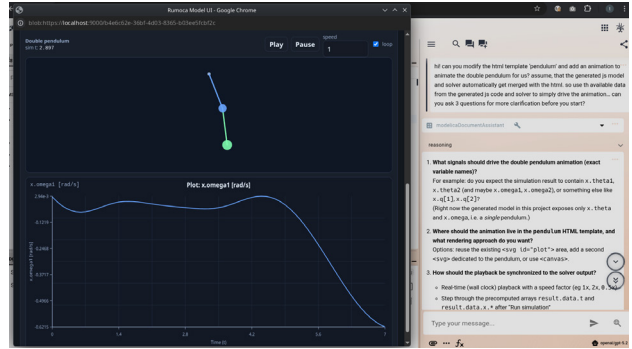


Figure 3. Modelica simulations can be exported into fully self-contained, pure HTML/js based and customizable files using rumoca with its js based templates.

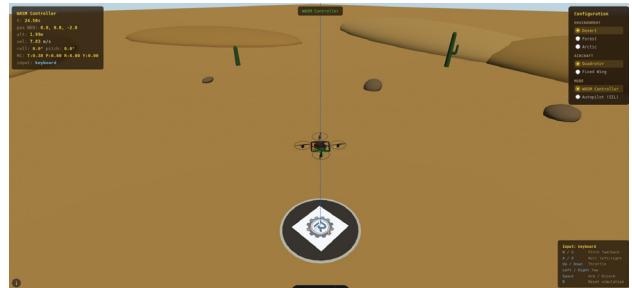


Figure 4. Software-in-the-loop setup: Rumoca runtime as real-time plant for Cerebri.

6.1 Realtime Software-in-the-Loop with Gamepad Control

The first case study runs Rumoca's native simulation runtime as a realtime SIL plant for the CogniPilot Cerebri autopilot (CogniPilot Foundation 2025). The Modelica source is a 21-state quadrotor: six-degree-of-freedom rigid-body kinematics with a rotation-matrix attitude representation with Euler's equations for the angular dynamics

The runtime advances the integrator in lock-step with the autopilot, exchanging named-signal frames with Cerebri over UDP through a FlatBuffers codec at a fixed step rate of 250 Hz. A USB gamepad feeds the autopilot's setpoint channel, so the operator flies the simulated vehicle through the controller. A Three.js scene renders the state in a browser at the step rate, and a small WebSocket proxy bridges the UDP feed for remote operators. Figure 4 shows the setup.

The point is not realtime control quality. It is that the same compiler producing the symbolic artifacts of 6.2 also produces a Rust binary deterministic enough to hold a step-locked loop against a production autopilot, without giving up the equation-based modelling surface. The only contract between the model, the SIL host, the autopilot, and the viewer is the Modelica file and the signal schema.

6.2 Same Source, Multiple Algebraic Backends

The second case study drives four downstream workflows from one hand-written Modelica source: a planar vertical-take-off-and-landing aircraft (PVTOL), the canonical two-input textbook quadrotor. Its six states are the horizontal position x , the altitude z , the pitch angle θ , and their time derivatives $\dot{x}$, $\dot{z}$, $\dot{\theta}$; its two inputs are the total thrust T and the pitching moment M ; and its parameters are the mass m , the moment of inertia about the pitch axis J , and the gravitational acceleration g . Listing 2 gives the complete source. It is about twenty lines, small enough that the symbolic deliverable of 6.2.2 fits on the page and can be checked against a textbook. Hover trim is $\theta = 0$, $T = mg$, $M = 0$.

Listing 2. The complete PVTOL Modelica source. This one file feeds all four backends of 6.2.

```

model PVTOL "Planar VTOL aircraft"
  parameter Real m = 2.496 "Mass [kg]"
  ;
  parameter Real J = 0.0344 "Inertia
    about y [kg.m2]"
  parameter Real g = 9.80665 "Gravity [m
    /s2]"
  Real x, z, theta "Position
    , pitch"
  Real x_dot, z_dot, theta_dot "
    Velocities"
  input Real T "Total thrust [N]"
  input Real M "Pitching moment [N.m]"
equation

```

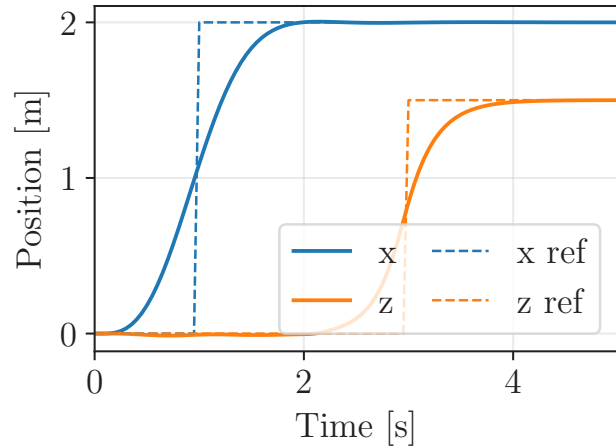


Figure 5. Linear MPC built on the Rumoca-emitted CasADi model tracking a step reference in the horizontal position x and altitude z . The prediction model is the hover linearization; the plant is the nonlinear emitted model.

```

der(x)      = x_dot;
der(z)      = z_dot;
der(theta)  = theta_dot;
m*der(x_dot) = -T*sin(theta);
m*der(z_dot) = T*cos(theta) - m*g
;
J*der(theta_dot) = M;
end PVTOL;

```

From this one source Rumoca emits four artifacts: a CasADi MX module; a SymPy Model class; a JAX module exposing the right-hand side as a function $\text{ode_fn}(t, y, \text{args})$, with time t , state vector y , and an argument tuple args holding the parameters and inputs; and an FMI 3.0 model-exchange FMU. Each artifact comes from one Rumoca compilation; the consumer scripts below import it as an ordinary module or FMU and never touch generated code.

6.2.1 Linear MPC with CasADi

We differentiate the CasADi-emitted state derivative at the hover trim to obtain the same hover linearization of 6.2.2, and use it as the prediction model of a linear model-predictive controller (MPC): forward Euler at 20 Hz over a 1.0 s horizon, with the thrust T and moment M bounded, posed as a quadratic program (QP) through CasADi's Opti interface and solved. The controller runs against the nonlinear emitted plant and tracks a step reference (Figure 5), with a mean QP solve time near 10 ms, inside the control period. The glue code is plant-agnostic: swapping the Modelica source re-derives the linearization automatically.

6.2.2 Symbolic Linearization and Cross-Backend Check with SymPy

The SymPy backend exposes the dynamics symbolically, so the Jacobians follow in closed form. At hover, the state

matrix A and input matrix B are

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -g & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{1}{m} & 0 \\ 0 & \frac{1}{J} \end{bmatrix}, \quad (1)$$

matching the textbook PVTOL linearization, and SymPy certifies that the controllability matrix $[B, AB, \dots, A^5B]$ has full rank 6 symbolically, independent of the parameters.

The same script then checks the backends against each other. We build CasADi's analytic Jacobians from the same source, lambdify SymPy's, and evaluate both at five random points in the state and input space. The maximum elementwise difference is 2.8×10^{-16} for A and 3.6×10^{-14} for B . Two backends emitted from one `.mo` file by separate templates encode numerically identical dynamics, the guarantee a hand-ported model cannot give.

6.2.3 Parameter Identification with JAX

The JAX module exposes the dynamics as the function `ode_fn` introduced above, so it composes with `jit`, `grad`, and `vmap` and slots into an adjoint-mode integrator. This makes the whole trajectory differentiable with respect to the model's physical parameters m and J .

We build a reference trajectory under the ground-truth values $m = 2.496 \text{ kg}$ and $J = 0.0344 \text{ kg m}^2$ with a thrust-step, sinusoidal-moment excitation over three seconds, then start from a poor initial guess for the mass and inertia ($\hat{m}_0 = 0.50m$, $\hat{J}_0 = 0.60J$, where $\hat{m}$ and $\hat{J}$ are the estimates) and minimize the mean-squared trajectory error by gradient descent on $(\log \hat{m}, \log \hat{J})$. After 60 Adam iterations the recovered values are $\hat{m} = 2.4985 \text{ kg}$ (+0.10%) and $\hat{J} = 0.034544 \text{ kg m}^2$ (+0.42%), with the loss falling by a factor of 5.5×10^5 (Figure 6). The consumer code never reimplements the dynamics or derives a sensitivity equation by hand; the adjoint integration comes from the emitted module.

6.2.4 FMI 3.0 Export

The FMI 3.0 backend produces a tool-independent FMU: a zip with the `modelDescription.xml` interface, a compiled shared object, and the source manifest. As of now, the FMU has only been tested with gcc on linux. Loaded with `fmpy`, it holds exact hover trim ($z(2s) = 3.8 \times 10^{-16} \text{ m}$) and, under a 10% over-thrust $T = 1.10mg$, follows the closed-form ballistic climb to $8 \times 10^{-15} \text{ m}$. Each step costs about $25 \mu\text{s}$ at 200 Hz, roughly $200\times$ faster than real time, leaving headroom to host the FMU inside a larger co-simulation rig.

Across the four workflows the modeller wrote one short Modelica source; the consumer chose the artifact, and SymPy certified that two of the bindings agree to machine precision.

7 Roadmap and Discussion

Rumoca turns Modelica into a frontend: one source, authored once, emitted into whatever ecosystem the work demands—CasADi, JAX, SymPy, FMI, or a native Rust runtime—with cross-backend agreement to machine precision that a hand-ported model cannot match. Two lines of work carry that thesis forward.

The first is coverage. End-to-end MSL simulation is 28%, leaving room for improvement. We are taking a structured defensive coding approach to address this and focusing on built-in tracing tools that assist authors diagnose simulation and compile time errors back to source code through span preservation. The second is templates. Because code generation is template-driven, the reach of Rumoca grows by adding and hardening backends rather than rewriting the compiler. The near-term work is promoting the experimental Embedded C target to production, broadening FMI 3.0 coverage, and bringing the JAX, Julia, and ONNX templates to the maturity of the CasADi and SymPy paths, each one extending where a single Modelica model can be deployed.

We also note that MSL parity, having served as our development and regression corpus, functions more as a language-coverage test than as a measure of real-world utility, which the case studies of Section 6 address only from author-written models. No Modelica collection is curated for the optimization and machine-learning workloads Rumoca targets, so we invite the community to contribute models that already simulate in an established compiler; each such model is a known reference against which Rumoca's coverage and cross-backend agreement can be measured. Feedback from these early adopters will steer coverage priorities more directly than MSL parity alone.

Acknowledgements

This material is based on research sponsored by DARPA under agreement number FA8750-24-2-0500. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

- Agosta, Giovanni, Francesco Casella, Daniele Cattaneo, Stefano Cherubin, Alberto Leva, Michele Scuttari, and Federico Terraneo (2023). "MARCO: An Experimental High-Performance Compiler for Large-Scale Modelica Models". In: *Proceedings of the 15th International Modelica Conference*. Aachen, Germany, pp. 13–22. DOI: 10.3384/ecp20413.
- Åkesson, Johan, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit (2010). "Modeling and optimization with Optimica and JModelica.org — Languages and tools for solving large-scale dynamic optimization problems". In: *Computers & Chemical Engineering* 34.11, pp. 1737–1749. DOI: 10.1016/j.compchemeng.2009.11.011.

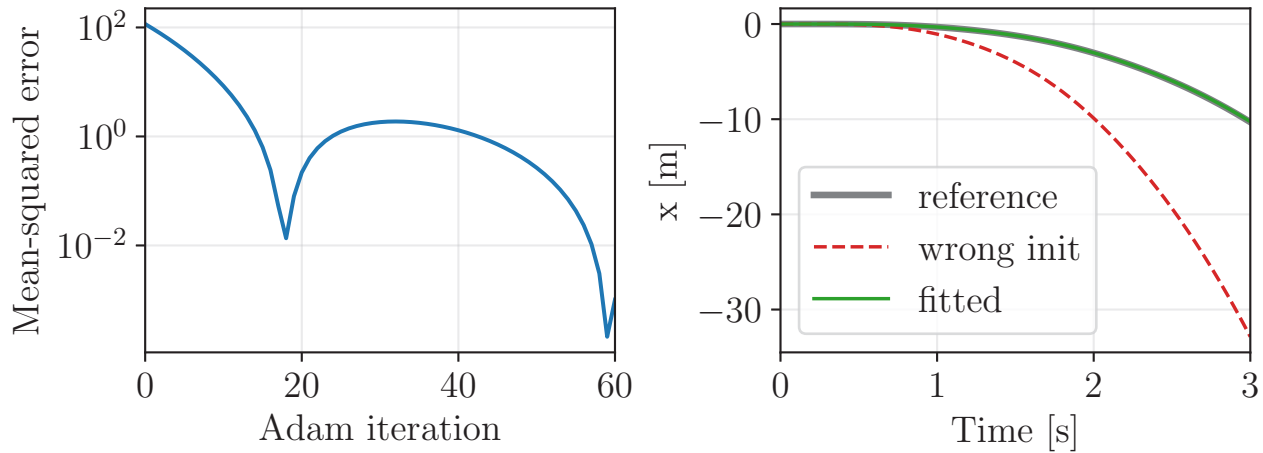


Figure 6. Parameter identification through a differentiable rollout of the Rumoca-emitted JAX model. Left: the mean-squared trajectory error over 60 Adam iterations. Right: the reference, the diverging wrong-initialization, and the fitted trajectory in the horizontal position x .

- Andersson, Joel A. E. (2023). “Import and Export of Functional Mockup Units in CasADi”. In: *Proceedings of the 15th International Modelica Conference*. Aachen, Germany, pp. 321–326. DOI: 10.3384/ecp204321.
- Andersson, Joel A. E., Joris Gillis, Greg Horn, James B. Rawlings, and Moritz Diehl (2019). “CasADi: A Software Framework for Nonlinear Optimization and Optimal Control”. In: *Mathematical Programming Computation* 11.1, pp. 1–36. DOI: 10.1007/s12532-018-0139-4.
- Andersson, Joel A. E. and James M. Goppert (2024). “Event Support for Simulation and Sensitivity Analysis in CasADi for use with Modelica and FMI”. In: *Proceedings of the American Modelica Conference 2024*. Storrs, Connecticut, USA, pp. 99–108. DOI: 10.3384/ecp20799.
- Bytecode Alliance (2025). *Craneflirt Code Generator*. URL: <https://craneflirt.dev>.
- CogniPilot Foundation (2025). *CogniPilot: Open-Source Autonomy Stack*. URL: <https://cognipilot.org>.
- Condie, Micah K., Abigaile Woodbury, James M. Goppert, and Joel A. E. Andersson (2025). “Rumoca: Towards a Translator from Modelica to Algebraic Modeling Languages”. In: *Proceedings of the 16th International Modelica & FMI Conference*. Lucerne, Switzerland, pp. 1009–1016. DOI: 10.3384/ecp2181009.
- Dassault Systèmes (2025). *Dymola*. Originally developed by Dynasim AB. URL: <https://www.3ds.com/products/catia/dymola>.
- Fritzson, Peter et al. (2020). “The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development”. In: *Modeling, Identification and Control* 41.4, pp. 241–295. DOI: 10.4173/mic.2020.4.1.
- Google (2025). *FlatBuffers: Memory Efficient Serialization Library*. URL: <https://flatbuffers.dev>.
- JuliaHub (2025). *Dyad: A New Language to Make Hardware as Easy as Software*. Formerly JuliaSim. URL: <https://juliahub.com/blog/dyad-making-hardware-as-easy-as-software>.
- Ma, Yingbo, Shashi Gowda, Ranjan Anantharaman, Chris Laughman, Viral Shah, and Christopher Rackauckas (2021). *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling*. DOI: 10.48550/arXiv.2103.05244. arXiv: 2103.05244 [cs.LG].
- Meschede, Thomas (2026). *Taskyon: An Open-Source Agent Interface Integrating Rumoca*. URL: <https://github.com/Xyntopia/taskyon> (visited on 2026-05-13).
- Modelica Association (2017). *Modelica® – A Unified Object-Oriented Language for Systems Modeling, Language Specification Version 3.4*. URL: <https://specification.modelica.org/maint/3.4/MLS.html>.
- Modelica Association (2022). *Functional Mock-up Interface Specification, Version 3.0*. FMI Modelica Association Project; latest maintenance release 3.0.2 (2024). URL: <https://fmi-standard.org/docs/3.0/>.
- Modelica Association (2023). *Base Modelica Requirements — MCP-0031*. URL: <https://github.com/modelica/ModelicaSpecification/blob/MCP/0031/RationaleMCP/0031/Base-Modelica-requirements.md> (visited on 2026-05-30).
- Modelica Association (2025). *Modelica Standard Library, Version 4.1.0*. Released 2025-05-23. URL: <https://github.com/modelica/ModelicaStandardLibrary/releases/tag/v4.1.0>.
- Pallets (2025). *Jinja: A Very Fast and Expressive Template Engine*. URL: <https://jinja.palletsprojects.com>.
- Robinson, Martin and Alex Allmont (2026). “diffsol: Rust crate for solving differential equations”. In: *Journal of Open Source Software* 11.117, p. 9384. DOI: 10.21105/joss.09384.
- Singer, Jörg (2025). *Introduction to the Parol Parser Generator*. URL: <https://jsinger67.github.io/Introduction.html> (visited on 2025-05-02).
- The Rust Project Developers (2025). *Rust Compiler Development Guide*. URL: <https://rustc-dev-guide.rust-lang.org>.

Differentiable Parameter Optimization for DAEs with State-Dependent Events

Ion Matei¹ Maksym Zhenirovskyy¹ Hon Yung Wong¹

¹Fujitsu Research of America, {imatei,mzhenirovskyy,awong}@fujitsu.com

Abstract

Equation-oriented modeling tools routinely generate differential-algebraic equations (DAEs) with state-dependent events, including switching logic, mode changes, impacts, and `when/reinit`-style state reinitializations. While mature DAE solvers provide robust forward simulation for such hybrid models, gradient-based parameter estimation remains difficult: algebraic variables are implicit, the times of state events depend on the parameters, and resets are discontinuous. We formulate parameter learning as constrained least squares and compare two gradient strategies. An AD-through-simulation method differentiates the algebraic solve by the implicit function theorem and uses segmented differentiable integration. An explicit discrete-adjoint method constructs smooth-segment and event residuals from an event-split trajectory and solves for their Lagrange multipliers; these residuals are equality constraints, not heuristic penalties. Experiments on one event-driven DAE and one ODE impact system illustrate the tradeoffs. Both methods compute local gradients for the event path selected by the forward simulation and assume fixed event ordering and transversal guard crossings.

1 Introduction

Equation-oriented modeling languages such as Modelica provide a compact and declarative way to describe multi-domain systems as equations, connections, parameters, and event logic. Toolchains then perform structural analysis, index reduction, algebraic-loop handling, event detection, and integration. Consequently, DAEs with events are central to equation-oriented simulation (Ascher and Petzold, 1998; Fritzson, 2014; Hairer and Wanner, 1996; Modelica Association, 2025).

Switches, mode changes, impacts, and `when/reinit` statements make the model a sequence of smooth DAE segments coupled by guards, resets, and post-event algebraic consistency.

Hard switches may instead be regularized or suppressed with Modelica’s `smooth` and `noEvent` operators, at the possible cost of approximation error, slower simulation, or reduced model reuse (Modelica Association, 2023). Here the original guard/reset structure is retained; only output reconstruction is optionally blended during optimization (Section 3.1). “Times of the events” below means

parameter-dependent *state-event* times, not Modelica *time events*, whose times are known in advance (Modelica Association, 2023).

Existing Modelica and DAE toolchains provide mature and robust mechanisms for forward simulation. DASSL and SUNDIALS/IDA provide implicit integration and nonlinear solves, while Modelica tools add equation sorting, algebraic-loop detection, and tearing (Casella, 2015; Elmquist and Otter, 1994; Hindmarsh et al., 2005; Petzold, 1982; Täuber et al., 2014; Zimmer, 2009). These capabilities do not automatically provide end-to-end derivatives through events, resets, and algebraic consistency.

Section 6.1 outlines a Modelica/FMI deployment path; the present experiments are Python-only and do not benchmark a complete Modelica toolchain.

Libraries such as `torchdiffeq` and `diffra` support differentiable ODE solving and event-aware examples (Chen, 2018; Chen et al., 2021, 2018; Kidger, 2021). They primarily target ODEs; a DAE implementation must also solve and differentiate algebraic variables during integration, after events, and during output reconstruction.

We compare two strategies. AD-through-simulation solves the algebraic equations inside the vector field, differentiates that solve by the implicit function theorem, and uses segmented integration with right-continuous output evaluation.

The explicit discrete adjoint instead consumes an external solver’s event-split trajectory and constructs smooth DAE and event residuals. These are equality constraints, not heuristic penalties; their multipliers give the gradient without forming full trajectory or event-time sensitivity matrices.

The contributions are:

1. formulate parameter learning for event-driven DAEs as a constrained least-squares problem with algebraic constraints, event guards, and reset maps;
2. present an AD-through-simulation implementation in which algebraic variables are differentiated by an implicit-function-theorem custom JVP and events are handled by segmented differentiable integration;
3. derive an explicit discrete-adjoint formulation that can use an external forward DAE solver and computes gradients from an event-split residual system;
4. compare them on an electrical DAE and a planar bouncing-ball system.

Only the Cauer benchmark exercises the DAE algebraic solve; the multi-ball case is an ODE-with-events scaling study. Thus the experiments do not establish scaling with DAE index, algebraic-loop size, or tearing.

Both approaches compute gradients that are local to the event path selected by the forward simulation. They assume transversal event crossings and no change in the number, identity, or ordering of events under sufficiently small parameter perturbations. This local-path interpretation is consistent with hybrid-system sensitivity analysis, where perturbations are propagated across events using jump conditions or saltation matrices (Saccon et al., 2014; Kong et al., 2024). These assumptions also clarify the limitations of gradient-based optimization near grazing events, simultaneous events, or event-order changes.

The implementation used for the experiments is available at <https://github.com/ionmatei/diff-dae-events/>.

Paper structure: Section 2 formulates parameter learning for DAEs with state-dependent events as a constrained least-squares problem. Section 3 presents the AD-through-simulation approach. Section 4 develops the explicit discrete-adjoint formulation. Section 5 reports the numerical experiments. Section 6 discusses related work and implementation tradeoffs. Section 7 concludes the paper. Full derivations and implementation details omitted for space are retained in the extended arXiv version (Matei et al., 2026).

2 Problem Formulation

Let

$$0 = \tau_0 < \tau_1 < \dots < \tau_M < \tau_{M+1} = T$$

denote the event times generated by the hybrid DAE. On each interval (τ_m, τ_{m+1}) , the system satisfies

$$\dot{x}_m(t) = f(t, x_m(t), z_m(t), p), \quad 0 = g(t, x_m(t), z_m(t), p).$$

At an event time τ_m , an active guard satisfies

$$\phi_{e_m}(\tau_m, x_m^-, z_m^-, p) = 0,$$

and a reset map or reinitialization equation relates the pre-event and post-event variables,

$$(x_m^+, z_m^+) = \Psi_{e_m}(\tau_m, x_m^-, z_m^-, p),$$

with the understanding that algebraic variables may be recomputed after the differential state is reset.

A hat denotes a prediction, a star a reference value, $-/+$ the event's left and right limits, and $(\cdot)^\top$ transpose. The subscript h denotes a discrete quantity: $\mathcal{J}_h$ is the discrete objective, $R_{m,k}$ a smooth step residual, E_m an event residual, W all discrete states, and $\lambda_{m,k}, \mu_m$ their respective multipliers.

Given output data y_i^{data} at observation times t_i , the parameter-estimation problem can be written as

$$\begin{aligned} \min_{p, \{x_m, z_m\}, \{\tau_m\}} \quad & \frac{1}{2} \sum_{i=1}^{N_{\text{obs}}} \|h(t_i, x(t_i), z(t_i), p) - y_i^{\text{data}}\|_2^2 \\ \text{s.t.} \quad & \dot{x}_m(t) - f(t, x_m(t), z_m(t), p) = 0, \\ & g(t, x_m(t), z_m(t), p) = 0, \\ & E_m(\tau_m, w_m^-, w_m^+, p) = 0, \quad m = 1, \dots, M, \end{aligned}$$

where $w = (x, z)$ and E_m collects the guard condition, reset equations, continuity conditions for variables not modified by the event, and post-event algebraic consistency.

This constrained formulation is the common starting point for both methods studied in this paper. The AD-through-simulation method eliminates the trajectory variables by executing the simulator and differentiating the reduced map from parameters to outputs. The discrete-adjoint method keeps the residual constraints explicit after discretization and solves for the corresponding Lagrange multipliers.

3 AD-through-Simulation for DAEs with Events

The AD-through-simulation approach treats the event-driven DAE simulator as a differentiable program. For a selected parameter vector p_{opt} , the simulator produces predicted outputs

$$\hat{Y}(p_{\text{opt}}) = \begin{bmatrix} \hat{y}(t_1; p_{\text{opt}}) \\ \vdots \\ \hat{y}(t_N; p_{\text{opt}}) \end{bmatrix}.$$

The reduced learning problem is then

$$\min_{p_{\text{opt}}} \widehat{\mathcal{J}}(p_{\text{opt}}) = \mathcal{J}(\hat{Y}(p_{\text{opt}})),$$

and the gradient is computed by automatic differentiation:

$$\nabla_{p_{\text{opt}}} \widehat{\mathcal{J}} = \text{AD} \left[\mathcal{J}(\hat{Y}(p_{\text{opt}})) \right].$$

The main DAE-specific issue is the algebraic variable. During integration, z is defined implicitly by

$$g(t, x, z, p) = 0.$$

We write the solution as

$$z = \zeta(t, x, p), \quad g(t, x, \zeta(t, x, p), p) = 0.$$

The differential state is advanced using the reduced vector field

$$\dot{x} = f(t, x, \zeta(t, x, p), p).$$

In the implementation, the algebraic solve is performed by chord Newton iterations and differentiated using a custom JVP derived from the implicit function theorem. Thus,

gradients propagate through the algebraic constraint without differentiating through every Newton iteration.

For completeness, write $g_q = \partial g / \partial q$, with all partial derivatives evaluated at $(t, x, \zeta(t, x, p), p)$. A perturbation (dt, dx, dp) induces dz satisfying

$$g_z dz + g_t dt + g_x dx + g_p dp = 0,$$

and hence the Jacobian-vector product (JVP) used by the implementation is

$$dz = -g_z^{-1}(g_t dt + g_x dx + g_p dp).$$

This expression assumes that g_z is nonsingular at the evaluated point. In practice the linear system is solved; the inverse is not formed explicitly.

Events are handled by segmented integration. At the beginning of each segment, all guard functions are evaluated and guards that are already active are masked out to avoid immediate retriggering. The remaining active guards are combined into a scalar event condition,

$$\Phi(t, x, p) = \min_{e \in \mathcal{A}} \phi_e(t, x, \zeta(t, x, p), p),$$

where $\mathcal{A}$ is the set of active guards. A root finder locates the first zero crossing of this composite guard. The corresponding event index is selected, the reset map is applied, the algebraic variables are recomputed, and the next segment starts from the post-event state.

To make this process compatible with JIT compilation, the simulator is written as fixed-length scans over a prescribed maximum number of event-bounded segments. Physical segments are recorded until the terminal time is reached; remaining iterations are padding segments. Target-time evaluation is performed by reintegrating each recorded segment with the requested target times clipped to that segment. The output at each target time is selected from the latest real segment whose closed interval contains the target, giving the right-continuous convention

$$\hat{y}(\tau_m) = h(\tau_m, x(\tau_m^+), z(\tau_m^+), p).$$

3.1 Differentiable target-time blending

The Cauer experiment uses optional soft segment selection to widen the loss basin. Both routes use the same *output reconstruction* surrogate; the DAE dynamics, guard localization, and reset map themselves are not smoothed. Let segment k occupy $[a_k, b_k]$, let $r_k \in \{0, 1\}$ distinguish physical from padded segments, and let $X_{k,i}$ be the state obtained by evaluating segment k at observation time t_i clipped to $[a_k, b_k]$. Define

$$\omega_{k,i} = \sigma(\beta(t_i - a_k)) \sigma(\beta(b_k - t_i)) r_k, \\ \bar{\omega}_{k,i} = \frac{\omega_{k,i}}{\sum_{\ell} \omega_{\ell,i} + \varepsilon_{\omega}}, \quad \hat{x}_{\beta}(t_i; p) = \sum_k \bar{\omega}_{k,i} X_{k,i},$$

where $\sigma(s) = 1/(1 + e^{-s})$, $\beta > 0$ controls transition sharpness, and $\varepsilon_{\omega} > 0$ prevents division by zero. Outputs are

reconstructed via $\hat{z}_{\beta}(t_i; p) = \zeta(t_i, \hat{x}_{\beta}(t_i; p), p)$ and $\hat{y}_{\beta} = h(t_i, \hat{x}_{\beta}, \hat{z}_{\beta}, p)$. Finite β blends neighboring segment values; away from events the weights approach hard selection as $\beta \rightarrow \infty$.

For Cauer, both optimizers use the blended objective with $\beta = 150$. Reported Cauer losses are re-evaluated with hard, right-continuous segment selection using a common IDA forward solve. The values compared in Table 1 are therefore not blended metrics. The surrogate can introduce optimization bias for finite β ; the present experiment does not quantify that bias separately.

The resulting loss and gradient are evaluated with a JIT-compiled `value_and_grad` call. These gradients pass through the algebraic solve, ODE integration, event-time computation, reset maps, target-time selection, and output reconstruction. They are local to the event sequence selected by the forward simulation.

A useful way to interpret this method is that it differentiates the numerical algorithm actually executed by the simulator. This includes solver steps, algebraic solves, event-time refinement, reset maps, and target-time selection. The advantage is that the gradient is tightly coupled to the solver tolerance and implementation used for the forward pass. The drawback is that reverse-mode AD must retain or reconstruct the computational trace of the segmented simulation. This can become expensive when the number of states, events, or solver steps grows. Further derivations are in the extended arXiv version (Matei et al., 2026).

4 Discrete-Adjoint Parameter Learning for DAEs with Events

The AD-through-simulation approach requires a simulator whose numerical operations are visible to the automatic-differentiation system. Many mature DAE solvers, however, are available only as external numerical engines. The discrete-adjoint approach targets this setting. It uses the external solver to generate the event-split forward trajectory at the current parameter values, including the event times and pre- and post-event states. These quantities are then used to construct a differentiable discrete residual system, from which the Lagrange multipliers and parameter gradient are computed explicitly.

The discrete-adjoint method starts from the same constrained parameter estimation problem, but keeps the constraints explicit after discretization. The forward solve produces an event-split trajectory: smooth segments between events and event records containing the event time, active event index, pre-event state, and post-event state.

Let segment m be bounded by event times τ_m and τ_{m+1} . The implementation stores normalized time coordinates

$$0 = \eta_{m,0} < \eta_{m,1} < \dots < \eta_{m,N_m-1} = 1,$$

and reconstructs physical node times as

$$t_{m,k} = \tau_m + \eta_{m,k}(\tau_{m+1} - \tau_m).$$

Thus, event times determine both the segment boundaries and the physical time steps used in the residuals.

For $w_{m,k} = [x_{m,k}^\top, z_{m,k}^\top]^\top$, define the column-width abbreviations

$$f_{m,k} = f(t_{m,k}, x_{m,k}, z_{m,k}, p),$$

$$g_{m,k} = g(t_{m,k}, x_{m,k}, z_{m,k}, p),$$

$$h_{m,k} = t_{m,k+1} - t_{m,k}.$$

The smooth-step residual is then the compact trapezoidal DAE residual

$$R_{m,k} = \begin{bmatrix} -x_{m,k+1} + x_{m,k} + \frac{h_{m,k}}{2}(f_{m,k} + f_{m,k+1}) \\ g_{m,k+1} \end{bmatrix} = 0.$$

This form is mathematically identical to the expanded expression but fits within one column of the conference template. Event blocks are represented by residuals of the form

$$E_m(\tau_m, w_m^-, w_m^+, p) = 0,$$

which enforce the guard condition, reset equations, continuity of unchanged variables, and algebraic consistency after the event.

The finite-dimensional constrained problem is therefore

$$\begin{aligned} \min_{W, \tau, p} \quad & \mathcal{J}_h(W, \tau, p) \\ \text{s.t.} \quad & R_{m,k}(W, \tau, p) = 0, \\ & E_m(W, \tau, p) = 0, \end{aligned}$$

where W denotes the collection of the differential-state and algebraic-variable values at all mesh nodes. Problem (1) is not a penalty relaxation. It is the discrete constrained form of the same parameter-estimation problem. The corresponding discrete Lagrangian is

$$\begin{aligned} \mathcal{L}_h = \mathcal{J}_h(W, \tau, p) &+ \sum_{m,k} \lambda_{m,k}^\top R_{m,k}(W, \tau, p) \\ &+ \sum_m \mu_m^\top E_m(W, \tau, p). \end{aligned}$$

Because the residuals vanish on feasible trajectories, $\mathcal{L}_h = \mathcal{J}_h$ for any multipliers. The multipliers are chosen so that the stationarity conditions with respect to W and τ eliminate the need to compute explicit trajectory and event-time sensitivities. The gradient is then obtained from

$$\frac{d \mathcal{J}_h}{dp} = \frac{\partial \mathcal{L}_h}{\partial p}.$$

Computationally, the gradient is evaluated by a reverse block sweep. Smooth segment blocks solve local adjoint systems associated with the trapezoidal DAE residuals. Event blocks solve event-adjoint equations associated with the guard and reinitialization residuals. Because event times define adjacent segment boundaries, event-time stationarity couples an event block with the segment immediately before it. The implementation handles this by storing an affine pending-event adjoint and resolving

its scalar coefficient when the preceding segment is swept. The discrete-adjoint route trades implementation convenience for control over the gradient calculation. Its accuracy is tied to the residual discretization used for the adjoint calculation, not necessarily to the tolerance of the forward DAE solver. This distinction explains why a high-accuracy forward solve does not automatically imply a high-accuracy discrete-adjoint gradient: the adjoint residual mesh and quadrature must be sufficiently accurate for the optimization problem. Conversely, because the method does not require differentiating through the solver internals, it can be paired with mature external DAE solvers and can expose smaller linear systems when symbolic preprocessing or tearing is available. The complete block derivation is in the extended arXiv version (Matei et al., 2026).

An untested adaptive extension would retain detected event times as segment boundaries and refine each segment's interior trapezoidal mesh until successive parameter gradients agree. Forward IDA tolerances control the stored trajectory and event localization, but do not refine this separate adjoint mesh.

5 Experiments

We evaluate the proposed gradient pipeline on two hybrid-system parameter identification problems. The first is an electrical ladder network with a state-dependent reinit event and is used to compare the implicit-AD route (diffraX with a JAX-AD-friendly algebraic solver) against the discrete-adjoint route (Sundials/IDA forward solve, padded backward sweep for computing the multipliers). The second is a planar multi-ball system with contact-driven reinit events; we use it to study how each route scales with the number of balls N and to position our JAX-AD pipeline against a PyTorch-AD implementation. Only the first benchmark exercises an algebraic solve; the second is explicitly an ODE-with-events scaling study. The code for generating the experimental results can be found at <https://github.com/ionmatei/diff-dae-events/>.

5.1 Cauer ladder with a state-dependent event

Here $x = (i_{L_2}, i_{L_1}, v_{C_5}, v_{C_3}, v_{C_1})$ and $z = (\dot{v}_{C_5}, \dot{v}_{C_3}, \dot{v}_{C_1})$; the seven optimized positive parameters are $\theta = (C_1, C_2, C_3, C_4, C_5, L_1, L_2)$. A when clause detects $v_{C_3} > 0.5$ V and applies $\text{reinit}(v_{C_3}, 0)$. Its complete event residual is

$$E_C(\tau_C, w^-, w^+, p) = \begin{bmatrix} 0.5 - v_{C_3}^- \\ v_{C_3}^+ \\ P_{-3}(x^+ - x^-) \\ g(\tau_C, x^+, z^+, p) \end{bmatrix} = 0,$$

where P_{-3} selects the four differential states other than v_{C_3} ; the last block restores post-event algebraic consistency. The 20-second nominal trajectory has 10 events and 11 smooth segments. We initialize the parameters from

the truth via a log-uniform multiplicative bias of half-width 0.1 (so each component lies in $[\theta^* \cdot e^{-0.1}, \theta^* \cdot e^{+0.1}]$, i.e. a $\pm 10\%$ band) and run Adam for 500 iterations with step size 10^{-2} on 500 uniform target points. Both routes use the same sigmoid blending sharpness $\beta = 150$ across event boundaries to widen the loss basin by smoothing the effects of the events (Section 3.1).

Each method uses the same seed-0 initialization; these are single runs without standard deviations. Every five iterations, the snapshot with smallest blended training loss was retained (iteration 180 for AD and 500 for DA); neither ground-truth error nor hard validation loss selected it, and neither run met the gradient stopping tolerance. Figure 2 includes transient increases, so this offline fixed-budget rule is not evidence of monotone or across-seed convergence.

Table 1 reports the relative error per parameter, the hard-selection loss of the chosen snapshot, and the wall-clock cost. The implicit-AD route attains a lower reported hard-selection loss than the discrete-adjoint route (1.32×10^{-6} vs. 4.01×10^{-4}). It also recovers several parameters to below one percent relative error, although the remaining error on C_5 indicates that some parameter directions are weakly identifiable from the chosen output and time grid. The discrete-adjoint route recovers C_3 , C_4 , and L_1 accurately, but retains larger residual errors on C_1 , C_5 , and L_2 . Figure 1 overlays the optimized trajectories on the ground truth, and Figure 2 shows the loss histories.

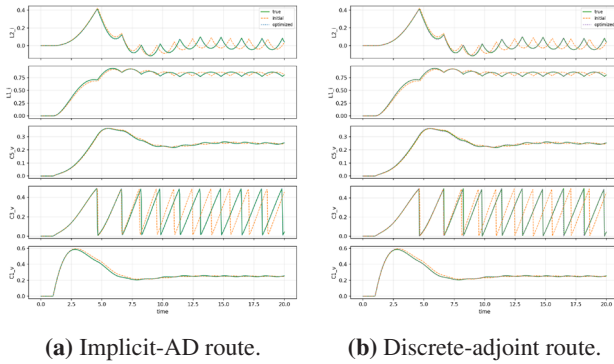


Figure 1. Cauer ladder: optimized vs. true trajectories at the best-loss snapshot. Both routes capture the qualitative event structure; the AD route additionally matches the post-reset oscillation envelope.

The parameter errors should be interpreted in the context of output sensitivity and identifiability. Different parameter combinations can produce nearly indistinguishable output trajectories, especially along weakly identifiable directions. Therefore, matching the output loss is the primary optimization objective, while exact parameter recovery depends on the informativeness of the measured outputs and the time grid.

From the optimization perspective, any parameter combination that minimizes the loss is acceptable; multiple local minima can therefore be equivalent with respect to

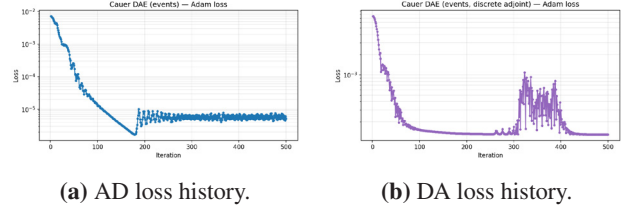


Figure 2. Cauer ladder: Adam loss histories on log scale. The AD curve descends ~ 5 decades whereas the DA curve plateaus at $\sim 10^{-4}$; the gap is consistent across the last ~ 100 iterations.

the observed outputs. The accuracy gap between the two routes on this benchmark traces back to two effects. First, the discrete-adjoint pipeline constructs the Jacobian based on a trapezoidal-step approach, so its gradient quality is bounded by the adjoint mesh density rather than by the forward solver tolerance. Second, the loss surface near the truth is anisotropic and dominated by long-time-scale plateaus between events; with a uniform target grid the trapezoidal Jacobians of the discrete adjoint accumulate a small but coherent bias along the soft directions, observable here as the residual error on C_5 and C_1 . The implicit-AD pipeline gradient inherits the diffrax tolerance ($\text{rtol}=\text{atol}=10^{-6}$) directly. The price is that AD must hold the entire diffrax tape per segment in JAX memory; on this 7-parameter system this is negligible, but it scales with the number of state dimensions and the length of the dense-output tape, a distinction that drives the next experiment. As specified in Section 3.1, the tabled losses are common-solver, hard-selection evaluations of the parameters chosen using the blended training objective.

5.2 Multi-ball planar bouncing system: scaling with N

We consider a planar system of N identical bouncing balls in a square box of half-width 10, with elastic collisions against the four walls (restitution e_b) and inelastic ground impacts (restitution e_g). The dynamics is an ODE with state-dependent events, $4N + \binom{N}{2}$ event clauses for $N \in \{3, 7, 15\}$. We optimize $4N + \binom{N}{2}$ parameters $\theta = (g, e_g, e_b)$ from a randomly biased start ($\pm 10\%$ log-uniform). All three identification routes share the same 500 points uniform target grid, the same Adam step size, and the same convergence tolerance; the only differences are how the gradient is obtained and which framework holds the autograd tape:

jax_ad our JAX/diffrax AD-through-simulation pipeline, using the same segmented event machinery as in 5.1; because this benchmark is an ODE, no algebraic solve or algebraic custom JVP is exercised.

jax_da the JAX padded-discrete-adjoint pipeline of 5.1, ported to ODE-with-events.

pytorch_ad PyTorch implementation using torchdiffeq with hand-written event-time

Table 1. Cauer ladder with a state-dependent reinit event. Per-parameter relative error (%) after 500 Adam iterations from a $\pm 10\%$ initial bias. One seed-0 run is reported without standard deviations; loss is re-evaluated by a common IDA solve with hard selection at the best blended-loss checkpoint. AD: implicit differentiation through diffrax with a JAX-AD-friendly algebraic solver. DA: discrete adjoint via Sundials/IDA forward and padded backward sweep.

Method	C_1	C_2	C_3	C_4	C_5	L_1	L_2	Loss	Time/iter
Initial	2.78	4.50	8.77	9.22	6.47	8.61	2.16	7.59e-3	—
AD	0.29	1.84	0.01	0.09	3.28	1.13	0.04	1.32e-6	0.03 s
DA	2.97	1.80	0.02	0.03	5.44	0.53	1.09	4.01e-4	0.042 s

root-finding and direct backpropagation through the entire segmented integration tape via `odeint`; `odeint_adjoint` is not used.

Each row is one fixed-initialization run without error bars, timed on one CPU thread. PyTorch uses direct `odeint` backpropagation (`adjoint=False`), so the comparison does not establish speedups over GPU execution or `odeint_adjoint`; neither was benchmarked.

Table 2 summarizes the result. Per-parameter relative errors are reported as percentages, the final validation loss is on a denser held-out target grid, and timing is total wall-clock for the budget of iterations actually used.

Three observations stand out.

(i) *Per-iteration scaling.* For these CPU implementations, JAX-AD is flattest ($74 \rightarrow 71 \rightarrow 110$ ms/iter), while PyTorch-AD grows from 554 to 6652 ms/iter. JAX-DA lies between them because its per-step Jacobians grow with segment count and event density.

(ii) *Accuracy/throughput.* At $N \in \{3, 7\}$, JAX-AD has the better measured tradeoff in these runs. At $N = 15$, PyTorch-AD reaches 3.9×10^{-8} versus 1.4×10^{-5} at $37\times$ the cost. The synthetic noise-free case supplies no application-specific error floor with which to value that additional accuracy.

(iii) *Discrete-adjoint bias.* JAX-DA is competitive on small problems, but bias from the per-segment trapezoidal mesh grows to about 4% on e_b at $N = 15$. This reflects the tested fixed mesh, not discrete adjoints in general.

6 Discussion and Comparison with Existing Approaches

Most differentiable continuous-time learning methods are formulated for ODEs. Neural ODE and `torchdiffeq` methods support backpropagation through ODE solvers or a continuous adjoint (Chen et al., 2018; Chen, 2018); event-aware extensions differentiate event times and resets for a locally fixed event sequence (Chen et al., 2021). Our setting additionally requires implicit algebraic solves during integration, after events, and during output reconstruction. Table 3 summarizes the resulting distinctions from ODE, DAE, and compiler-based approaches.

The AD-through-simulation method is closest in spirit to differentiable ODE solvers with event handling. Its advantage is implementation simplicity: once the DAE is re-

duced by solving the algebraic equations, the event-driven simulation can be written as a differentiable program. Its limitation is that the returned gradient is the derivative of the computational path selected by the forward event solver.

The explicit discrete-adjoint method is closer to constrained optimization. It keeps the residual equations visible and solves for their Lagrange multipliers. This gives direct control over the discrete residuals, event blocks, and event-time stationarity conditions. It also clarifies the theoretical role of residuals: they are not loss penalties, but equality constraints whose multipliers produce the adjoint gradient.

The discrete-adjoint construction used here is closely related to earlier work in the control and sensitivity-analysis communities. In power-system and hybrid systems analysis, trajectory sensitivities for differential-algebraic-discrete models were developed with explicit jump conditions at switching and reset events (Galán et al., 1999; Hiskens and Pai, 2000). Adjoint sensitivity methods were also developed for DAEs, including the derivation and numerical solution of adjoint DAE systems (Cao et al., 2003). More recently, discrete-adjoint methods have been implemented for hybrid dynamical systems with state- and time-dependent switching, with applications to preventive and corrective control (Zhang et al., 2017). Our formulation follows this same first-discretize-then-adjoint philosophy, but focuses on parameter learning for DAEs with event-triggered reinitialization and makes the event-split residual structure explicit. The distinction in this paper is not the existence of adjoint methods for hybrid systems, but the combination of an event-split DAE residual construction with a practical parameter-learning workflow and a comparison against AD-through-simulation implementations.

6.1 Relation to IDAS, Modelica, and FMI

IDAS integrates a user-defined backward DAE from forward checkpoints (SUNDIALS Project, 2026a), but IDASolveB does not perform backward rootfinding (SUNDIALS Project, 2026b,a). Hybrid use therefore still requires event splitting and guard, event-time, reset, and consistency jump terms. We did not implement a native IDAS-adjoint baseline and make no runtime or accuracy claim against it.

After Modelica compilation, AD-through-simulation

Table 2. Multi-ball bouncing system with $N \in \{3, 7, 15\}$ balls. Per-parameter relative error in %, final validation loss, and total wall-clock time. Budget: 300 iters for $N \in \{3, 7\}$ and 600 iters for $N = 15$. Times measured on a single CPU thread. Each entry is one run; the PyTorch method uses odeint direct backpropagation, not odeint_adjoint.

N	Method	$ \Delta g /g$	$ \Delta e_g /e_g$	$ \Delta e_b /e_b$	Loss	Iter	Total time	ms/it
3	jax_ad	0.002	0.000	0.001	1.40e-9	300	33.4sec	73.9
	jax_da	0.27	1.16	0.16	9.38e-5	300	22.1sec	61.9
	pytorch_ad	0.003	0.001	0.003	3.89e-9	300	168.6sec	554.4
7	jax_ad	0.002	0.001	0.001	1.63e-9	300	33.4sec	71.2
	jax_da	0.15	0.09	0.10	7.76e-6	300	84.0sec	254.8
	pytorch_ad	0.013	0.010	0.010	6.61e-8	300	525.5sec	1740.4
15	jax_ad	0.07	0.19	0.42	1.35e-5	600	77.2sec	110.1
	jax_da	0.21	2.67	4.04	9.28e-4	495	595.7sec	1157.5
	pytorch_ad	0.000	0.005	0.001	3.88e-8	423	2826.9sec	6652.2

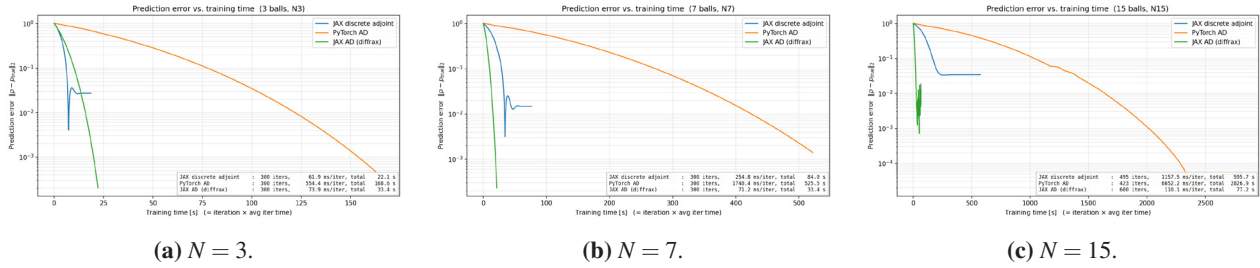


Figure 3. Bouncing balls: prediction error vs. wall-clock time. For the tested single-thread CPU implementations, the JAX-AD runs use approximately $5\times$, $16\times$, and $37\times$ less total wall-clock time than the PyTorch-AD runs for $N = 3, 7$, and 15 , respectively; the corresponding losses are reported separately in Table 2. These timing factors do not characterize GPU execution or odeint_adjoint.

could consume exported f, g, h , guards, resets, and local derivative products. Alternatively, the runtime could return an event-split forward record while generated residual and Jacobian callbacks populate the backward blocks. Both retain index reduction and tearing (Casella, 2015; Täuber et al., 2014). With FMI 3.0 Model Exchange, an importer could similarly record event indicators and Event Mode updates, then combine optional `fmi3GetDirectionalDerivative` or `fmi3GetAdjointDerivative` products with event-time and reset terms (Modelica Association Project FMI, 2024). This hybrid assembly is a proposed integration, not standardized FMI functionality.

Recent OpenModelica prototypes instead generate symbolic adjoints after BLT transformation and target compact sparse Jacobian products (Brandt et al., 2026b,a). Such products could replace local derivatives in our implicit and discrete-adjoint blocks while retaining our event-time/reset coupling. We neither implement this symbolic processing nor treat the continuous-component prototype as an equivalent hybrid-event baseline.

The algebraic solve used in our AD-through-simulation implementation should be viewed as a baseline realization of implicit differentiation rather than as a state-of-the-art Modelica compilation strategy (Fritzon, 2014). Production tools use structural analysis and tearing to reduce algebraic loops (Casella, 2015; Täuber et al., 2014); the

same reduction could shrink both our forward nonlinear and backward implicit solves. Equation-based AD can also exploit DAE structure more efficiently than a fully expanded trace (Elsheikh et al., 2015).

7 Conclusion

This paper studied differentiable parameter optimization for semi-explicit DAEs with state-dependent events, where gradients must include algebraic consistency, parameter-dependent event times, and resets.

AD-through-simulation differentiates an implicit algebraic solve and segmented event integration. The explicit discrete adjoint instead applies Lagrange multipliers to trapezoidal DAE and event equality residuals from an event-split forward trajectory.

AD is convenient in differentiable-programming toolchains; the explicit adjoint does not require an AD-through-simulation engine and gives more control over residuals and linear algebra. Both gradients are local to a fixed, transversal event path.

DAE-specific behavior is shown on one five-state, three-algebraic-variable Cauer model and all optimizations use one initialization. DAE indices, tearing, multi-seed variability, and native Modelica/IDAS-adjoint comparisons are untested.

Future work includes grazing or reordered events and

Table 3. Comparison of differentiable event-learning approaches.

Approach	Model class	Event treatment	Gradient mechanism
Neural ODE / differentiable ODE solvers	ODEs	Usually continuous trajectories; event handling available in some solvers	Backpropagation through solver operations or continuous adjoint
Neural event ODEs	ODEs with event functions and reset maps	Events terminate integration and trigger state updates	AD through event time, terminal state, and reset map; event sequence fixed locally
AD-through-simulation DAE method	Semi-explicit DAEs with state-dependent events	Events are guard-triggered resets; algebraic variables are re-solved after events	AD through reduced DAE simulation, implicit algebraic solve, event time, and reset map
Explicit discrete-adjoint DAE method	Discretized hybrid DAE residual system	Smooth-segment residuals coupled with event residuals	Backward solve for residual Lagrange multipliers; event-time stationarity handled explicitly
IDAS continuous adjoint	DAEs with user-defined backward problem	Forward rootfinding is available; hybrid resets and backward jump conditions require application-level treatment	Checkpointed forward trajectory and backward adjoint DAE
Symbolic adjoint compilation	Compiler-transformed equation blocks	Current OpenModelica prototype targets continuous strong components; hybrid event adjoints remain complementary work	Compile-time reverse equations, symbolic simplification, and sparse Jacobian products

smaller torn algebraic loops. Compiler-generated symbolic adjoints could simplify local products, but that processing is not implemented; an adaptive event-segment adjoint mesh is also planned. Further priorities are a larger DAE, multi-seed reporting, and IDAS-adjoint comparison. Modelica/FMI integration is also future work.

References

- Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998. doi:10.1137/1.9781611971392.
- Felix Brandt, Philip Hannebohm, and Bernhard Bachmann. Bidirectional jacobian evaluation using symbolic adjoint differentiation. 10th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT 2026), Bielefeld, Germany, June 2026a. URL <https://eoolt.org/2026/program.html>.
- Felix Brandt, Philip Hannebohm, and Bernhard Bachmann. Generation of symbolic adjoint derivatives: Efficient row-wise jacobian evaluation in openmodelica. Presentation at the 18th OpenModelica Annual Workshop, February 2026b. URL https://openmodelica.org/images/M_images/OpenModelicaWorkshop_2026/OpenModelica2026_Felix.pdf.
- Francesco Casella. Simulation of large-scale models in Modelica: State of the art and future perspectives. In *Proceedings of the 11th International Modelica Conference*, pages 459–468, Versailles, France, 2015. doi:10.3384/ecp15118459.
- Ricky T. Q. Chen. torchdiffeq: Differentiable ode solvers with full gpu support and o(1)-memory backpropagation. <https://github.com/rtqichen/torchdiffeq>, 2018. GitHub repository.
- Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K. Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, volume 31, pages 6571–6583, 2018. URL https://papers.nips.cc/paper_files/paper/2018/hash/69386f6bb1dfed68692a24c8686939b9-Abstract.html.

- Ricky T. Q. Chen, Brandon Amos, and Maximilian Nickel. Learning neural event functions for ordinary differential equations. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=kW_zpEmMLdP.
- Hilding Elmqvist and Martin Otter. Methods for tearing systems of equations in object-oriented modeling. In *Proceedings of the European Simulation Multiconference*, pages 326–332, Barcelona, Spain, 1994. Society for Computer Simulation.
- Ahmed Elsheikh, Francesco Casella, Dirk Zimmer, and Wladimir Schamai. An equation-based algorithmic differentiation technique for differential algebraic equations. *Journal of Computational and Applied Mathematics*, 281:135–151, 2015. doi:10.1016/j.cam.2014.12.006.
- Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. John Wiley & Sons, 2014. ISBN 978-1118859124. URL <https://onlinelibrary.wiley.com/doi/book/10.1002/9781118989166>.
- Santos Galán, William F. Feehery, and Paul I. Barton. Parametric sensitivity functions for hybrid discrete/continuous systems. *Applied Numerical Mathematics*, 31(1):17–47, 1999. doi:10.1016/S0168-9274(98)00125-1.
- Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer, Berlin, Heidelberg, 2 edition, 1996. doi:10.1007/978-3-642-05221-7.
- Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, 2005. doi:10.1145/1089014.1089020.
- Ian A. Hiskens and M. A. Pai. Trajectory sensitivity analysis of hybrid systems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(2):204–220, 2000. doi:10.1109/81.828574.
- Patrick Kidger. DiffraX: Numerical differential equation solvers in JAX. <https://github.com/patrick-kidger/diffrax>, 2021. Software library.
- Nathan J. Kong, J. Joe Payne, James Zhu, and Aaron M. Johnson. Saltation matrices: The essential tool for linearizing hybrid dynamical systems. *Proceedings of the IEEE*, 2024. doi:10.1109/JPROC.2024.3433157.
- Ion Matei, Maksym Zhenirovsky, and Anthony Wong. Differentiable parameter optimization for daes with state-dependent events, 2026. URL <https://arxiv.org/abs/2605.05395>.
- Modelica Association. *Modelica Language Specification, Version 3.6, Chapter 8: Equations*, 2023. URL <https://specification.modelica.org/maint/3.6/equations.html>. Sections 8.3.5–8.5: when-equations, reinit, events, smooth, and noEvent.
- Modelica Association. *Modelica Language Specification: Appendix B, Modelica DAE Representation*. Modelica Association, 2025. URL <https://specification.modelica.org/master/modelica-dae-representation.html>. Accessed 2026-05-04.
- Modelica Association Project FMI. *Functional Mock-up Interface Specification, Version 3.0.2*, November 2024. URL <https://fmi-standard.org/docs/3.0.2/>.
- Linda R. Petzold. Description of DASSL: A differential/algebraic system solver. Technical report, Sandia National Laboratories, September 1982. URL <https://www.osti.gov/biblio/5882821>.
- Alessandro Saccon, Nathan van de Wouw, and Henk Nijmeijer. Sensitivity analysis of hybrid systems with state jumps with application to trajectory tracking. In *Proceedings of the 53rd IEEE Conference on Decision and Control*, pages 3065–3070, 2014. doi:10.1109/CDC.2014.7039877.
- SUNDIALS Project. *Using IDAS for Adjoint Sensitivity Analysis*. Lawrence Livermore National Laboratory, 2026a. URL <https://sundials.readthedocs.io/en/latest/idas/Usage/ADJ.html>. SUNDIALS IDAS user documentation, accessed September 2, 2026.
- SUNDIALS Project. *Using IDAS for Initial-Value Problem Solution: Rootfinding*. Lawrence Livermore National Laboratory, 2026b. URL <https://sundials.readthedocs.io/en/latest/idas/Usage/SIM.html>. SUNDIALS IDAS user documentation, accessed September 2, 2026.
- Patrick Täuber, Lennart Ochel, Willi Braun, and Bernhard Bachmann. Practical realization and adaptation of Cellier’s tearing method. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 11–19. ACM Press, 2014. doi:10.1145/2666202.2666204.
- Hong Zhang, Shrirang Abhyankar, Emil M. Constantinescu, and Mihai Anitescu. Discrete adjoint sensitivity analysis of hybrid dynamical systems with switching. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(5):1247–1259, 2017. doi:10.1109/TCSI.2017.2651683.
- Dirk Zimmer. Module-preserving compilation of Modelica models. In *Proceedings of the 7th International Modelica Conference*, pages 556–565, Como, Italy, 2009. URL https://ep.liu.se/en/conference-article.aspx?Article_No=104&issue=43&series=ecp.

ModelScript: Rethinking the Digital Thread with a Web-Native, Polyglot Compiler for Modelica, SysMLv2 & Beyond

Mohamad Omar Nachawati¹

¹Independent Researcher, Founder of ModelScript.org, omar@modelscrip.org

Abstract

This paper introduces ModelScript, a web-native, polyglot compiler that reframes the Digital Thread as an incremental compilation problem. While integrating systems architecture and physical simulation traditionally relies on fragile data-linking or opaque co-simulation, ModelScript unifies SysML v2 and Modelica directly into a shared computable graph. By procedurally extracting multi-fidelity parameters from upstream 3D CAD geometries (e.g., STEP AP242), the compiler bridges the gap between mechanical design and 1D behavioral simulation. Powered by a demand-driven query engine, cross-domain flattening, and a sparse numerical simulation runtime, it generates executable equation systems entirely within the browser. To ensure broad interoperability, the platform natively exports standard FMI-compliant Functional Mock-up Units (FMUs). Experimental results demonstrate that this architecture maintains sub-300 ms responsive feedback for high-dimensional models (up to 5,000 equations), transitioning the engineering lifecycle toward interactive verification at the edge.

Keywords: *Modelica, SysML v2, FMI, AP242, MBSE, Polyglot Compiler, Digital Thread, Hardware-as-Code*

1 Introduction

Modern cyber-physical systems have reached high levels of complexity, demanding tight integration between requirements, systems architecture, 3D mechanical geometry, and multi-domain physics simulation. To manage this complexity, the engineering industry has increasingly transitioned away from fragile, document-centric workflows toward Model-Based Systems Engineering (MBSE). At the core of this transition is the Digital Thread: an unbroken chain of traceability intended to seamlessly connect different engineering models, such as SysML v2 requirements, STEP AP242 geometries, and Modelica physical simulations. Ideally, this thread promises to break down isolated engineering silos, establishing a single source of truth that enables early and continuous verification across domains.

In reality, implementations of the Digital Thread often fall short of this ideal. While interoperability standards like the Functional Mock-up Interface (FMI) are highly effective for system-level co-simulation, relying on them during highly iterative design phases forces en-

gineers to exchange black-box simulation models, which slows down the verification process. Because these compiled models hide the underlying physical equations, they easily become outdated and inconsistent as the original designs change. This exposes a fundamental bottleneck: separate engineering tools do not natively communicate. As a result, engineers cannot instantly check for errors across domains. Instead, they must rely on tedious export-import cycles to find basic mistakes rather than receiving immediate compiler feedback. Another major challenge in realizing this vision is balancing rigor and agility (Tolentino and Wood 2018). While rigor is essential for ensuring accuracy, it can reduce agility by extending the time required to implement design changes. This tradeoff is especially challenging in early-stage projects, where system definitions are highly volatile.

To overcome these limitations, this paper introduces ModelScript: a web-native compiler designed to instantly check for errors across domains. Instead of exchanging files between tools, ModelScript directly parses SysML v2 and Modelica, unifying systems architecture, physical behavior, and procedural CAD geometries into a shared computable graph. By leveraging modern web architecture, ModelScript creates an active, computable model directly in the browser without needing a remote server. This work illustrates how applying compiler principles to the Digital Thread can facilitate rapid verification across system architectures, mechanical geometries, and physical simulations. By combining different models into a unified graph, the compiler maps architectures and physics into a solvable system of equations, while simultaneously evaluating design requirements as mathematical assertions. By moving from simple metadata links to unified compiler models, this approach transitions the engineering lifecycle from delayed, offline synchronization to web-native interactive verification.

To realize this vision, this paper makes three core contributions. First, we establish the Code-centric Digital Thread, a novel integration paradigm that breaks isolated engineering silos by dynamically unifying different modeling languages. Second, we develop a responsive, polyglot compiler architecture featuring an incremental query engine, cross-domain resolution and flattening, and a multi-target code generation pipeline backed by a sparse numerical simulation runtime. Finally, we deploy this architecture within VS Code for the Web to enable continu-

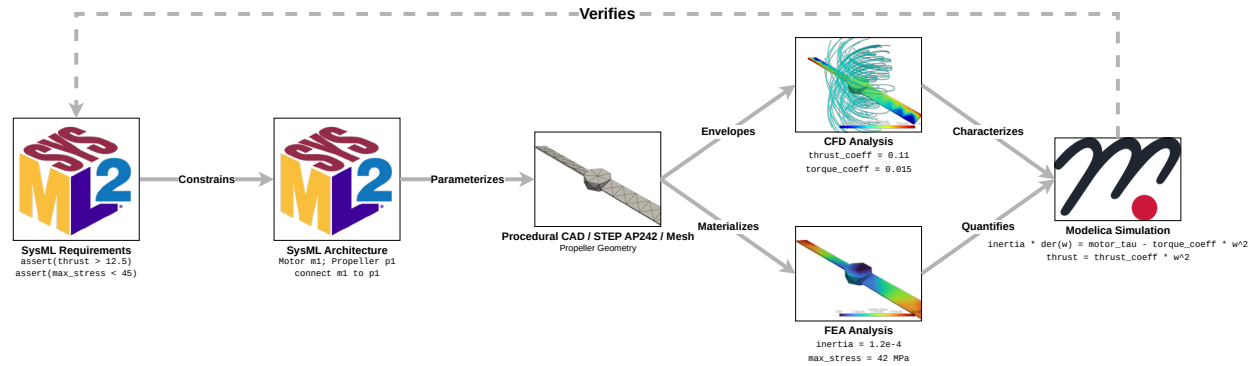


Figure 1. The Code-centric Digital Thread topology. The compiler maps geometric parameters (CAD, CFD, FEA) and architectural topology (SysML) into a unified Modelica physical simulation, against which SysML verification bounds are continuously evaluated.

ous engineering verification, and empirically prove that it maintains sub-300ms incremental compilation latency for models scaling up to 5,000 equations.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces the core philosophy of the platform, formalizing how the Code-centric Digital Thread breaks traditional engineering silos. Section 4 details the underlying polyglot compiler architecture that makes this continuous verification possible. Section 5 presents concrete industrial applications to demonstrate the practical value of this approach. Finally, Section 6 evaluates the compiler’s performance through experimental benchmarks, and Section 7 concludes the paper.

2 Related Work

To properly contextualize the contributions of ModelScript, it is essential to assess the current landscape of Digital Thread architectures and cross-domain interoperability frameworks. The challenge of cross-domain engineering integration has been approached from several distinct angles. However, existing solutions fundamentally differ from the polyglot compiler approach proposed in this paper, which aims to unify disjoint domains into the continuous topology illustrated in Figure 1.

The common approach for integrating different physics domains relies heavily on co-simulation frameworks and the Functional Mock-up Interface (FMI) standard. Tools such as Ansys ModelCenter or modeFRONTIER operate by compiling models into opaque execution units (FMUs) and orchestrating their inputs and outputs. Similarly, Model-to-Model (M2M) translation workflows have driven the development of integrated verification frameworks for SysML and Modelica (e.g., Sun et al. (2026)). While these frameworks successfully map architectural constraints to physics parameters, they remain constrained by their reliance on static code generation and fragile file-based exchanges. This “export-import-execute” cycle pre-

vents tools from natively understanding each other, introduces high delays, and makes it impossible to instantly check for errors during early-stage iterative design.

A separate approach focuses on traceability through Product Lifecycle Management (PLM) integration layers (Stark 2011), which provide the Authoritative Source of Truth (ASoT) mandated by modern engineering standards (INCOSE 2021). To overcome the limitations of these monolithic, tool-centric databases, the industry has seen a rapid evolution toward ontology-based Knowledge Graphs, exemplified by the NIO Semantic Digital Thread (Hooshmand 2025), Intercax Syndeia, and JPL’s OpenCAESAR (Elaasar et al. 2023). Recent efforts have even adopted graph-based metadata models to modernize co-simulation DevOps (Reiterer et al. 2023). However, these ontology databases act as *passive* linkers (e.g., relying on external variable-mapping between SysML and CAD (Bajaj et al. 2016)). They track dependencies and manage versions—identifying *that* a link exists—but lack an understanding of the underlying code. Consequently, they cannot automatically check for structural errors or solve equations across tool boundaries.

To address early-stage synchronization, collaborative concurrent engineering platforms like Valispace utilize centralized cloud databases as a single source of truth for engineering variables (e.g., mass, power budgets). While highly effective for parameter synchronization in the enterprise “Outer Loop” (Brost et al. 1993), these systems function similarly to advanced collaborative spreadsheets: they synchronize scalar numbers and simple formulas but cannot parse, unify, or verify complex hierarchical architectures or continuous geometries required by the “Inner Loop” of agile engineering.

Finally, there is a nascent industry shift toward “Hardware-as-Code” paradigms and incremental compiler frameworks. This transition naturally extends the long-standing success of Modelica (Fritzson and Engelson 1998), which pioneered dual-language architectures for

physical modeling. Furthermore, recent work has demonstrated the viability of decentralized, NPM-style package management for these hardware architectures (Nachawati et al. 2022). While modern tools have demonstrated the viability of code-centric engineering and edge execution, existing solutions remain heavily siloed within single domains. Although incremental compilation architectures exist, none have yet applied these techniques to create a unified graph spanning SysML, procedural CAD geometries, and Modelica. ModelScript bridges this specific gap.

3 The Code-centric Digital Thread

ModelScript moves the Digital Thread beyond simple data links. A primary contribution of the platform is its ability to unify different models directly at the compiler level. By reading these models and checking them against each other directly in the browser, ModelScript creates a foundation for continuous verification and traceability across engineering domains.

The Computable Semantic Graph. To achieve this, the Digital Thread must act like a compiler instead of just a database. While modern databases successfully trace relationships across models, they treat physics simulations as black boxes. Because they lack a native compilation engine, they cannot instantly verify the underlying math. ModelScript overcomes this by directly parsing the source code of SysML v2 and Modelica—alongside native procedural CAD constructs—into domain-specific Abstract Syntax Trees (ASTs). The compiler preserves each language’s native structure rather than forcing them into a single format.

Polyglot Semantic Projection. These independent models are unified on demand. The compiler maps the structure and parameters of a SysML v2 architecture directly into the solvable equations of its Modelica implementation. By analyzing both the architecture and the math, ModelScript checks for errors across domains. This reports inconsistencies as compiler errors, catching integration mistakes before simulation. Because this process runs in memory, it eliminates high-latency export-import cycles. This instant feedback is ideal for the rapid iteration of early-stage design. The system can also generate standard Functional Mock-up Units (FMUs) for external tools.

Continuous Verification Architecture. With this engine in place, traceability becomes a compile-time verification step, enabling concurrent hardware-software co-design. Software logic can be tested against fixed physical constraints before manufacturing.

ModelScript divides this process into two phases: static analysis and dynamic simulation. During the static phase, the compiler performs cross-domain type-checking. For instance, if a Modelica model assigns a String value of “fast” to a SysML *Real* speed attribute, the compiler catches the error immediately. ModelScript highlights the error directly in the source code.

Static verification also goes beyond basic type-checking. Because the compiler analyzes the entire system mathematically, it performs deep structural verification. Consider a scenario where a SysML test harness hardcodes a propeller’s speed, but the Modelica model defines this speed as a dynamic state dependent on inertia and drag (Figure 1). A standard type-checker would accept this, but ModelScript maps the equations and instantly detects that the variable is constrained by two conflicting equations. It emits a structural over-constraint error before simulation.

For constraints that must be evaluated over time, the compiler seamlessly transitions into dynamic simulation. By running simulations directly in the same environment, ModelScript unifies static checking and dynamic execution, providing an interactive development experience.

Web-Native Execution. Relying exclusively on a centralized cloud architecture for compilation introduces deployment constraints, particularly for engineers in disconnected environments like test tracks or remote field sites.

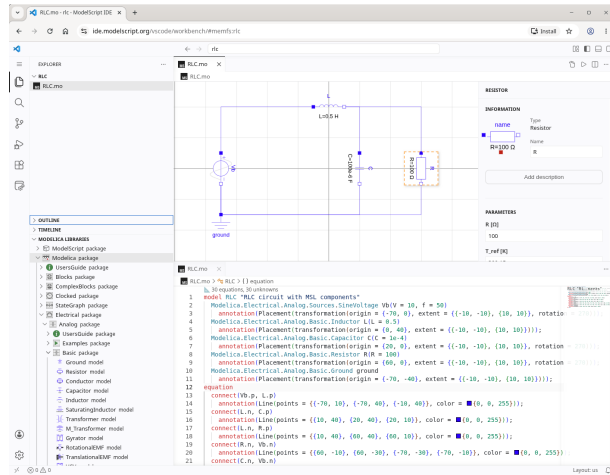
ModelScript resolves this bottleneck with its web-native compiler architecture. It runs the entire compilation pipeline—including parsing, mathematical flattening, and code generation—directly in the browser. For simulations, the platform compiles both the equations and standard numerical solvers (like SUNDIALS and SuiteSparse KLU) into WebAssembly (WASM). This allows models to run at near-native speed on the user’s device.

By running locally, ModelScript enables robust, offline multi-physics verification. Engineers can compile models into animated digital twins completely offline. This allows teams to visually identify physical issues before manufacturing, providing flexibility for distributed workflows.

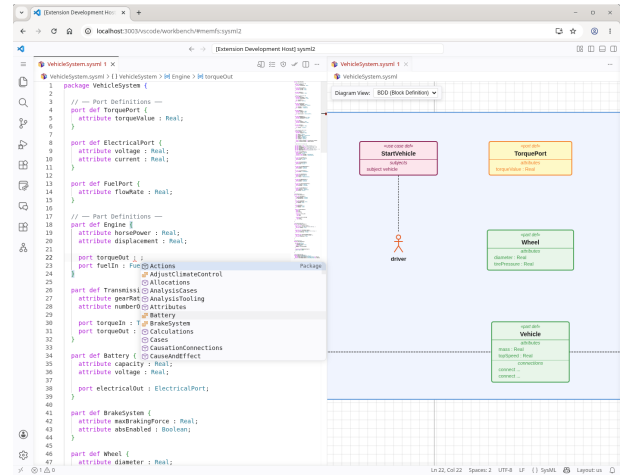
4 Implementation

To realize the interactive edge-native verification described in Section 3, ModelScript transitions from a traditional, pass-based compilation model to a demand-driven, reactive query engine inspired by the Rust-analyzer’s Salsa framework. The polyglot pipeline, spanning incremental Tree-sitter parsing, semantic indexing, and flattening, is modeled as a dependency graph of memoized, pure functions. When an engineer modifies a source file, the compiler invalidates only the specific downstream queries affected by the syntax change, rather than recompiling the entire system. This incremental architecture enables true continuous engineering agility; a parameter change propagates instantly through algorithmic generation rules, auto-adjusting tolerances dynamically and providing real-time diagnostic feedback on large industrial libraries.

The Incremental Query Engine. Under the hood, this architecture operates by strictly separating base “input queries” (raw source text) from “derived queries” (resolved symbols and flattened equations). As derived queries execute, the framework automatically records their precise data dependencies, enabling surgical cache



(a) Synchronized Modelica code and diagram editing.



(b) SysML v2 code editing with synchronized diagram.

Figure 2. Interactive digital thread workflows facilitated by the ModelScript IDE directly in the browser.

invalidation. To ensure the compiler can flatten high-dimensional models without exhausting strict browser memory limits, ModelScript leverages an arena-based memory allocation strategy for its Differential Algebraic Equation (DAE) representation. Rather than instantiating millions of independent, garbage-collected objects during compilation, flattened equations and variables are stored sequentially within flat memory arenas and referenced via integer IDs. This contiguous layout drastically reduces the compiler's memory footprint and eliminates unpredictable garbage collection pauses during the flattening phase. However, the incremental AST parsing and semantic symbol indexing still rely on standard garbage-collected V8 objects, which remains the primary source of memory overhead when scaling to large systems. Active work is underway to introduce this contiguous memory strategy further up the pipeline to accelerate these earlier phases.

Polyglot Language Parsing. The foundation of the polyglot compiler is its ability to process multiple modeling languages. To support evolving standards like SysML v2, ModelScript uses a schema-driven approach to language processing powered by a central `language.ts` definition. Rather than relying on rigid, handwritten parsers and disconnected configurations, `language.ts` consolidates the language semantics into a single authoritative model.

From this single source of truth, the framework automatically produces multiple compiler artifacts. This includes not only the `grammar.js` required by the Tree-sitter parsing engine, but also the semantic lint queries, syntax highlighting rules, and code folding definitions. A critical advantage of this generated Tree-sitter integration is its fault-tolerance; the parser recovers from syntax errors, allowing the compiler to provide helpful IDE diagnostics even while the user is actively typing. The incremental query engine ensures that once an error is

fixed, any affected downstream calculations are automatically updated. This responsiveness supports real-time editing workflows, providing synchronized code and diagram editing for both Modelica (Figure 2a) and SysML v2 (Figure 2b) architectures directly within the IDE.

Cross-Domain Resolution and Flattening. Instead of performing direct source-to-source text translation, the parsers unify the models into a shared graph. To establish consistency, ModelScript implements a mapping framework inspired by the OMG SysML-Modelica Transformation (SyM) standard (Object Management Group (OMG) 2012). The compiler maps SysML v2 constructs directly to Modelica equations. By integrating at this structural level, the compiler enforces type consistency across domains, catching mismatches instantly.

The compiler then reduces this unified graph into a flat, solvable system of Differential Algebraic Equations (DAEs). To extract an optimized minimal set, the flattening process employs Equality Graphs (e-graphs) for advanced algebraic simplification, utilizing specific heuristics to isolate variables and analytically reduce complex terms. To assign equations to the unknown variables, the compiler constructs a bipartite graph and transforms the system into Block Lower Triangular (BLT) form. This structural matching mechanism catches conflicting boundary conditions and over-constraints across domains.

Because standard numerical integration routines (such as CVODE) require explicit Ordinary Differential Equations (ODEs) rather than high-index DAEs, the compiler must perform index reduction. ModelScript applies the Pantelides algorithm to systematically differentiate specific equations until all hidden algebraic constraints are exposed, successfully lowering the system's differentiation index. Following this, the Dummy Derivatives algorithm is employed to statically select a valid set of state variables, allowing the system to be safely formulated as an ODE. While many of these structural analysis algo-

rithms are fundamental to the standard Modelica backend, adapting them within a web-native query engine ensures that the entire cross-domain verification pipeline remains both mathematically rigorous and responsive.

Multi-Target Code Generation. A modern engineering platform must support diverse execution environments, balancing low-latency browser execution with standard co-simulation interoperability. To achieve this, ModelScript implements a backend-agnostic code generation pipeline that lowers the resolved explicit ODE representation into multiple distinct execution targets.

For rapid UI prototyping and debugging, the compiler emits pure JavaScript bound to a lightweight embedded solver (e.g., Dopri5). For high-performance browser execution, the pipeline generates WebAssembly (WASM), linking the compiled equations directly against a cross-compiled SUNDIALS CVODE solver suite. This edge-native approach delivers continuous simulation entirely within the client without relying on cloud infrastructure. Finally, to ensure broad interoperability with standard co-simulation environments, the platform generates Functional Mock-up Units (FMUs) packaged in both native C and WASM formats. As demonstrated in Section 6, the resulting native C targets achieve execution speed and memory efficiency at near-parity with the highly optimized OpenModelica baseline. While browser and Node-based WASM targets naturally carry the base memory overhead of their host JavaScript engines, the underlying simulation runtimes prove that web-native compilation can yield production-grade numerical simulation performance.

Sparse Numerical Simulation Runtime. To provide Jacobians to the numerical solvers, ModelScript utilizes automatic differentiation directly at the compiler level. By computing these exact gradients and structuring them in the Compressed Sparse Column (CSC) format, the compiler enables direct integration with sparse linear solvers such as KLU. This memory-efficient approach prevents Out-Of-Memory (OOM) failures on bounded WebAssembly engines, ensuring rapid compilation and exact gradient accuracy for large optimizations.

For simulation, the platform uses the SUNDIALS CVODE solver suite cross-compiled to WebAssembly. To support high-dimensional models—such as the 10,000-component systems evaluated in Section 6—the runtime integrates the SuiteSparse KLU solver. KLU is highly optimized for the specific sparsity patterns found in physical equations. By providing the KLU solver with sparse Jacobians, the WebAssembly runtime bypasses the bottlenecks that typically cripple browser-based numerical tools, achieving the scalability required for industrial models.

5 Applications

ModelScript allows engineers to connect high-level system requirements directly to executable physics models. By unifying these domains at the compiler level, it catches

cross-domain design errors interactively in the code editor, avoiding the high-latency synchronization steps required by traditional, siloed engineering tools. This section demonstrates the practical value of this architecture by detailing four core workflows: Simulation-Based Verification, Multi-Fidelity Parameter Binding, Data-Driven Parameter Calibration, and Hardware Continuous Integration.

Simulation-Based Verification. ModelScript converts static SysML v2 operational requirements (e.g., maximum temperature limits) into executable trajectory checks. The pipeline extracts SysML `verification def` blocks and compiles their limits into post-simulation assertions. After the WebAssembly solver computes the dynamic Modelica physics, the compiler evaluates these assertions against the generated time-series data. If any requirement is found to be violated—such as the battery pack thermal runaway fault evaluated in Section 6—the pipeline flags the failure and traces the violation directly back to the offending SysML source line.

Multi-Fidelity Parameter Binding. ModelScript can procedurally extract multi-fidelity parameters from upstream 3D CAD tools directly into the 1D behavioral simulation. If an engineer updates a drone propeller's pitch, the aerodynamic `thrust_coeff` and rotational `inertia` can be re-computed and mapped into Modelica `parameter` declarations. By separating quick static checks from full dynamic simulations, the compiler instantly emits static cross-domain lints if a parameter violates a basic SysML limit, while using WebAssembly to verify how the entire system behaves over time.

Data-Driven Parameter Calibration. To synchronize digital twins with real-world telemetry, ModelScript automates parameter calibration. A visual IDE panel allows engineers to bound unknown parameters. The platform then reads CSV sensor data and executes a Levenberg-Marquardt optimization routine, utilizing compiler-generated analytic gradients for rapid convergence. The tuned parameters can then be verified against SysML architectural limits to ensure the updated model remains compliant before saving the modifications back to the codebase.

Hardware Continuous Integration. By treating physical architectures as code, ModelScript integrates hardware engineering into modern Git-based CI/CD pipelines. When changes are proposed via Pull Requests, the headless compiler flattens and solves the SysML and Modelica integration. If a change introduces a structural over-constraint or violates a physical requirement—such as the conflicting boundary conditions evaluated in Section 6—the build fails. Upon merge, the release pipeline can export the verified Bill of Materials, 3D geometries, and standard FMUs directly to enterprise PLM systems. This workflow isolates the fast engineering “inner loop” from the formal “outer loop”, preserving agility while maintaining the authoritative source of truth.

6 Experiments

This section evaluates the performance of the ModelScript polyglot compiler through a series of experiments. The evaluation focuses on four core criteria: semantic flattening correctness, incremental compilation latency, execution speed across varying runtimes, and cross-domain verification and latency. All experiments were executed on an Intel Core Ultra 9 285H processor (16 cores) with 32 GB of RAM, running Debian GNU/Linux 13, Node.js v24.14.0, and VS Code version 1.100.0. To automate the rigorous benchmarking pipeline, experiments were executed in Node.js. Because Node.js utilizes the same V8 JavaScript engine as modern Chromium browsers, the CPU metrics provide a reliable proxy for browser-native performance. For baseline correctness comparisons, the OpenModelica (OMC) 1.26.7 toolchain (Fritzson, Aronsson, et al. 2005) was utilized as the canonical reference.

6.1 Flattening Correctness against OMC

Table 1. Flattening correctness results across the OpenModelica Modelica flattening test suite categories.

Category	Total Models	Passed	Pass Rate
<i>Complex Instantiations</i>			
scodeinst	1324	224	16.9%
<i>Object-Oriented & Architecture</i>			
modification	88	34	38.6%
connectors	82	11	13.4%
redeclare	71	10	14.1%
extends	56	23	41.1%
scoping	26	5	19.2%
packages	20	6	30.0%
<i>Data Structures & Types</i>			
arrays	186	20	10.8%
declarations	100	31	31.0%
records	42	0	0.0%
enums	28	1	3.6%
types	27	11	40.7%
<i>Mathematics & Logic</i>			
algorithms-functions	132	46	34.8%
built-in-functions	79	28	35.4%
equations	60	10	16.7%
operators	56	9	16.1%
<i>Advanced & External</i>			
others	108	20	18.5%
mosfiles	84	5	6.0%
msl	60	0	0.0%
ffi	29	0	0.0%
synchronous	17	1	5.9%
external-functions	15	2	13.3%
streams	14	0	0.0%
expandable	9	0	0.0%
asserts	6	3	50.0%
statemachines	8	3	37.5%
blocks	4	2	50.0%
external-objects	2	0	0.0%
modelica-output	1	0	0.0%
Total	2749	505	18.4%

Compiler correctness was evaluated by comparing the ModelScript flattener against the open-source OpenModelica (OMC) toolchain. We generated baseline results using OMC and compared them against ModelScript’s generated equation sets using a strict text-based comparison to confirm they match.

As detailed in Table 1, ModelScript successfully flattened 505 out of 2749 models, achieving an overall pass rate of 18.4%. It scored 38.6% on Modifications, 10.8% on Arrays, and 16.9% on SCodeInst. The compiler is quite capable and supports many advanced Modelica features, such as state machines, event-driven `when`-equations, and complex class modifications. However, it currently fails a majority of the OpenModelica reference tests. Some tests fail simply because our compiler outputs equations in a different order or simplifies them differently, causing a strict text-based mismatch even when the underlying math is identical. Fixing these remaining issues is a major focus of our ongoing work.

6.2 Simulation Accuracy against OMC

We validated simulation accuracy by exporting ModelScript models across multiple backends (JavaScript, WASM, and C-FMU). Evaluations confirm high numerical accuracy, with the maximum difference between ModelScript and OpenModelica trajectories remaining below 3×10^{-2} across all test models. We compared the results between OMC and the ModelScript engines across 8 test models (including `SimpleCircuit`, `BouncingBall`, and `CoupledClutches`).

As shown in Figure 3, the trajectories between ModelScript and OMC remain tightly overlapping across the entire simulation, demonstrating high fidelity regardless of the chosen backend.

6.3 Incremental Compilation Latency

To evaluate the responsiveness of the query engine, we measured IDE latencies using a 1D Heat Conduction model scaled up to $N = 5,000$ equations. The structure of the test model is shown in Listing 1.

The experiment isolated eight editing conditions to contrast local edits against global changes: (1) *Cold Start*: full parsing and evaluation; (2) *Root Parameter Rename*: renaming `L` to `L_new`, triggering a global update; (3) *Global State Modifier*: changing `start=zeros(N)`; (4) *Isolated Equation Edit*: modifying a constant to test local updates; (5) *Isolated Structural Edit*: adding an algebraic loop; (6) *Local Variable Rename*: renaming `alpha`, which triggers scattered updates across every equation; (7) *Introduce Type Error*: replacing a number with a string; and (8) *Introduce Syntax Error*: removing a semi-colon.

As shown in Figure 4, the compiler provides fast feedback (under 300 ms) even for large models ($N \leq 5,000$), and near-instant feedback (under 10 ms) for small systems. By combining incremental Tree-Sitter parsing with a caching verification pipeline, the compiler decouples

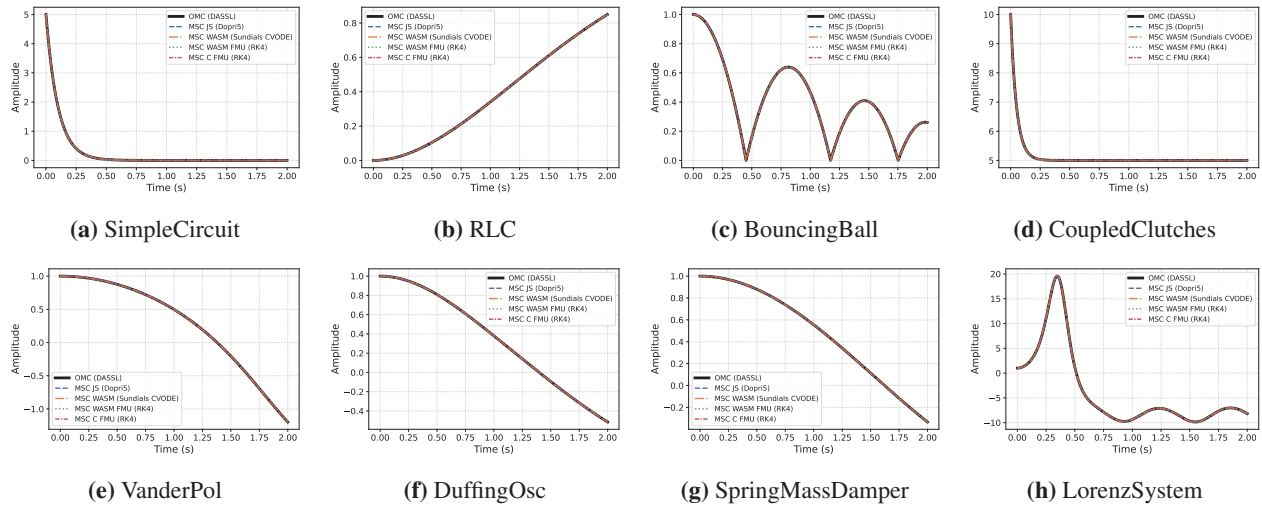


Figure 3. Simulation trajectory comparison between the native OpenModelica reference (OMC DASSL) and the generated Mod- elScript simulation engines (MSC JS Dopri5, MSC WASM Sundials CVODE, MSC C FMU RK4, and MSC WASM FMU RK4) across representative test models.

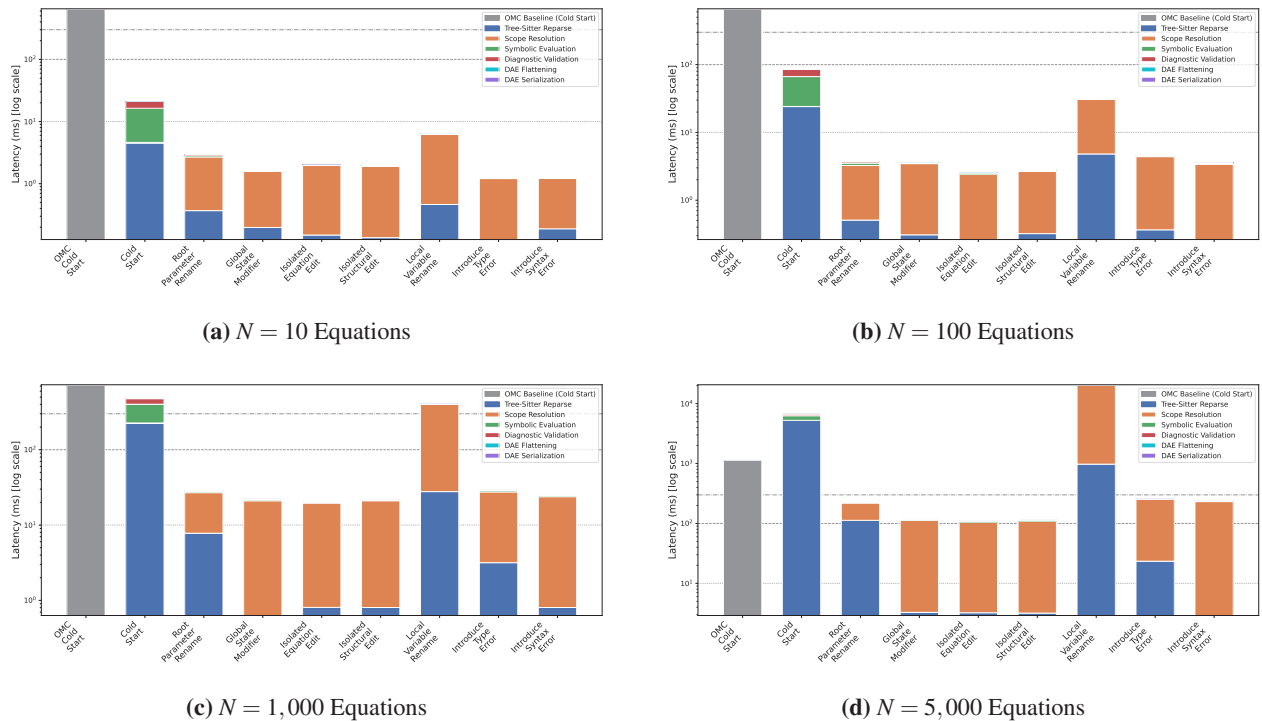


Figure 4. Incremental Compilation Latency Breakdown across various model scales (N). The visualizations capture the total millisecond latency required to execute the compiler pipeline phases (Reparsing, Scope Resolution, Symbolic Evaluation, and Diagnostic Verification) in response to different editing conditions, compared against OpenModelica (OMC) cold-start initialization times. The dash-dotted, dashed, and dotted lines indicate 300 ms, 100 ms, and 10 ms responsiveness budgets respectively.

Listing 1. Skeleton structure of the synthetic 1D Heat Conduction test model.

```

model HeatConduction1D_N
  parameter Integer N = 5000;
  parameter Real L = 1.0;
  parameter Real dx = L / N;
  parameter Real alpha = 1e-4;
  Real T[N] (start=zeros(N));
equation
  der(T[1]) = alpha * (100.0 - 2.0*T[1] + T
    [2]) / (dx^2);
  // ... N-2 inner elements ...
  der(T[N]) = 2.0 * alpha * (T[N-1] - T[N]) /
    (dx^2);
end HeatConduction1D_N;

```

parsing and evaluation time from the overall file size. This results in near-instant feedback for local edits. Crucially, the architecture caches individual equations to localize updates, minimizing the impact of the model's total size. While global operations like *Local Variable Rename* take longer because they modify every equation, isolated edits bypass these global updates and verify instantly.

Listing 2. Skeleton structure of the synthetic N -equation Cascade evaluation model.

```

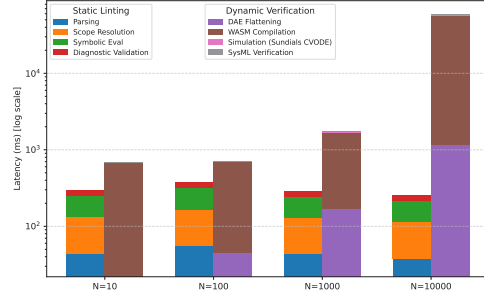
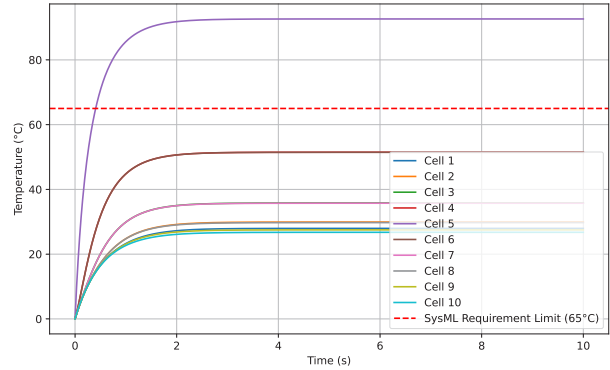
model Cascade_N
  Real x1(start=1.0);
  Real x2(start=1.0);
  // ... N-2 inner elements ...
  Real xN(start=1.0);
equation
  der(x1) = -x1;
  der(x2) = x1 - x2;
  // ... N-2 inner elements ...
  der(xN) = x_{N-1} - xN;
end Cascade_N;

```

6.4 Compilation Scalability

Because ModelScript parses and flattens models entirely within the browser and Node.js, its resource constraints differ significantly from traditional desktop toolchains. To evaluate the scalability of this web-native pipeline, we measured the compilation CPU time and peak memory consumption for a linear cascade system scaled from $N = 10$ to $N = 5,000$ equations. The structure of this evaluation model is shown in Listing 2.

As shown in Figure 7, the compilation metrics highlight a distinct memory profile when executing in JavaScript environments. While the internal arena allocator efficiently manages the flattened DAE representation with minimal overhead, scaling up to high-dimensional systems ($N \geq 5,000$) exposes the inherent limitations of the V8 JavaScript engine's garbage collection. Specifically, the millions of transient objects generated during AST parsing and semantic indexing lead to higher peak memory usage compared to the highly optimized, native Open-

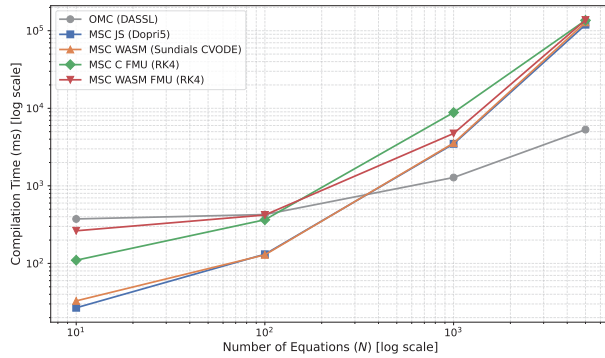
**Figure 5.** Computational latency of static polyglot projection and dynamic SysML requirements verification across a parameterized N -component battery pack architecture. Because the physical system scale is defined via an array parameter rather than explicit source-code generation, parsing latency is naturally constant, highlighting the query engine's ability to resolve high-dimensional architectural arrays in $O(1)$ time during static diagnostics.**Figure 6.** Dynamic simulation of the 1D thermal model verifying the SysML MaxTemperatureCheck requirement. The native WASM CVODE/KLU solver continuously evaluated the thermal propagation, accurately detecting the localized thermal runaway fault and immediately triggering a SysML requirement violation as the temperature crossed the 65°C safety threshold.

Modelica baseline. We are actively implementing further contiguous memory optimizations to mitigate this V8 overhead. Despite this compilation-phase memory footprint, once the equations are successfully lowered into WebAssembly or native C execution targets, the resulting simulation runtime drops to become exceptionally lightweight, maintaining robust execution stability.

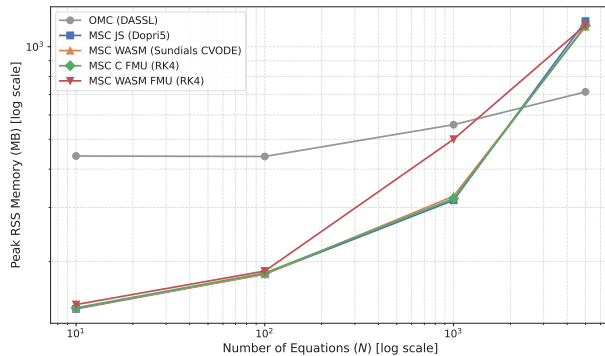
6.5 Cross-Domain Verification and Latency

To evaluate the compiler's ability to maintain a continuously verified Digital Thread at scale, we modeled a parameterized N -component battery pack spanning SysML v2 architectural constraints (Listing 3) and Modelica physical dynamics (Listing 4). We evaluated both static diagnostic accuracy and dynamic requirements verification.

First, we injected structural errors to evaluate the compiler's cross-domain checking. High-level constraints defined in SysML can conflict with the physical equations



(a) Compilation CPU Time



(b) Compilation Peak Memory

Figure 7. Scalability metrics comparing ModelScript compilation performance against native OpenModelica baselines for N identical cascade equations. (Logarithmic scaling on both axes).

defined in Modelica. For example, the Modelica thermal system is perfectly balanced, meaning the number of unknown variables matches the number of equations. If we add a SysML constraint that hardcodes a cell's temperature (e.g., `cells[1].temp == 50.0`), the combined system has too many constraints. ModelScript detected 100% of these discrepancies at compile-time. Because the system size is defined by an array parameter, the parsing phase is identical across all sizes. The query engine mapped the SysML constraint down to the Modelica array without fully unrolling the equations, localizing the fault in a 10,000-component system in just 255 milliseconds.

Second, we dynamically verified a SysML maximum temperature requirement using ModelScript's WebAssembly pipeline. While static checks are nearly instant, dynamic simulation involves compiling the equations and solving them over time. The compiler successfully evaluated the thermal trajectory for a 1,000-component system in a few seconds, and verified a 10,000-component system in 57 seconds. Note that a 10,000-component battery pack ($N = 10,000$ cells) generates approximately 20,000 underlying scalar equations (for temperature and power), which pushes the scale beyond the 5,000 equation benchmark in Section 6.3. By catching structural errors at

compile-time in just 255 milliseconds, ModelScript provides immediate feedback, while still providing a robust simulation pipeline in the browser (Figure 5). To test error reporting, we injected a heat fault. As the thermal run-away caused the temperature to exceed the 65°C limit set by SysML, the post-simulation verification engine flagged the failure (Figure 6). For the $N = 10$ model, this produced the following verification failure:

```
[VERIFICATION FAILED] Requirement 'max_temp'
violated:
pack_instance.cells[5].temp reached 92.8 (
limit: 65.0) at t=1.40s
```

7 Conclusion and Future Work

The persistence of isolated engineering tools remains a critical bottleneck in cyber-physical system design. This paper introduced ModelScript to overcome these limitations by reframing the Digital Thread as a compiler problem. By parsing and unifying engineering models into a shared computable graph, ModelScript establishes a Code-centric Digital Thread. Experimental results validate the efficacy of this approach: the polyglot compiler maintains sub-300 ms responsive feedback for high-dimensional models ($N \leq 5,000$), while securely lowering the complex architectural graphs into exceptionally lightweight WebAssembly and native C simulation runtimes. By executing directly within the web browser, this architecture transitions the engineering lifecycle from delayed synchronization to interactive verification.

While ModelScript establishes a robust foundation for continuous verification, several critical avenues remain for future research. The immediate roadmap prioritizes solidifying the core compiler to fully support the Modelica standard, with a specific focus on tackling the complex edge cases responsible for the remaining failures in the OpenModelica compliance test suite. Furthermore, to support even larger models under strict browser memory limits, future work will extend the contiguous arena memory allocation strategy—currently used for the DAE representation—further up the compiler pipeline to accelerate earlier phases like syntax parsing and semantic indexing.

Acknowledgements

During the preparation of this work, the author utilized Google Gemini 3.1 Pro and Anthropic Claude 3.6 Opus to aid in structural outlining, language polishing, and code generation for the ModelScript compiler. After using these tools, the author reviewed, edited, and verified the content and takes full responsibility for the final content of the publication.

References

Bajaj, Manas, Dirk Zwemer, Russell Peak, et al. (2016). "Integration of systems engineering and mechanical CAD using

- the digital thread”. In: *2016 Annual IEEE Systems Conference (SysCon)*, pp. 1–8.
- Brost, RC, DR Strip, and PJ Eicker (1993). “The technology base for agile manufacturing”. In: *6th Annual Workshop on Space Operations Applications and Research (SOAR 1992)*, Volume 2, pp. 66–71.
- Elaasar, Maged, Nicolas Rouquette, David Wagner, Bentley James Oakes, Abdelwahab Hamou-Lhadj, and Mohammad Hamdaqa (2023). “openCAESAR: Balancing agility and rigor in model-based systems engineering”. In: *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 221–230.
- Fritzson, Peter, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman (2005). “The OpenModelica modeling, simulation, and development environment”. In: *Proceedings of the 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS 2005)*, pp. 83–90.
- Fritzson, Peter and Vadim Engelson (1998). “Modelica—A unified object-oriented language for system modeling and simulation”. In: *European Conference on Object-Oriented Programming*, pp. 67–90.
- Hooshmand, Yousef (2025). “Data-Centric Systems Engineering (DCSE) based on Semantic Digital Thread (SeDT)”. In: *Proceedings of PLM Road Map & PDT Europe 2025*, pp. 1–25.
- INCOSE (2021). *Systems Engineering Vision 2035: Engineering Solutions for a Better World*. Tech. rep. International Council on Systems Engineering (INCOSE).
- Nachawati, Mohamad Omar, Gianmaria Bullegas, Andrey Vasilyev, Joe Gregory, Adrian Pop, Maged Elaasar, and Adeel Asghar (2022). “Towards an open platform for democratized model-based design and engineering of cyber-physical systems”. In: *Proceedings of the American Modelica Conference 2022*, pp. 102–114.
- Object Management Group (OMG) (2012). *SysML-Modelica Transformation Specification, Version 1.0*. Tech. rep. Object Management Group.
- Reiterer, Stefan H., Clemens Schiffer, and Mario Schwaiger (2023). “A Graph-Based Meta-Data Model for DevOps: Extensions to SSP and SysML2 and a Review on the DCP Standard”. In: *Proceedings of the 15th International Modelica Conference*, pp. 159–166.
- Stark, John (2011). “Product Lifecycle Management”. In: *Product Lifecycle Management: 21st Century Paradigm for Product Realisation*, pp. 1–16.
- Sun, Yiyao, Chengwei Yang, Yu Yang, and Yunlu Yu (2026). “Integrated SysML/Modelica-based modeling and validation of a lightweight autonomous aerial propulsion system”. In: *Simulation Modelling Practice and Theory*.
- Tolentino, Glenn and John Wood (2018). “Balancing Systems Engineering Rigor with Agile Software Development Flexibility”. In: *INSIGHT*, pp. 25–28.

Listing 3. SysML v2 architectural structure defining the logical battery pack hierarchy, constraints, and requirements verification.

```

package BatteryArchitecture_N {
  part def BatteryCellSysML {
    attribute temp: Real;
  }
  part def BatteryPackSysML {
    attribute T_ambient: Real = 25.0;
    part cells: BatteryCellSysML[N];
    // OVERCONSTRAINT: Hardcoded constant
    // conflicts with DAE differential eq for T
    [1]
    // constraint { cells[1].temp == 50.0 }
  }
  // Import the physical implementation
  import BatteryPack_N;
  // Instantiate the fully realized model
  part pack_instance: BatteryPack_N;
  verification def MaxTemperatureCheck {
    objective max_temp {
      subject target : BatteryPack_N;
      verify pack_instance.cells[N / 2]
        .temp <= 65.0;
    }
  }
}

```

Listing 4. Modelica implementation extending the SysML v2 architecture to inject the physical 1D thermal dynamics of the N -component battery pack.

```

model BatteryCell
  extends BatteryCellSysML(temp(start=0.0));
  Real power;
equation
end BatteryCell;
model BatteryPack_N
  import BatteryArchitecture_N.*;
  extends BatteryPackSysML(
    redeclare BatteryCell cells[N]
  );
  parameter Integer N_cells = N;
  parameter Real C = 5.0;
  parameter Real R = 0.1;
equation
  for i in 1:N_cells loop
    cells[i].power = 10.0;
    C * der(cells[i].temp) = cells[i].power
      - (cells[i].temp - T_ambient) / R
      - (if i > 1 then
        (cells[i].temp - cells[i-1].temp) / R
      else 0.0)
      - (if i < N_cells then
        (cells[i].temp - cells[i+1].temp) / R
      else 0.0);
  end for;
  annotation(experiment(
    StartTime=0, StopTime=10.0, Interval=0.05
  ));
end BatteryPack_N;

```

APPENDICES

A Code for Cross-Domain Verification

MMChvdcLibrary: Modeling Arm Average Value Modular Multilevel Converters in Modelica for Power System Simulation

Md Nazmul Hassan Chowdhury¹ Luigi Vanfretti¹

¹Electrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY, USA,
{chowdm5,vanfrel}@rpi.edu

Abstract

High Voltage Direct Current (HVDC) power transmission based on the Modular Multilevel Converters (MMCs) has become the reference topology for modern HVDC links and future multi-terminal DC grids, driven by renewable integration and long-distance power transfer needs. While electromagnetic transient models accurately capture switching behavior, their computational cost limits their use in large-scale system studies, motivating the development of phasor-domain arm average value representations. This paper introduces *MMChvdcLibrary*, a Modelica library developed in Dymola for phasor-domain arm average value modeling of MMC-based HVDC systems. The library provides modular converter models, control blocks, sensors, and mathematical operators, assembled into station-level models supporting P - Q , V_{dc} - Q , and V_{ac} phasor control modes, along with a user-configurable master HVDC station integrating all three. Examples are presented to demonstrate and validate the library across these operating modes. The library is distributed with the BSD 3-Clause "New" or "Revised" License and is available online at: <https://github.com/ALSETLab/MMChvdcLibrary/>

Keywords: Arm average-value model, Modular multilevel converter (MMC), HVDC, Power Systems, OpenIPSL

1 Introduction

The increasing share of power-electronics-interfaced generation driven by offshore wind integration and the need to interconnect asynchronous AC areas is reshaping how transmission systems are planned and simulated. HVDC transmission based on MMC has become a cornerstone of this transition, offering superior efficiency, controllability, and scalability over earlier VSC topologies (Samimi, Gruson, Delarue, et al. 2016; Shinoda et al. 2018). As deployments grow, so does the need for simulation tools that are both physically accurate and computationally tractable at system scale. This paper presents *MMChvdcLibrary*, a Modelica library implemented in Dymola that provides phasor-domain arm average value model of the MMC converter and the MMC-HVDC stations across multiple operating modes, designed for seamless integration with standard AC network simulation environments.

1.1 Motivation

The MMC is now the reference topology for HVDC links and envisioned Multi-Terminal DC (MTDC) grids (Freytes et al. 2016; Samimi, Gruson, Delarue, et al. 2016). Beyond its low switching losses and modular construction, its defining feature is the ability to store and control energy in distributed submodule capacitors independently of the DC bus voltage — a degree of freedom absent in classical VSC designs, and one that creates a corresponding modeling challenge (Saad et al. 2015; Shinoda et al. 2018). Operationally, an MMC station may regulate active power, control the DC bus voltage (Samimi, Gruson, Guillaud, et al. 2015; Samimi, Gruson, Delarue, et al. 2016), or impose a stable AC voltage when supplying a passive or islanded network such as an offshore wind farm (Chai et al. 2013). Electromagnetic Transient (EMT) models reproduce switching-level phenomena accurately but are prohibitively expensive for grids with many AC buses, machines, and HVDC terminals (Jovcic and Jamshidifar 2015; Freytes et al. 2016). Phasor-domain models that preserve the essential average dynamics — including internal energy — are therefore essential for transient stability workflows. The Modelica language, with its acausal, equation-based paradigm, is well suited to such component libraries and has been used effectively for AC stability studies (Mukherjee et al. 2021) and grid-forming converter assessment (Gonzalez-Torres et al. 2021). A structured, multi-mode, openly available MMC-HVDC Modelica library nevertheless remains absent from the literature, and *MMChvdcLibrary* addresses this gap.

1.2 Related Works

The Arm Average Value Model (AAVM) formalized in (Saad et al. 2015) replaces each arm's submodule stack with an equivalent controlled voltage source and lumped capacitor, yielding an eleven-state representation that serves as the starting point for further simplifications (Freytes et al. 2016; Gonzalez-Torres et al. 2021). Building on this, energy-based control frameworks have established the MMC's stored energy as a controllable degree of freedom: (Samimi, Gruson, Guillaud, et al. 2015; Samimi, Gruson, Delarue, et al. 2016) showed that coupling the energy reference to the square of the DC voltage effectively increases the apparent DC-side capacitance, and (Shinoda et al. 2018) generalized this through Vir-

tual Capacitor Control with a tunable coefficient, k_{vc} . For passive-network operation, (Chai et al. 2013) developed a dq -frame AC voltage controller combining feed-forward compensation with decoupled current PI loops — the basis for the AC voltage mode adopted here. Grid-following modes require a PLL; (Ortega and Milano 2018) compared five implementations and found the LPF-PLL offered the best responsiveness–noise trade-off for system-level studies. For phasor-domain MMC representation, (Jovcic and Jamshidifar 2015) derived an analytical model in rotating dq and d_2q_2 frames, while (Freytes et al. 2016) reduced the AAM to four states by aggregating arm capacitors into a single $C_{eq} = 6C_{tot}$ and per-phase differential currents into one i_{dc} , validated against a 401-level EMT reference. This reduced model was subsequently implemented in Modelica using OpenIPSL (Gonzalez-Torres et al. 2021; Mukherjee et al. 2021), with which MMChvdcLibrary is designed to be compatible.

1.3 Contributions

This paper presents MMChvdcLibrary, a Modelica library implemented in Dymola for phasor-domain arm average value modeling of MMC-based HVDC systems. The library is grounded in the simplified MMC model of (Freytes et al. 2016), which preserves the converter's internal energy dynamics while remaining computationally efficient. The control structure follows the energy-based framework established in (Samimi, Gruson, Guillaud, et al. 2015; Samimi, Gruson, Delarue, et al. 2016; Shinoda et al. 2018), supporting both constant and DC-voltage-linked energy references. The AC voltage control mode implemented in the library builds on the passive-network control strategy developed in (Chai et al. 2013), extended to the phasor-domain setting. The library is structured around two main packages. The ComponentLibrary provides modular building blocks including converter models, inner and outer loop control blocks, basic functions like sensors and mathematical operations. The MMC_HVDCstation package assembles these components into validated station-level models supporting three operating modes: P - Q control, V_{dc} - Q control, and V_{ac} phasor control. A user-configurable MasterHVDCstation integrates all three modes into a single reconfigurable station model. Example models are provided to validate each operating mode against reference responses consistent with established EMT-validated simplified MMC models from the literature. The specific contributions of this paper are:

- A structured, reusable Modelica library for phasor-domain MMC-HVDC simulation encompassing three physically motivated operating modes and composable control components.
- Three physically motivated, ready-to-use station models (P - Q , V_{dc} - Q , V_{ac}) plus a reconfigurable Master HVDC station that integrates all three under a Boolean mode-selection input.

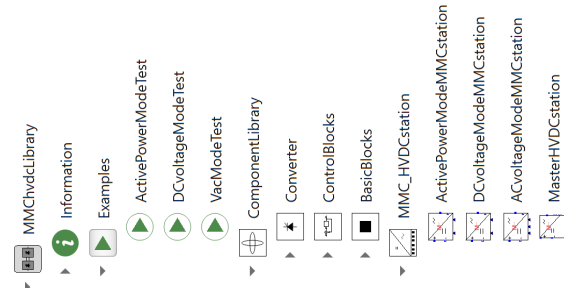


Figure 1. MMChvdcLibrary package structure.

- Demonstration of the library's capability for efficient MMC-HVDC simulation in a Modelica-based environment compatible with standard AC network libraries.

2 Library Overview

The package structure of the MMChvdcLibrary is shown in Figure 1. Test cases for each operating mode of the MMC-HVDC station are located in the Examples package, which contains ActivePowerModeTest, DCvoltageModeTest, and VacModeTest. These examples demonstrate the dynamic response of the HVDC station under active power, DC voltage, and AC voltage control modes, respectively. The ComponentLibrary package contains the modular building blocks from which the MMC-HVDC stations are assembled. It is organized into three sub-packages. The Converter package contains the phasor-domain AAVM MMC model, derived from the simplified representation of (Freytes:2016) that preserves the converter's internal energy dynamics while reducing the switching details. The ControlBlocks package contains the inner and outer control loops, including AC current controllers in the dq frame, DC current controllers, energy controllers, DC voltage controllers, active and reactive power controllers, the AC voltage controllers, and a Synchronization block for grid synchronization. The BasicBlocks package contains general-purpose mathematical operators, transformations, and sensors used throughout the library, such as dq transformations, power calculation, and interfaces. The MMC_HVDCstation package assembles the components from the ComponentLibrary into station-level models. It contains four ready-to-use station models: ActivePowerModeMMCstation for P - Q control, DCvoltageModeMMCstation for V_{dc} - Q control, ACvoltageModeMMCstation for V_{ac} phasor control, and MasterHVDCstation, a user-configurable station that integrates all three operating modes and exposes a mode-selection parameter so the user can reconfigure the same component instance for different roles in an HVDC link. The Information package contains library documentation, including a description of the underlying converter model, the control architecture of each station, and usage notes for the example models. The library depends on the other following libraries:

- Modelica Standard Library, v4.1.0 (Modelica Association 2023).
- OpenIPSL, v3.1.0 (Baudette:2018), for compatibility with phasor-domain AC network components in system-level studies.

3 Component Library

The `ComponentLibrary` package, shown in Figure 1, provides the fundamental building blocks for simulating MMC-based HVDC systems. It is organised into three sub-packages: `Converter`, `ControlBlocks`, and `BasicBlocks`. Each sub-package is described in the following subsections.

3.1 Converter subpackage

The `Converter` subpackage contains the MMC model used throughout the library. The core model, `AvgMMCmodel`, is a simplified AAVM model of the MMC that captures the converter's internal energy dynamics without modeling individual switching events. This simplification makes the model suitable for transient stability studies while reducing the computational cost. The phasor domain AAVM MMC circuit is shown in Figure 2 whose dynamic behavior can be expressed by the following equations:

$$v_{md} - v_{gd} = L_{eq}^{ac} \frac{di_{gd}}{dt} + R_{eq}^{ac} i_{gd} - \omega L_{eq}^{ac} i_{gq} \quad (1)$$

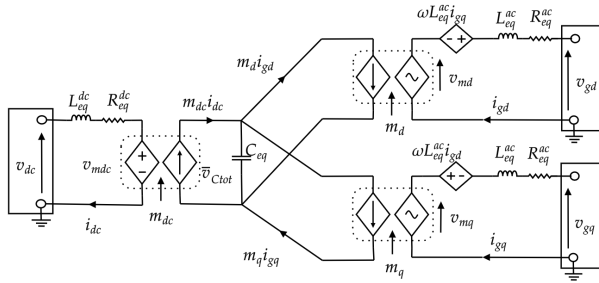


Figure 2. Phasor domain MMC model.

$$v_{mq} - v_{gq} = L_{eq}^{ac} \frac{di_{gq}}{dt} + R_{eq}^{ac} i_{gq} + \omega L_{eq}^{ac} i_{gd} \quad (2)$$

$$v_{dc} - v_{mdc} = L_{eq}^{dc} \frac{di_{dc}}{dt} + R_{eq}^{dc} i_{dc} \quad (3)$$

$$C_{eq} \frac{d\bar{v}_{C_{tot}}}{dt} = m_{dc} i_{dc} - m_d i_{gd} - m_q i_{gq} \quad (4)$$

where

$$m_d = \frac{v_{md}}{\bar{v}_{C_{tot}}}, \quad m_q = \frac{v_{mq}}{\bar{v}_{C_{tot}}}, \quad m_{dc} = \frac{v_{mdc}}{\bar{v}_{C_{tot}}} \quad (5)$$

$$R_{eq}^{dc} = \frac{2R_{arm}}{3}, \quad L_{eq}^{dc} = \frac{2L_{arm}}{3} \quad (6)$$

$$R_{eq}^{ac} = \frac{R_{arm} + 2R_f}{2}, \quad L_{eq}^{ac} = \frac{L_{arm} + 2L_f}{2} \quad (7)$$

The modulated voltages v_{md} , v_{mq} , and v_{mdc} serve as control inputs that regulate the converter dynamics, along with the corresponding grid voltages v_{gd} and v_{gq} . Besides, i_{gd} and i_{gq} represent the current components of the grid on the d - and q -axes, and ω is the frequency of the AC system. The average voltage of the arm-capacitor is $\bar{v}_{C_{tot}}$, with an equivalent capacitance $C_{eq} = 6C_{tot}$. The modulation indices are m_d , m_q , and m_{dc} . The equivalent AC parameters are R_{eq}^{ac} and L_{eq}^{ac} , while the DC parameters are R_{eq}^{dc} and L_{eq}^{dc} , where R_{arm} , L_{arm} , R_f , and L_f denote the resistances and inductances of the arm and transformer.

The Dymola graphical implementation of the `AvgMMCmodel` is depicted in Figure 3. The main electrical circuit is implemented with sources, resistors, inductors, capacitors, and pins from the Modelica Standard library. These components are used to represent the equivalent electrical structure of the arm average value MMC model. The mathematical operations, scaling, and multiplication of internal signals required by the converter model are also implemented using the Modelica Standard Library. The `RealToComplex` block from `Modelica.ComplexBlocks.ComplexMath` is used to convert real-valued signals into complex variables. This conversion is required because the AC side of the converter is connected to the phasor-domain interface of OpenIPSL. besides, implementing the AC-side

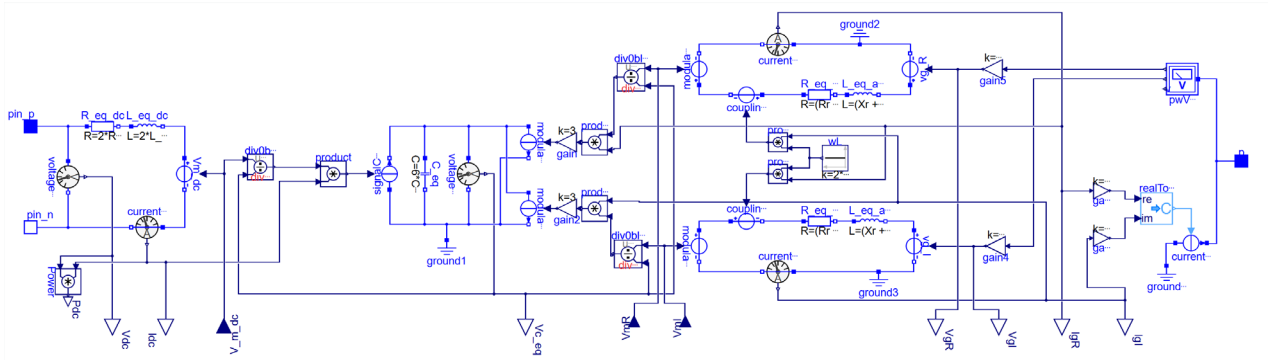


Figure 3. Phasor domain MMC model implementation in Dymola.

interface, the OpenIPSL Library is used to implement the converter terminal voltage, and the division operation inside the converter.

Model initialization is performed in the Dymola text layer, an excerpt of which is shown in Listing 1. The user supplies power-flow data as parameters such as the connection-bus voltage magnitude V_{gd} and its angle $teta_bus$. These are then converted into the complex terminal voltage V_{t0} used to initialize the AC-side state variables.

Listing 1. Initialization excerpt of AvgMMCmodel.

```
parameter Real Vgd = Vgd
  "Connection bus voltage magnitude (V)";
parameter SI.Angle teta_bus = teta_bus
  "Connection bus voltage angle (rad)";
parameter Complex Vt0 =
  CM.fromPolar(Vgd/sqrt(3), teta_bus)
  "Connection bus voltage in complex form";
```

3.2 ControlBlocks Subpackage

The `ControlBlocks` subpackage contains the control-system components that govern the dynamic behavior of the HVDC stations. It is divided into two layers: `OuterLoopControls` and `InnerLoopControls`.

3.2.1 OuterLoopControls

The outer-loop controllers act on system-level objectives and provide reference signals to the inner-loop controllers. The subpackage includes:

- **EnergyControl:** maintains energy balance between the AC and DC sides by adjusting the DC-link voltage reference.
- **ActivePowerControl:** regulates the active power exchange between the converter and the AC grid.
- **ReactivePowerControl:** governs reactive-power exchange to support AC voltage regulation.
- **DCvoltageControl:** keeps the DC-link voltage at its setpoint.
- **ACvoltageControl:** maintains the AC terminal voltage.

The `ActivePowerControl` block is used here as a representative example of the outer-loop controller implementation, as shown in Figure 6. The block receives the active power reference, the measured active power, and the d -axis terminal voltage as real-valued input signals. The difference between the reference and measured active power is calculated using an add block. This signal is then processed by a PI controller. The PI-controller output is divided by the terminal voltage, and the resulting signal is provided as the d -axis AC current reference.

The gain and time constant of the PI controller are propagated to the higher-level station model. Therefore, the controller can be tuned directly from the station interface. Other outer-loop controllers in the library are implemented in a similar way. The PI-controller gain k and time

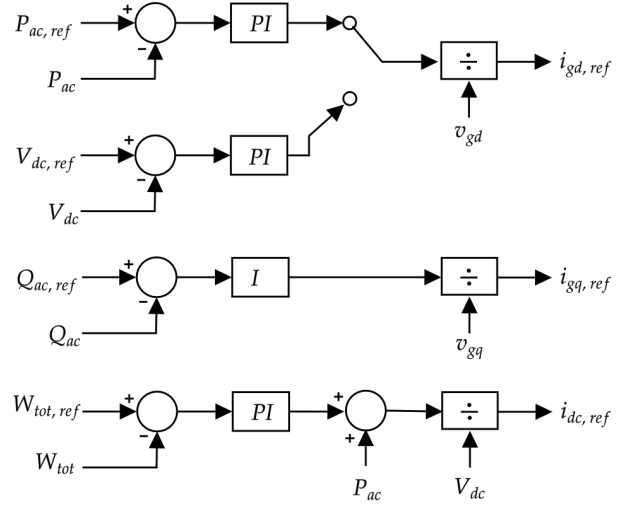


Figure 4. Block diagrams of the outer-loop controllers.

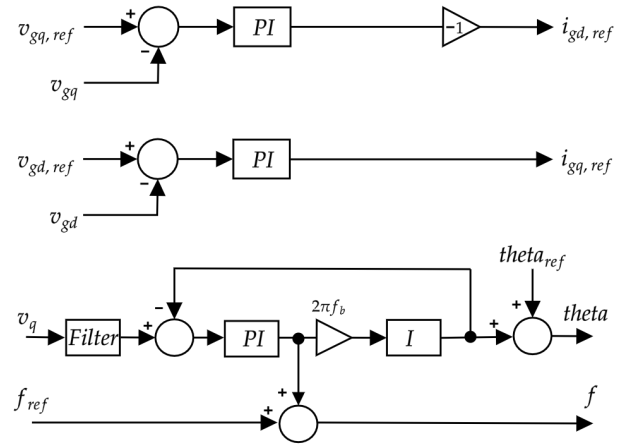


Figure 5. Block diagram of the AC voltage phasor controller.

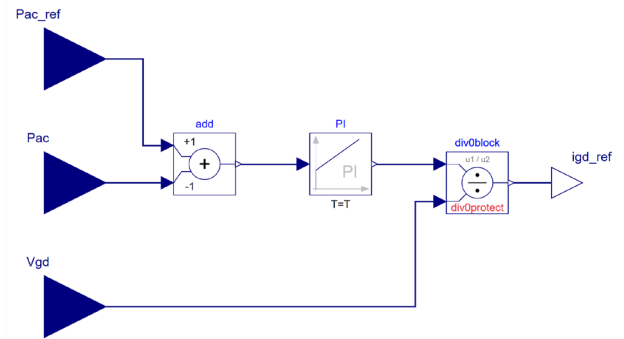


Figure 6. Implementation of `ActivePowerControl`.

constant T are propagated to the enclosing station model through the text layer shown in Listing 2. Other outer-loop controllers in the library are implemented in the same way.

Listing 2. Parameter propagation in `ActivePowerControl`.

```
parameter Real k = k "Gain";
parameter Modelica.Units.SI.Time T
  "Time Constant (T>0 required)";
```

3.2.2 InnerLoopControls

The inner-loop controllers shown in Figure 7 operate on the fast current dynamics of the converter and are driven by the reference signals produced by the outer-loop layer. The subpackage includes:

- **DCcurrentControl:** regulates the DC current to support the DC-voltage control objective.
- **ACcurrentControl:** enforces the d- and q-axis current references on the AC side, ensuring that the converter injects the correct amount of active and reactive power.

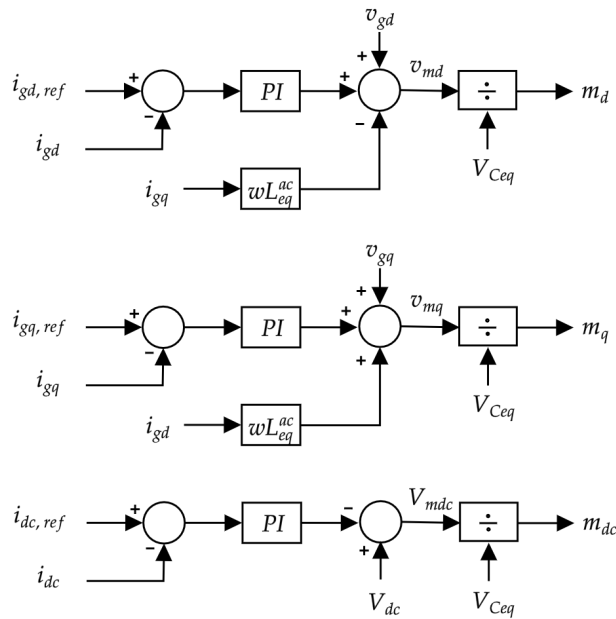


Figure 7. Block diagrams of the inner-loop controllers.

The DCcurrentControl block is used here as a representative example of the inner-loop controller implementation, as shown in Figure 8. This block receives four real-valued input signals: the DC current reference $i_{dc,ref}$, the measured DC current i_{dc} , the DC-side voltage V_{dc} , and

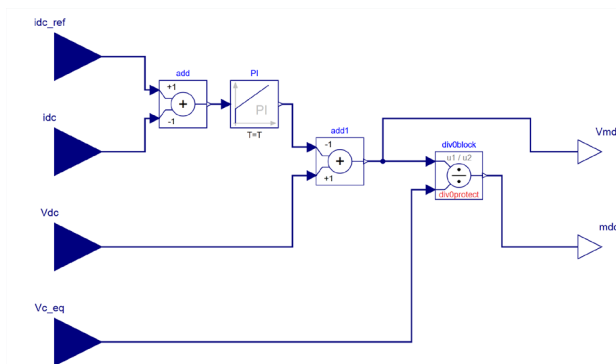


Figure 8. Implementation of DCcurrentControl

the equivalent capacitor voltage V_{Ceq} . The difference between the reference and measured DC current is first calculated and then processed by a PI controller. The controller output is combined with the measured DC voltage to generate the modulated DC voltage signal V_{mdc} . The resulting voltage signal is divided by the equivalent capacitor voltage V_{Ceq} to calculate the DC modulation m_{dc} . The gain and time constant of the PI controller are propagated to the higher-level station model so that the DC current controller can be tuned directly.

3.3 BasicBlocks Subpackage

The BasicBlocks are used to process signals, apply transformations, and perform calculations, supporting the overall control system and converter operations. The subpackage includes PhasorModels for AC voltage and current representation in phasor form, Calculator blocks for active power, reactive power, and energy computations, Sensors for real-time measurements, Transformation blocks for reference-frame conversions (dq/abc), and PLL blocks for grid synchronization.

Whereas most components are assembled graphically from MSL and OpenIPSL primitives, some require a purely equation-based implementation in the Dymola text layer. The ACpowerSensor model from ComponentLibrary.BasicBlocks.Sensors is a representative example and its complete text-layer implementation is shown in Listing 3.

Listing 3. Text-layer implementation of ACpowerSensor.

```

model ACpowerSensor "Measure the AC power"
  outer OpenIPSL.Electrical.SystemBase
    SysData;
  import Modelica.ComplexMath.conj;
  import Modelica.ComplexMath.real;
  import Modelica.ComplexMath.imag;
  import Modelica.ComplexMath.j;
  parameter Real S_b = SysData.S_b
    "System base power (MVA)";
  Real P12PU; Real P21PU;
  Real Q12PU; Real Q21PU;
  OpenIPSL.Types.ActivePower P12, P21;
  OpenIPSL.Types.ReactivePower Q12, Q21;
  Complex vs(re=p.vr, im=p.vi);
  Complex is(re=p.ir, im=p.ii);
  Complex vr(re=n.vr, im=n.vi);
  Complex ir(re=n.ir, im=n.ii);
  OpenIPSL.Interfaces.PwPin p;
  OpenIPSL.Interfaces.PwPin n;

equation
  P12PU = real(vs*conj(is));
  P21PU = real(vr*conj(ir));
  Q12PU = imag(vs*conj(is));
  Q21PU = imag(vr*conj(ir));
  P12 = real(vs*conj(is))*S_b;
  P21 = real(vr*conj(ir))*S_b;
  Q12 = imag(vs*conj(is))*S_b;
  Q21 = imag(vr*conj(ir))*S_b;
  complex(0,0) = is + ir;
  complex(0,0) = vs - vr;
end ACpowerSensor;

```

4 MMC HVDC Station Models

The MMC_HVDCstation package integrates the components described in Section 3 into four complete, ready-to-use HVDC station models. Each station is parameterizable and exposes all relevant controller parameters at the top level.

4.1 ActivePowerModeMMCstation

The ActivePowerModeMMCstation, illustrated in Figure 9, operates under $P-Q$ control, i.e., it simultaneously regulates the active power and the reactive power exchange. The station is assembled from five functional zones labeled A–E in Figure 9:

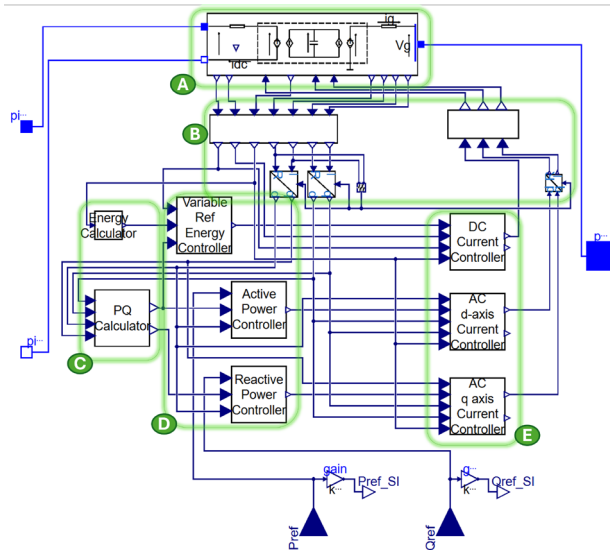


Figure 9. ActivePowerModeMMCstation implementation.

- **Zone A:** the AvgMMCmodel converter from ComponentLibrary.Converter, which provides the electrical interface between the DC link and the AC network.
- **Zone B:** the components gathered from the ComponentLibrary.BasicBlocks.Transformation that converts electrical units between per-unit and SI, and also converts AC quantities into the rotating dq frame required by the controllers.
- **Zone C:** the measurement component blocks like PQcalculator and EnergyCalculator from ComponentLibrary.BasicBlocks.Calculator, providing active power, reactive power, and energy measurements to the feedback loops.
- **Zone D:** the outer-loop controller blocks like EnergyControl, ActivePowerControl, and ReactivePowerControl from ComponentLibrary.ControlBlocks.OuterLoopControls, responsible for maintaining the desired active and reactive power in the system.

- **Zone E:** the inner-loop controller blocks such as the DCcurrentControl, dAxisACcurrentControl, and qAxisACcurrentControl from ComponentLibrary.ControlBlocks.InnerLoopControls, regulate the DC and AC currents, ensuring precise current control to maintain system stability.

4.2 DCvoltageModeMMCstation

The DCvoltageModeMMCstation, shown in Figure 10, operates under $V_{dc}-Q$ control. In this mode the DC-link voltage is the primary controlled variable, while reactive-power exchange with the AC grid is independently regulated. Compared to the ActivePowerModeMMCstation, the DCvoltageControl block replaces ActivePowerControl in the outer-loop layer (Zone D). This control architecture is typically used for the slack-bus converter in a multi-terminal HVDC network that must absorb power imbalances caused by generation variations at other stations.

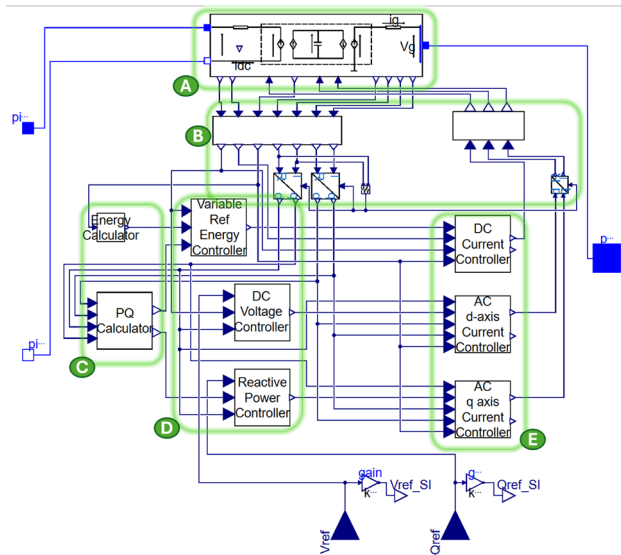


Figure 10. DCvoltageModeMMCstation implementation.

4.3 ACvoltageModeMMCstation

The ACvoltageModeMMCstation, depicted in Figure 11, is designed to control the AC terminal voltage of a passive network by providing AC voltage phasor control, maintaining both magnitude and angle of the AC bus voltage at their reference values. The ACvoltageControl block takes the place of the power-regulating blocks in the outer-loop layer (Zone D), making this station the appropriate choice for energizing an islanded AC network. Along with all the conversion blocks at zone B, a dedicated Synchronization component from ComponentLibrary.BasicBlocks.PLL provides the synchronization angle required by the dq transformations throughout the station.

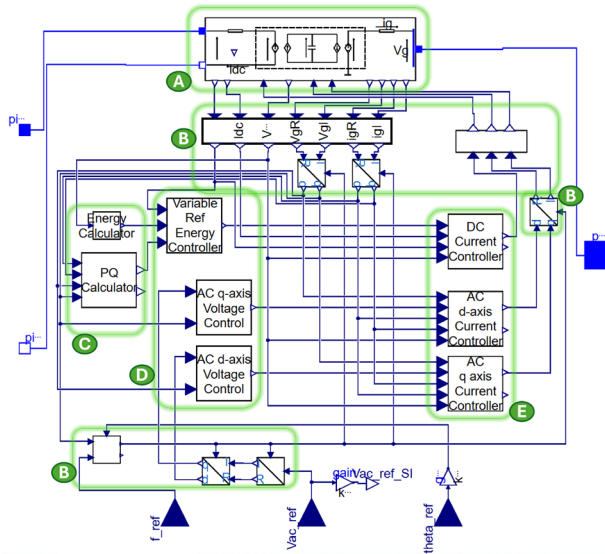


Figure 11. AC voltage Mode MMC station implementation.

4.4 MasterHVDCStation

The MasterHVDCStation, shown in Figure 12, is a user-configurable station model that combines the three operating modes. This model allows the user to select the desired operating mode from a single component by using two Boolean parameters, *Vdc_Pac_flag* and *Vac_Qac_flag*. The control mode is selected using the logic in Table 1. The main practical modes used in this work are P_{ac} - Q_{ac} , V_{dc} - Q_{ac} , and V_{ac} control.

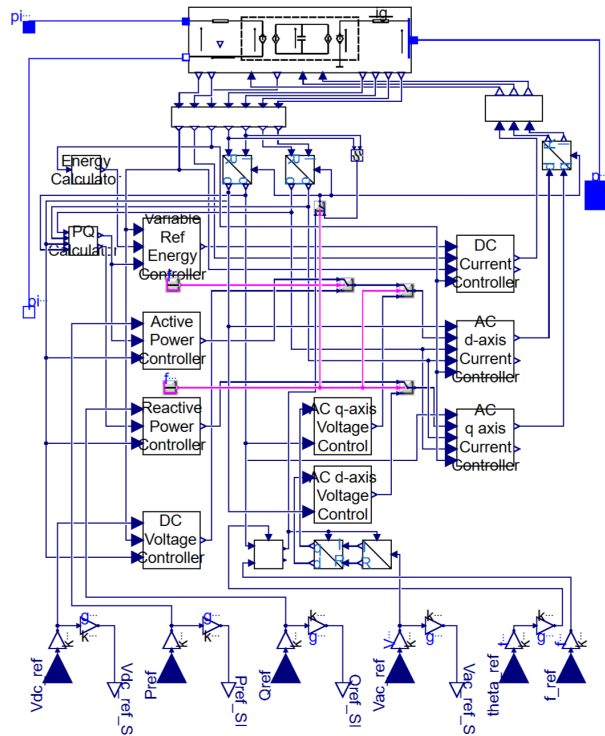


Figure 12. MasterHVDCStation implementation.

Table 1. Mode-selection logic of the MasterHVDCStation.

<i>Vdc_Pac_flag</i>	<i>Vac_Qac_flag</i>	Control mode
true	true	P_{ac} - Q_{ac}
false	true	V_{dc} - Q_{ac}
true	false	V_{ac}
false	false	No control

Listing 4 shows the implementation used for the mode-selection logic. The Boolean parameters *Vdc_Pac_flag* and *Vac_Qac_flag* are first defined by the user. These parameters are then passed to Boolean constant blocks, allowing the selected mode to be used within the graphical model. The listing also includes several unity-gain blocks with the suffix *_mock*, such as *Vdc_ref_mock*, *Pref_mock*, *Qref_mock*, and *Vac_ref_mock*. These blocks are used as controlled signal paths for references that are inactive in the selected mode. For example, when *Vdc_Pac_flag* is true, the active power control path is selected and the DC voltage reference path is disabled by setting the input of *Vdc_ref_mock* to zero.

Listing 4. Text-layer implementation of mode selection.

```

model MasterHVDCStation
import CM = Modelica.ComplexMath;
import      Modelica.Units.SI;
import      Complex;
parameter Boolean Vdc_Pac_flag = true "
  True-Pac(Vac) mode; False-Vdc mode";
parameter Boolean Vac_Qac_flag = true "
  True-Qac mode; False-Vac mode";

...
public
Modelica.Blocks.Sources.BooleanConstant
  flag2(k=Vac_Qac_flag);
Modelica.Blocks.Sources.BooleanConstant
  flag1(k=Vdc_Pac_flag);
Modelica.Blocks.Math.Gain Vdc_ref_mock(k=
  1);
Modelica.Blocks.Math.Gain Pref_mock(k=1);
Modelica.Blocks.Math.Gain Qref_mock(k=1);
Modelica.Blocks.Math.Gain Vac_ref_mock(k=
  1);
Modelica.Blocks.Math.Gain theta_ref_mock(
  k=1);
Modelica.Blocks.Math.Gain f_ref_mock(k=1)
;
equation
if Vdc_Pac_flag then
  Vdc_ref_mock.u = 0;
end if;
if Vac_Qac_flag then
  Vac_ref_mock.u = 0;
  theta_ref_mock.u = 0;
  f_ref_mock.u = 0;
end if;
if not Vdc_Pac_flag then
  Pref_mock.u = 0;
end if;
if not Vac_Qac_flag then
  Qref_mock.u = 0;
end if;
end MasterHVDCStation;

```

5 Simulation Examples

The Examples package contains three test models that are used to demonstrate the operating modes implemented in the MasterHVDCstation. The three examples correspond to P_{ac} - Q_{ac} control, V_{dc} - Q_{ac} control, and V_{ac} phasor control.

5.1 Active Power Mode Test

The first example shows the operation of the station in active and reactive power control mode. The circuit used for this test is shown in Figure 13a. In this example, the ActivePowerModeMMCstation is connected between a DC voltage source and an AC grid equivalent. The AC side is connected to an infinite bus through a measurement block, while the active and reactive power references are provided as external input signals. At $t = 1.5$ s, the active-power reference $P_{ac,ref}$ is stepped from 0.2 pu to 0.3 pu (i.e., from 200 MW to 300 MW on a 1000 MVA base). Figure 13b shows the active power output of the station together with its reference. The active-power output P_{ac} reaches its new reference within approximately 200 ms, with a response shape set by the tuning of the d-axis inner current loop. Figure 13c shows the reactive power output and its reference. The reactive power Q_{ac} remains close to the specified reference 0.01 pu (i.e., 10 MVar on a 1000 MVA base) during the step change of the active power. This result indicates that the implemented P_{ac} - Q_{ac} control structure can regulate the two power channels independently in this test case.

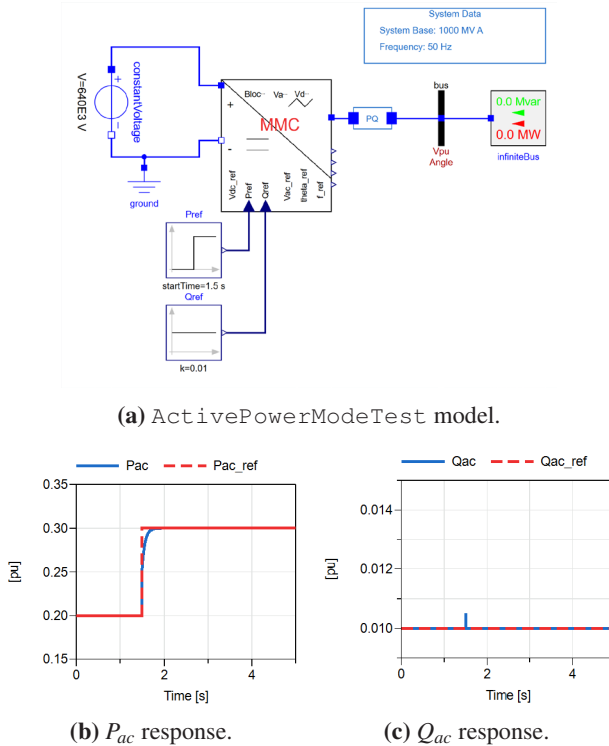


Figure 13. P - Q mode test. (a) The test model, (b) active power response, and (c) reactive power responses.

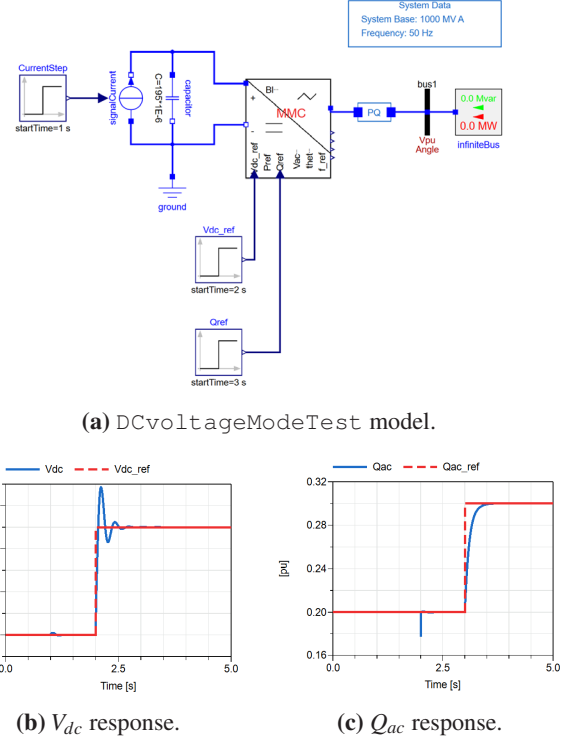


Figure 14. V_{dc} mode test. (a) The test model, (b) DC voltage response, and (c) reactive power responses.

5.2 DC Voltage Mode Test

The second example shows the operation of the station in DC voltage and reactive power control mode. The circuit used for this test is shown in Figure 14a. In this example, the DCvoltageModeMMCstation is connected between a DC-side current source and an AC grid equivalent. The AC side is connected to an infinite bus through a measurement block, while the DC voltage and reactive power references are provided as external input signals. The DC-side current source has an offset of 0.5 pu and is stepped to 0.6 pu at $t = 1$ s to create a disturbance on the DC side of the converter. At $t = 2$ s, the DC voltage reference $V_{dc,ref}$ is stepped from 1.0 pu to 1.5 pu, and at $t = 3$ s, the reactive-power reference $Q_{ac,ref}$ is stepped from 0.2 pu to 0.3 pu, corresponding to a change from 200 MVar to 300 MVar on a 1000 MVA base. The three events are separated in time so that the rejection of disturbance (current source step) and reference tracking ($V_{dc,ref}$ and $Q_{ac,ref}$) can be independently assessed.

Figure 14b shows the DC voltage response of the station together with its reference. The current disturbance at $t = 1$ s produces only a small excursion in V_{dc} , of the order of 0.02 pu, which diminishes within approximately 0.1 s; the disturbance is first absorbed by the arm capacitors, represented by the equivalent capacitance $C_{eq} = 6C_{tot}$, before the DC voltage controller acts. The $(V_{dc,ref}$ step at $t = 2$ s produces an underdamped tracking response, with a peak of about 1.68 pu and a damped oscillation that settles

in approximately 0.9 s, without steady-state error. This oscillation arises from the interaction between the outer DC-voltage loop and the inner DC-current loop through the converter's stored energy: a change in $V_{dc,ref}$ alters the energy held in C_{eq} , and the energy controller adjusts the DC-link voltage reference accordingly.

Figure 14c shows the reactive power output and its reference. Q_{ac} is unaffected by the DC-side disturbance at $t = 1$ s, and the DC voltage step at $t = 2$ s causes only a brief dip that recovers quickly. After the reference step at $t = 3$ s, Q_{ac} reaches 0.3 pu without overshoot. This result indicates that the implemented V_{dc} - Q_{ac} control structure can regulate the DC-side voltage while controlling the reactive power exchange with the AC network.

5.3 AC Voltage Mode Test

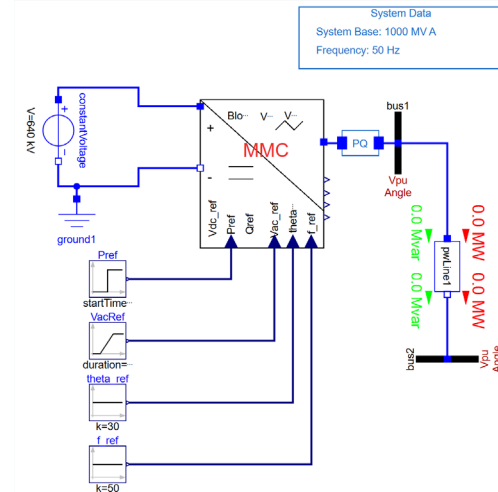
The third example shows the operation of the station in AC voltage phasor control mode. The circuit used for this test is shown in Figure 15a. In this example, the ACvoltageModeMMCstation is connected to an AC network representation at the receiving side. The voltage magnitude, phase angle, and frequency references are provided as external input signals. This operating mode is intended to test whether the station can regulate the AC terminal voltage phasor directly, rather than regulating active power, reactive power, or DC voltage.

At $t = 1$ s, the AC voltage magnitude reference $|V_{ac,ref}|$ is increased from zero to 1.0 pu using a ramp input. At the same time, the voltage angle reference θ_{ref} is changed to the specified operating value. Figure 15b shows the AC voltage magnitude response together with its reference. The terminal voltage magnitude $|V_{ac}|$ follows the commanded ramp and reaches the specified value. Figure 15c shows the voltage angle response and its reference. The angle θ follows the commanded reference after the reference change. This result indicates that the implemented V_{ac} control structure can regulate both the magnitude and angle of the AC terminal voltage phasor in this test case.

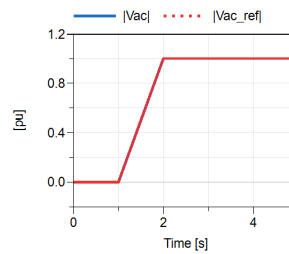
6 Conclusion and Future Work

This paper presented MMChvdcLibrary, a Modelica library developed in Dymola for phasor-domain arm average value modeling of MMC-based HVDC systems. The library is organized in a modular structure, where the converter model, control blocks, basic measurement and transformation blocks, and station-level models are implemented as reusable components. This organization allows users to assemble and test MMC-HVDC stations under different operating modes without rebuilding the model structure for each case.

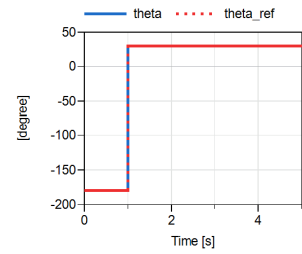
The implemented station models support three main operating modes: P_{ac} - Q_{ac} control, V_{dc} - Q_{ac} control, and V_{ac} phasor control. A user-configurable MasterHVDCstation was also developed to integrate these modes into a single component. The simulation examples demonstrated the operation of the library in each control mode. The ActivePowerModeTest showed



(a) ACvoltageModeTest model.



(b) $|V_{ac}|$ response.



(c) θ response.

Figure 15. V_{ac} mode test. (a) The test model, (b) magnitude response, and (c) angle response.

active power tracking while maintaining reactive power close to its reference. The DCvoltageModeTest showed the station response under DC voltage and reactive power control. The VacModeTest showed the ability of the station to regulate the magnitude and angle of the AC terminal voltage phasor.

The results indicate that the proposed library can be used as a reusable modeling framework for computationally efficient MMC-HVDC system studies in a Modelica-based environment. Since the library uses a phasor-domain arm average value representation, it is suitable for system-level dynamic studies where detailed switching behavior is not required. Future work will focus on extending the library with additional validation cases, point-to-point and multi-terminal HVDC network examples, and black-start oriented test systems. Further work will also include comparison against detailed EMT models, refinement of the master station mode-selection logic, and integration with larger OpenIPSL-based AC network models for studying HVDC interaction with realistic power systems.

The library is distributed with the BSD 3-Clause "New" or "Revised" License and is available online at: <https://github.com/ALSETLab/MMChvdcLibrary/>

Acknowledgements

This paper is based, in part, on work supported in whole or in part by the New York Independent System Operator (NYISO) Funded Project: “Dynamic Modeling of Voltage Source Converters for Black Start Capability Analysis” whose support is gratefully acknowledged.

References

- Chai, R. Z., B. H. Zhang, Z. G. Hao, T. Zheng, S. Y. Ma, and D. Z. Chen (2013). “Control Strategies of MMC-HVDC Connected to Passive Networks”. In: *Proceedings of the IEEE PES Asia-Pacific Power and Energy Engineering Conference (APPEEC)*, pp. 1–4. DOI: 10.1109/APPEEC.2013.6837129.
- Freytes, Julian, Lampros Papangelis, Hani Saad, Pierre Rault, Thierry Van Cutsem, and Xavier Guillaud (2016). “On the Modeling of MMC for Use in Large Scale Dynamic Simulations”. In: *Proceedings of the Power Systems Computation Conference (PSCC)*. Lausanne, Switzerland, pp. 1–7. DOI: 10.1109/PSCC.2016.7540919.
- Gonzalez-Torres, Juan Carlos, Rayane Mourouvin, Kosei Shinoda, Ahmed Zama, and Abdelkrim Benchaib (2021). “A Simplified Approach to Model Grid-Forming Controlled MMCs in Power System Stability Studies”. In: *Proceedings of the American Modelica Conference*. Online. DOI: 10.3384/ecp21181149.
- Jovcic, Dragan and Aliakbar Jamshidifar (2015). “Phasor Model of Modular Multilevel Converter with Circulating Current Suppression Control”. In: *IEEE Transactions on Power Delivery* 30.4, pp. 1889–1897. DOI: 10.1109/TPWRD.2014.2372780.
- Mukherjee, Biswarup, Marcelo de Castro Fernandes, and Luigi Vanfretti (2021). “A PMU-Based Control Scheme for Islanded Operation and Re-synchronization of DER”. In: *International Journal of Electrical Power & Energy Systems* 133, p. 107217. DOI: 10.1016/j.ijepes.2021.107217.
- Ortega, Álvaro and Federico Milano (2018). “Comparison of Different PLL Implementations for Frequency Estimation and Control”. In: *Proceedings of the 18th International Conference on Harmonics and Quality of Power (ICHQP)*, pp. 1–6. DOI: 10.1109/ICHQP.2018.8378840.
- Saad, Hani, Xavier Guillaud, Jean Mahseredjian, Sébastien Denetière, and Samuel Nguefeu (2015). “MMC Capacitor Voltage Decoupling and Balancing Controls”. In: *IEEE Transactions on Power Delivery* 30.2, pp. 704–712. DOI: 10.1109/TPWRD.2014.2338861.
- Samimi, Shabab, François Gruson, Philippe Delarue, Frederic Colas, Mohamed Moez Belhaouane, and Xavier Guillaud (2016). “MMC Stored Energy Participation to the DC Bus Voltage Control in an HVDC Link”. In: *IEEE Transactions on Power Delivery* 31.4, pp. 1710–1718. DOI: 10.1109/TPWRD.2016.2540926.
- Samimi, Shabab, François Gruson, Xavier Guillaud, and Philippe Delarue (2015). “Control of DC Bus Voltage with a Modular Multilevel Converter”. In: *Proceedings of the IEEE PowerTech Conference*. Eindhoven, The Netherlands, pp. 1–6. DOI: 10.1109/PTC.2015.7232513.
- Shinoda, Kosei, Abdelkrim Benchaib, Jing Dai, and Xavier Guillaud (2018). “Virtual Capacitor Control: Mitigation of DC Voltage Fluctuations in MMC-Based HVDC Systems”. In: *IEEE Transactions on Power Delivery* 33.1, pp. 455–465. DOI: 10.1109/TPWRD.2017.2723939.

Real-time Simulation and CHiL Testing of Power System Stabilizers on Microcontrollers with Modelica and eFMI

Luigi Vanfretti¹ Christoff Bürger² Joseph Pizzimenti¹ Kyle Wilt¹ Hao Chang¹

¹Electrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute, USA,
{vanfr1,pizzij,wilkt2,changh7}@rpi.edu

²Dassault Systèmes AB, Sweden, christoff.buerger@3ds.com

Abstract

Modern power grids face significant stability challenges, as shown by recent incidents such as the 2025 oscillations of the Iberian grid. These issues highlight the need for robust Model-Based Systems Engineering (MBSE) to enable rapid control redesign and deployment. Yet, the move to Controller Hardware-in-the-Loop (CHiL) testing is hindered by inefficient, manual re-implementation workflows, complex tooling and technology stacks with high software and hardware costs, and challenges to incorporate real-time suited plant models for controller testing. This lack of efficiency disrupts the traceability between design and implementation.

This paper pioneers an automated workflow using the Modelica language and the eFMI (Functional Mock-up Interface for embedded systems) standards to bridge this gap. We introduce the `OpenIPSL_CHiL` library, which provides extensions to the Open-Instance Power System Library (OpenIPSL) specifically for embedded real-time applications. Using Dymola's eFMI tooling, both the power system stabilizer (PSS) (a damping controller) and the generator interconnected to a power grid (the plant) are systematically synthesized into production-ready code and deployed onto low-cost ARM-based microcontrollers of the STM32 ecosystem.

The methodology is validated by a full suite of Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), and CHiL tests, ensuring that embedded behavior aligns with physics-based offline simulations, with the non-functional quality of generated production code ensured in the process by MISRA C:2023 and SEI CERT C Coding Standard compliance and real-time capability checks.

The developed framework offers an open-source, traceable, and cost-effective alternative to expensive proprietary real-time platforms and significantly reduces capital expenditures for grid control research.

Keywords: *power systems, power grid, power system stabilizer, real-time simulation, CHiL, embedded control, Modelica, eFMI, Dymola, STM32, ARM Cortex-M*

1 Introduction

1.1 Motivation

Challenges of power grid stability require better power system control: Modern power grids need a robust Model-Based Systems Engineering (MBSE) method to connect control design with real-world control deployment and routine tuning. Recent incidents, including the inter-area oscillations observed in Europe in 2016 and 2017 (ENTSO-E Sub-Group System Dynamics and Protection 2017; ENTSO-E 2018), highlight how crucial proper tuning of a Power System Stabilizer (PSS) is to preserve adequate damping throughout the system. More recently, the Iberian grid incident of 2025 (ENTSO-E 2025; ENTSO-E 2026) revealed complex 0.63 Hz and 0.2 Hz oscillations. Although initiated by inverter-based resources, the incident report emphasizes that poor coordination and the absence of detailed models for active damping controls—including existing PSS—limited operators' ability to foresee and contain the disturbance, forcing rapid revisions to operating rules.

Discrepancies between simulations and field behavior often stem from outdated modeling workflows that fail to capture actual hardware logic. Such control-induced issues have been increasingly observed in inverter-based resources, including irregular dynamics in wind plants due to setpoint truncation (Pudasaini et al. 2026) and non-sinusoidal responses in solar PV installations (Mishra, Vanfretti, Delaree, et al. 2024). This fragility extends to traditional synchronous generation; for example, a 1.5 Hz forced oscillation event was recently triggered during a reactive power capability test at a conventional power plant in Dominion (Mishra, Vanfretti, Baldwin, et al. 2024). Although the power plant model was accurate, the PSS was not designed for the specific operating condition encountered, leading to sustained oscillations. These challenges demand adaptable controllers that can be redesigned and deployed throughout their operational lifecycle (Bombois and Vanfretti 2024; Lauria et al. 2024).

Development and deployment challenges of power system controllers hinder adoption of novel stabilizers: When developing new control strategies to address grid stability issues, the path from new control designs to Con-

troller Hardware-in-the-Loop (CHiL) tests is currently inefficient due to a lack of dynamic model portability and broken traceability. Offline power system simulation tools, such as PSS®E or PSCAD®, cannot be directly used for real-time simulation. This necessitates manual re-implementation, which creates ambiguity and consumes significant time and resources. Furthermore, control code is often hardware-dependent; for example, specialized wide-area PSS implementations frequently require bespoke code for specific platforms such as National Instruments® hardware (Rebello et al. 2020). This fragmented workflow and the high cost of real-time simulators and prototyping hardware, such as CompactRIO® (National Instruments), create significant barriers to deploying novel control strategies.

eFMI tooling for Modelica provides new opportunities for power system design and control: The acausal, equation-based Modelica®¹ modeling language has been successfully used to design and simulate power systems and grids (Vanfretti, Rabuzin, et al. 2016; de Castro, Vanfretti, Gómez, et al. 2023). Its object-oriented, system-level design is particularly well-suited for rapid prototyping of design variants and modifications (Bürger 2019). Modelica also provides the means of modeling clocked subsystems, such as digital controllers, seamlessly integrated within the continuous simulation of plant models (Elmqvist et al. 2012). This makes Modelica an excellent choice for physics-based modeling and simulation of power systems with novel stabilizing controllers. Recent support for the new eFMI® Standard² (Modelica Association 2024) by Modelica tools (such as Dymola®³) extends the applicability of physics models beyond offline simulation to the embedded domain. The eFMI capabilities provided by Dymola facilitate the deployment of Modelica-based models in safety-critical applications that operate under stringent hard real-time constraints.

1.2 Contributions

This paper pioneers a Modelica and eFMI-based approach to design power system damping controllers. We use Modelica and eFMI to (1) rapidly prototype and conduct Model- and Software-in-the-Loop (MiL, SiL) testing of PSS designs, (2) deploy the designed models (plant and controller models) directly onto microcontrollers, and finally (3) verify in CHiL tests that embedded behavior aligns with the offline simulations. Our approach solves many of the previously bespoke design, development, and deployment challenges associated with a PSS:

- Common physics-based models are the single source of truth, reducing model-misalignment and inconsistency errors, and preventing behavior that violates or neglects laws of physics.
- The MBSE approach reduces errors during the tran-

sition from physics-based models to actual embedded solutions, as is typical for manually developed control code or code based on heterogeneous tooling at different abstraction levels that requires manual adaptation within deep technology stacks.

- Directly deploying models (controller and plant) on microcontrollers enables rapid control prototyping and reduces the risk of late detection of system-integration issues. Expensive real-time platforms can be avoided and the complexity of the tooling stacks involved can be reduced.
- Gaps in model portability and traceability are minimized.

To validate our approach, we designed real-time-suited Modelica models based on the Open-Instance Power System Library⁴ (OpenIPSL), generated the corresponding eFMI solutions in Dymola, system-integrated them on STM32® microcontrollers with the help of STM32CubeMX⁵ and STM32CubeIDE⁶, and systematically validated each step in respective MiL, SiL, and CHiL tests. All materials are provided open-source under the 3-Clause BSD License, as an extensible workbench framework⁷ ultimately empowering researchers to rapidly prototype grid controllers with guaranteed traceability and significantly reduced capital expenditure. Our approach provides a low-cost proof-of-concept platform for PSS microcontroller prototyping.

1.3 Paper structure

The remainder of this paper is organized as follows: Section 2 provides an overview of the eFMI Standard and the proposed MBSE workflow. Section 3 details the modeling approach and MiL validation using the new OpenIPSL_CHiL library. Section 4 discusses the synthesis of eFMUs and SiL tests and code analyses for such, while Section 5 describes the system integration on STM32 hardware. Real-time simulation results and CHiL performance analyses are presented in Section 6. Section 7 provides conclusions and outlines future work, followed by an Appendix summarizing the related work that highlights the limitations of current practices in the power industry related to damping controls.

2 eFMI & use-case workflow overview

This section provides a high-level overview of the eFMI Standard insofar as it is required to identify the main artifacts and generative model-based software engineering (MBSE) processes that it provides for our use-case; afterward, the objectives and workflow of the actual tooling we are using are presented.

¹<https://modelica.org/language/>

²<https://www.efmi-standard.org/>

³<https://www.3ds.com/products/catia/dymola>

⁴<https://github.com/OpenIPSL/OpenIPSL>

⁵<https://www.st.com/en/development-tools/stm32cubemx.html>

⁶<https://www.st.com/en/development-tools/stm32cubeide.html>

⁷<https://github.com/ALSETLab/efmi-pss-use-case>

2.1 The eFMI Standard

The eFMI Standard² (Modelica Association 2024) is an open standard of the Modelica Association^{®8}. The official website² provides a concise summary of the primary objectives of eFMI:

The *eFMI[®] Standard* is an open standard for the stepwise, model-transformation-based development of advanced control functions suited for safety-critical and real time targets. Its container architecture defines the common ground for collaboration among the stake-holders and toolchains along the various abstraction levels from high-level modeling and simulation – e.g., a-causal, equation-based physics in Modelica – down to actual embedded code. The model representations it defines are interlinked for traceability and semi-automatic code generation and do not only capture functional, but also non-functional quality requirements like coding standards, static memory allocation, worst time execution, and documentation, enabling credible development within a standardized workspace.⁹

The abstraction levels eFMI supports—each represented by a dedicated container type with well-defined content and with all containers of an embedded solution wrapped in a single embedded functional mock-up unit (eFMU)—are¹⁰ (note that, in contrast to FMUs, an eFMU is an open workspace container potentially still in development; eFMI can be used as a credible MBSE process to develop—i.e., implement—co-simulation FMUs for safety-critical real-time simulation):

Algorithm Code container: Algorithmic high-level solution of a sampled system. Algorithms are implemented in GALEC, a new guarded algorithmic language for embded control defined in the eFMI Standard. Although imperative, GALEC provides many high-level concepts like multi-dimensional arithmetic operators, derived but well-defined dimensional sizes, and a built-in function standard library to abstract target-specific characteristics and provide common math operations, including interpolation and solving systems of linear equations. GALEC is computationally safe; a powerful static evaluation concept is used to guarantee all indexing is within bounds, all loops have an upper-bound, and all potential runtime errors like IEEE 754 not-a-number (NaN) values, singular linear equation systems or domain errors of built-in functions (like poles of trigonometric functions) are handled or explicitly exposed to the runtime environment (error signals of interface functions). GALEC is also

well-defined regarding its imperative side-effects; for every two GALEC statements it is decidable whether they can be swapped, enabling, together with its well-defined dimensionality analysis, target-specific concurrent execution optimizations. Note that GALEC has no foreign function interface (FFI), only built-in functions; GALEC programs are therefore by nature self-contained. From a Modelica perspective, for example, a GALEC solution of a sampled system of equations (potentially with events or mixed system of equations) requires the solver to be inline-integrated as GALEC code, eliminating dependencies on third party numerical libraries.

These characteristics—self-contained, target independent, high abstraction level, and well-defined behavior under all circumstances—make GALEC an ideal intermediate language between simulation tools (that can target GALEC for code generation) and the embedded domain (where GALEC can serve as a source for code generation). Once a simulation tool finds a sampled solution for the simulation problem—which is not always possible for general equation systems—and expresses it as GALEC program, further tooling towards embedded C code "only" has to preserve these characteristics and can focus on code optimization (e.g., loop and multi-dimensional expression unrolling, or concurrent execution) and embedded target environment compliance (e.g., compliance with style guidelines, code analysers, interfaces, or other system integration requirements).

The sampled block-interface of the GALEC program of an Algorithm Code container is also exposed by the XML manifest file of the container, such that derived eFMU artifacts, like production code, can refer to it; this is important for traceability, but also to enable tooling to work with eFMUs without having to process GALEC (e.g., if just using production code without generating it). Naturally, each eFMU must have a single Algorithm Code container.

Production Code containers: Target-specific production code (C program) implementing the GALEC program of the eFMU's Algorithm Code container. Typical challenges of production code generators are related to efficiently translating GALEC multi-dimensional arithmetic operations, providing implementations of the used built-in functions (or preferable reuse target ecosystem routines considering restricted availability of the C Standard Library), complying with embedded coding standards like MISRA C:2023 or the SEI CERT C Coding Standard¹¹, following target ecosystem system integration APIs, generating ISO/IEC 9899 (C11, C17, C24, etc.) compliant code, and finding good trade-offs between portability and optimization.

An XML manifest file within each Production Code container maps the respective code's API to the well-defined (i.e., standardized in eFMI) block-interface of the eFMU's Algorithm Code container. This enables auto-

⁸<https://modelica.org/association/>

⁹Originally written by Christoff Bürger.

¹⁰Parts of the following Algorithm Code container section are from (Kamp et al. 2025) and have been originally written by Christoff Bürger.

¹¹<https://cmu-sei.github.io/secure-coding-standards/sei-cert-c-coding-standard/>

matic system integration—as for example exploited by Dymola’s production code export options presented in Section 4.4—although eFMI does not standardize any production code interface (unlike FMI¹² 3.0, which does with `fmi3Functions.h`). Naturally, an eFMU can contain many different production codes, e.g., for various floating-point precisions or code optimizations.

Binary Code containers: Target specific binary code generated from one of the eFMU’s Production Code containers. An XML manifest file lists the target platform, used compiler and linker options, and links back to the source Production Code container. Naturally, an eFMU can contain many different binaries, even for the same production code (e.g., based on different compiler and linker settings like debug and release versions, or for different targets).

Behavioral Model containers: Test scenarios with reference trajectories stored in comma separated value (CSV) files and additional meta-information—like trajectory units, the index-wise splicing of multi-dimensional trajectories into separate scalar trajectories, absolute and relative tolerances, or scenario descriptions—stored in an XML manifest file. Two kinds of test scenarios are supported: variable time grid scenarios and equidistant time grid scenarios. The latter enforce a fixed sampling period within strict time tolerances between adjacent trajectory points, as, for example, typical for HiL tests. Variable time grid scenarios are more flexible, but require well-defined interpolation (and for that reason typically higher tolerances) when used for testing a sampled system like GALEC programs or derived production code; they typically stem from continuous simulations with flexible step size (e.g., DASSL).

Naturally, an eFMU can contain many Behavioral Model containers, e.g., created by different simulation or measurement tools. Automatic instantiation—although the APIs of production and binary codes are not standardized—for testing is possible thanks to all eFMU containers (including Behavioral Model ones) linking back to the well-defined block-interface of an eFMU’s Algorithm Code container.

2.2 Use-case tooling & workflow

In our use-case, we develop two models with the final objective of embedded deployment for CHiL testing: a power system stabilizer (i.e., an actual controller) and a hard real-time simulation of the controlled power system (i.e., the plant model). To that end, we used the following tools:

- **Dymola^{®3} 2026x Refresh 1:** Physics-based design of Modelica models of the controller and plant; systematic MiL testing of design variants from continuous to sampled experiments with controller and plant models refined for eFMI (cf. Section 3); controller and

plant eFMU generation (cf. Section 4.1) and SiL tests of their production code against MiL experiments (cf. Section 4.2).

- **Cppcheck[®] Premium¹³ 26.3.0 and clang-tidy¹⁴ (Clang Tools 19.1.5):** Strict code quality analyses—in particular for MISRA C:2023 compliance—of the eFMU production code of controller and plant (cf. Section 4.3).
- **STM32CubeMX⁵ 6.17.0:** Selection of STM32 microcontrollers for controller and plant; configuration of their peripherals—like pins, DACs, and timers—and generation of respective template code linking to the STM32CubeH7¹⁵ 1.13.0 software package with hardware abstraction layer (HAL), lower-layer (LL) drivers, and Arm[®] Cortex[®]-M Common Microcontroller Software Interface Standard (CMSIS) implementations¹⁶ (cf. Section 5.1).
- **STM32CubeIDE⁶ 2.1.0:** Import of CubeMX generated template codes of controller and plant as individual C development projects; incorporation of their eFMU Production Code containers as symbolically linked source and include directories; system integration of eFMU production codes into the `main()` execution routine of the microcontrollers; compilation and deployment on STM32 boards (cf. Section 5.2).
- **WaveForms¹⁷ 3.25.1:** Interactive stimuli generation and plotting of CHiL test signals with a Digilent[®] Analog Discovery 3 oscilloscope (cf. Section 6).

Note, that our approach to deploy the plant model—which is required for CHiL testing the controller—directly on a microcontroller avoids the need for a real-time HiL platform, e.g., from dSPACE[®], Typhoon HiL[®] or OPAL-RT Technologies[®]; this reduces not only hardware costs but also the complexity of the required software and tooling stack and its licensing. All tools, except Dymola and Cppcheck Premium, are freely available. To generate eFMUs, the *Dymola Source Code Generation* option, besides a standard Dymola license, is required. Although an open-source version of Cppcheck exists (and could have been used), its MISRA C:2023 and SEI CERT C Coding Standard¹¹ coverage is significantly inferior to the commercial Premium version. And while production code analyses are optional for our use-case, they are of general interest for certification (Cppcheck Premium for example has TÜV SÜD^{®18} certification against EN 50128, EN 50657, EN 50716, IEC 61508, and ISO 26262).

¹³<https://www.cppcheck.com/>

¹⁴<https://clang.llvm.org/extra/clang-tidy/>

¹⁵<https://www.st.com/en/embedded-software/stm32cube7.html>

¹⁶STM32CubeH7 also includes middleware libraries supporting, for example, a USB stack, TCP/IP networking stack, file system, or even real-time operation system (RTOS); we do not use any of the STM32CubeH7 middleware libraries.

¹⁷<https://digilent.com/shop/waveforms/>

¹⁸<https://www.tuvsud.com/en>

¹²<https://fmi-standard.org/>

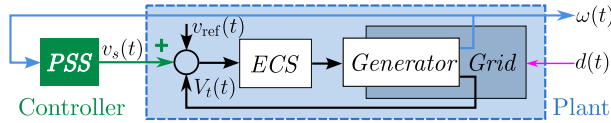


Figure 1. Diagram of a power plant's electrical generator equipped with a PSS interconnected with a power grid.

3 Modeling and MiL testing

This section describes the power system model that will be used to demonstrate the application of eFMI to both synthesis and deployment of the controller code. The system to be modeled is illustrated in Figure 1, which shows a power plant interconnected to a power grid. The power plant's generator, a synchronous machine, is equipped with a PSS that is placed before the Excitation Control System (ECS).

The main purpose of the ECS is to supply the exciter with sufficient excitation current and voltage, $v_f(t)$, to control the terminal voltage, $V_t(t)$, with respect to a reference value, $v_{ref}(t)$, forming what is typically called the “voltage control loop” of the plant. The PSS uses the generator's speed, $\omega(t)$, as a feedback signal and is placed in a cascade with the ECS. This increases the damping of the plant's response through the effect of the positive feedback signal, $v_s(t)$.

The goal of the PSS is to provide adequate damping and ensure the stability of the connection to the grid (in this case, a single power plant), or an entire interconnected region, when the grid is subject to disturbances, $d(t)$, such as faults, tripping of power lines, etc. However, to achieve this goal, these damping controllers must be properly tuned; if left without frequent tuning, they may lead to major grid stability problems, as described in the introduction. The PSS structure (a transfer function) is fixed, and the values of its parameters must be set through design studies before being implemented in the field.

Conventions: Referring to Figure 1, we will use the term **controller** to refer to the PSS, the term **generator unit** to refer to the generator and the ECS sub-system (with or without a PSS in some cases), and the term **plant** to refer to the sub-system comprising the generator, ECS, and remainder of the grid *without* the PSS.

3.1 OpenIPSL overview

To model the system shown in Figure 1, OpenIPSL⁴ 3.1.0 is used. OpenIPSL is an open-source Modelica¹ library of power system components for phasor-based modeling and simulation for dynamic analysis of power systems, known as “transient simulation” or “RMS simulation” in the power system literature. Although the original focus was on modeling high-voltage transmission networks using positive sequence phasors (Vanfretti, Rabuzin, et al. 2016), it is also capable of modeling distribution networks with unbalanced three-phase phasors (de Castro, Vanfretti, and Gómez 2019) without requiring the use of cosimula-

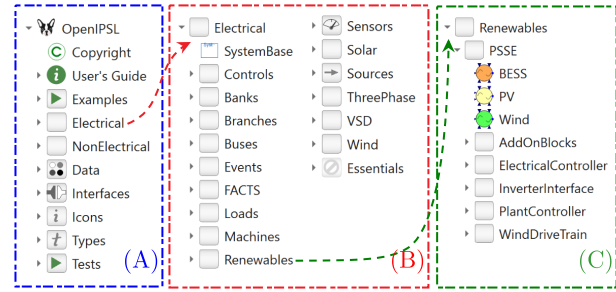


Figure 2. OpenIPSL 3.1.0 library overview.

tion, as is the case for domain-specific tools. In addition, it facilitates multi-time-scale modeling by integrating the electro-magnetic transients (EMTs) and transient simulation (TS) models (de Castro and Vanfretti 2025). The main structure of the OpenIPSL library is shown in Fig. 2 (A). The reader is referred to (Vanfretti, Rabuzin, et al. 2016; Baudette et al. 2018; de Castro, Vanfretti, Gómez, et al. 2023) for details about its sub-packages; the following are particularly relevant to this work:

- `*.Electrical` includes models of devices that make up an electrical power system. As shown in Figure 2 (B), it includes models of electrical machines, loads, transmission lines, transformers, and regulators/control devices. Each of the sub-packages in Fig. 2 has its own sub-packages (e.g., Fig. 2 (C)); for example, `*.Electrical.Renewables` includes other sub-packages that can be used to model photovoltaic (PV) solar, wind, and battery energy storage systems (Fachini et al. 2025).
- `*.NonElectrical` contains dedicated blocks, functions, and models that are utilized by the models in the `*.Electrical` package. It comprises extensions of MSL components as well as specialized elements required to ensure that the simulations are consistent with those from PSS[®]E (PSS[®]E 34.2.0 Model Library 2017; Laera et al. 2023).
- `*.Interfaces` is the sub-package containing connectors and interfaces from which every `*.Electrical` model is derived. It includes both connectors and a partial model for generators. The PwPin Modelica connector, together with its variants, specifies the *physical* phasor-based connector that forms the foundation of the library. The Generator partial model provides an interface for assembling detailed power generation units composed of internal components, e.g., the ECS and PSS.

Because the PSS is a controller that requires tuning, the OpenIPSL models used to construct the system shown in Figure 1 must be refactored for this purpose. For details on object-oriented modeling for control design with OpenIPSL—an approach applied to the models built for this work—the reader is referred to our work in (Vanfretti and Laughman 2024; Bhattacharjee et al. 2024).

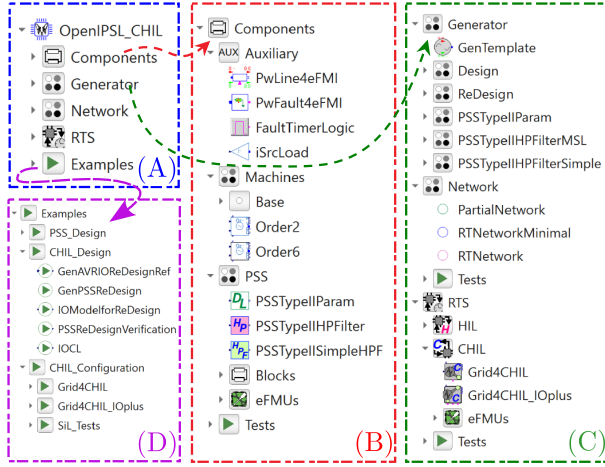


Figure 3. OpenIPSL_CHIL library overview.

3.2 OpenIPSL_CHIL library overview

For the purposes of this work the OpenIPSL_CHIL library, shown in Figure 3, was developed. As its name implies, the library provides extensions of OpenIPSL for embedded real-time applications. Figure 3 (A) shows the main structure of the library, which consists of:

- *.Components (Figure 3 (B)) is a sub-package featuring modified OpenIPSL components tailored for embedded deployment, alongside new components stemming from additional embedded domain requirements. Notably, the PSS sub-package includes variants of the **controller**, specialized blocks used for the variants, and the corresponding eFMU generation configurations for each PSS variant. In addition, models from OpenIPSL that required modifications for eFMU generation are within the **Auxiliary** and **Machines** sub-packages, which include modified models of the transmission line, fault, and machine components. The **Tests** sub-package provides unit tests for the components, including continuous, clocked, and eFMU variants of the PSS models.
- *.Generator (Figure 3 (C)) is a sub-package that contains multiple variants of the **generator unit** model. As the names of its sub-packages indicate, variants are built for different stages of the control design in the continuous domain, redesign, and to account for modifications required for the embedded hardware integration process.
- *.Network (Figure 3 (C)) provides variants of the power network without the **generator unit** that are used to build models for simulation and eFMU export, along with their corresponding unit tests.
- *.RTS (Figure 3 (C)) consists of **plant** models for deployment on real-time simulation-systems, both Hardware-in-the-Loop (HiL) and Controller-Hardware-in-the-Loop (CHiL). Only the CHiL **plant** models are used in this paper and their sub-package includes their eFMU generation configurations. A **Tests** sub-package provides unit tests for continuous, clocked and eFMU variants of these plant models.

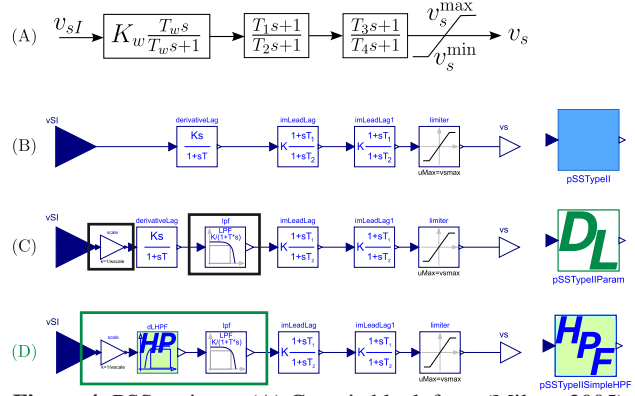


Figure 4. PSS variants: (A) Generic block from (Milano 2005), (B) OpenIPSL implementation, (C) Modified OpenIPSL implementation for eFMI export, (D) Final variant with a simplified high pass filter replacing the washout filter.

- *.Examples (Figure 3 (D)) contains whole system experiments with closed-loop **controller** and **plant** for offline simulations from continuous, toward sampled system and eventually expected final CHiL setup behavior (MiL and SiL).

In the following sections, we present the primary models in OpenIPSL_CHIL developed to illustrate how continuous-time representations can be converted into digital representations with Modelica synchronous and then be prepared for eFMU code generation.

3.3 Power system stabilizer (PSS) models

One of the simplest PSS models used in power system simulation software is shown in Figure 4 (A), and corresponds to the PSS Type II of PSAT (Milano 2005) and the STAB1 model in PSS[®]E (PSS[®]E 34.2.0 Model Library 2017). In this model v_{SI} is the input signal, i.e., ω from Figure 1, and v_s is the output signal. K_w is the stabilizer gain, which is multiplied by a washout filter with a T_w time-constant in seconds. The next two blocks represent two stages of lead-lag compensators that, along with K_w , must be designed by determining the time-constants T_1 - T_4 and the gain. Finally, a limiter sets the maximum and minimum output limits of the PSS, $v_s^{\max}$ and $v_s^{\min}$.

The implementation of this **controller** model is shown in Figure 4 (B) as found in OpenIPSL.Electrical.Controls.PSAT.PSS.PSSTypeII. However, the PSS model needs to be modified for export as eFMU and to consider embedded hardware aspects. First, the model is very sensitive to correct initialization; in particular, initial states depend on the initial value of the v_{SI} input, which likely is not just 0. To enable input dependent initialization, we add the top-level parameter

```
parameter OpenIPSL.Types.PerUnit vSI_start = 1.0;
```

which, in terms of eFMI GALEC, is a tunable parameter that can be recalibrated at initialization time, and modify the model to use

```
vSI(start = vSI_start, fixed = true)
```

Next, we need to consider that the code will run on embedded hardware that processes physical inputs, e.g., the speed signal. To address this, in Figure 4 (C), the extended and modified model has two additional blocks, enclosed by black squares. Observe that the input signal will come from the analog output of an embedded development board running the **plant** model (NUCLEO-H723ZG), which has a limited dynamic range and thus the signal is scaled up to provide sufficient resolution. Therefore, when read by the analog input of the embedded development board (NUCLEO-L476RG) serving as the **controller**, the signal will need to be scaled back to its nominal values when running on the PSS. This is achieved simply by applying a gain with value $1/w_{scale}$, where $w_{scale}=10$ is the multiplier applied to the output signal. Another hardware aspect that the models in Figure 4 (A)-(B) are not equipped to address is the noise that will be present in the analog signals. To this end, a low-pass filter with a cut-off frequency of 5 Hz is employed.

Finally, an additional variant is created, as shown in Figure 4 (D), which includes the previous modifications for the generation of eFMU code and hardware considerations (enclosed by a green square), and can be found under `*.Components.PSS.PSSTypeIISimpleHPF`. Comparing Figures 4 (C) and (D), it is possible to observe that the `derivativeLag` block of OpenIPSL, which models a washout filter, has been replaced by a simple high-pass filter. Observe that the goal of a washout filter is to approximate the response of (originally) an analog circuit whose role is to remove the effect of very low frequencies that arise due to the normal operation of a grid by attenuating these low frequencies in the frequency domain in the same way that an ideal derivative would. Such analog circuits were easy to implement with an operational amplifier circuit with a resistor and a capacitor, from which T_w would be determined by the RC time-constant. However, since our goal is to implement the PSS in an embedded controller, we can adopt a simpler solution to provide the same functionality, i.e., to have the same frequency response of an ideal derivative. This can be achieved using a first order Butterworth high-pass filter, as shown in Listing 1.

The implementation in Listing 1 preserves the same parametrization as the other variants, avoids differentiation (and thus potential numerical noise amplification), and is simpler as it only contains one state variable, one differential equation, and one algebraic output equation.

This is the final variant that we will use to illustrate eFMI export and integration into embedded hardware as the **controller**.

Listing 1. HighPassSimple component used in the PSS of Figure 4 (D).

```
1 block HighPassSimple
2 ...
3 parameter Real K = Kw*Tw "Gain K = Kw*Tw";
4 parameter Real T = Tw "Time constant T = Tw";
5 ...
6 initial equation
```

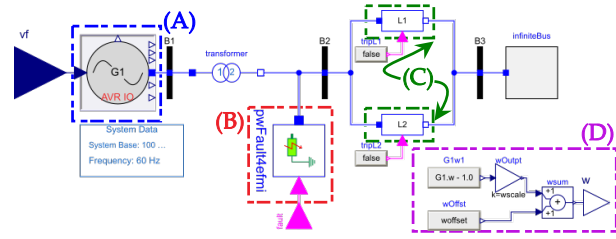


Figure 5. The **plant** model: Generator with ECS interconnected with a power grid—`*.RTS.CHIL.Grid4CHIL`, referred to as GCHIL.

```
7 der(x) = 0;
8 equation
9   der(x) = (u - x)/T;
10  y = K/T*(u - x);
11 end HighPassSimple;
```

3.4 Power system models

Returning to Figure 1, we now develop the models for the **plant**, which is composed of a generator with its ECS and the remainder of the grid. Although multiple **plant** variants of the model were built, only the model used for eFMU export and final system integration is discussed here, which is shown in Figure 5. The components enclosed with a hatched square labeled (A), (B), and (C) required modifications for eFMU GALEC code generation, while (D) is used to scale the analog output signal. All other components are taken from OpenIPSL and used unchanged.

Let us first discuss the **generator unit**, shown in Figure 5 (A). This is a sub-system shown expanded in Figure 6; it includes both the ECS (see Figure 6 (A)) and the synchronous machine (see Figure 6 (B)) models. The ECS is modeled using the AVR Type III model from (Milano 2005) implemented in OpenIPSL, which does not require any modifications. Meanwhile, the synchronous machine model “Order VI” from (Milano 2005), as implemented in OpenIPSL.Electrical.Machines.PSAT.BaseClasses.baseMachine, required modifications to the base class, which are shown in Listing 2.

Listing 2. `baseMachine` modifications for eFMU GALEC code generation.

```
1 // Original OpenIPSL baseMachine:
2 partial model baseMachine
3 ...
4 parameter Types.Angle delta0 = CM.arg(
5   Vt0 + ((ra + CM.j*xq0) * Z_MBtoSB*I0))
6   "Init. val. rotor angle";
7 ...
8 end baseMachine;
```

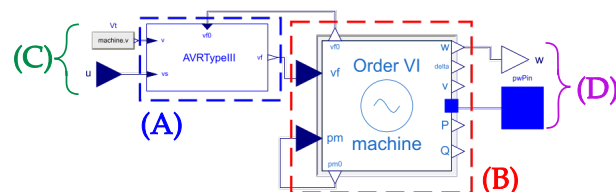


Figure 6. The **generator unit** model: Generator with ECS—`*.Generator.ReDesign.GenAVRIOReDesign`.

```

9
10 // Modified OpenIPSL_CHIL baseMachine:
11 partial model baseMachine
12 ...
13 parameter Complex auxCMval =
14   Vt0 + ((ra + CM.j*xq0) * Z_MBtoSB*I0)
15   "Auxiliary variable to calc. delta0";
16 parameter OpenIPSL.Types.Angle delta0 = atan2(
17   auxCMval.im, auxCMval.re)
18   "Init. val. of mach. rotor angle";
19 ...
20 end baseMachine;

```

Comparing lines 4-6 and 13-18, it can be observed that the initial value for the machine angle, originally calculated with the function `CM.arg` (an alias of `Modelica.ComplexMath.arg` that passes the real and imaginary parts of its complex argument to `Modelica.Math.atan3` together with a parameterized phase angle `phi`) was replaced by defining an auxiliary parameter `auxCMval` and then computing the angle using `atan2`. This is necessary because eFMI GALEC only provides `atan2` as a built-in function, but not `atan3`. Interestingly, the phase angle `phi` is always 0 in `OpenIPSL`, in which case `atan3(c.im, c.re, phi) == atan2(c.im, c.re)` making the `atan2` replacement of `CM.arg` mathematically equivalent.

For completeness, refer to Figure 5 (C) to note that the input to the ECS in Figure 1 corresponds to u and the feedback signal, V_r , to v . Meanwhile, in Figure 5 (D) it can be seen that the output w corresponds to signal $\omega(t)$ in Figure 1, while `pwPin` is the main connector of `OpenIPSL`.

Next, let us discuss the components labeled (B) and (C) in Figure 5, which are part of the `plant` model. (B) is a fault component (`pwFault4efmi`), while (C) comprises instances of a line model (`L1` and `L2`). These models are a modified version of the `OpenIPSL` models `*.Events.PwFault` and `*.Branches.PwLine` from the electrical sub-package. In `OpenIPSL`, these components use the Modelica built-in variable `time`, which is used to activate/deactivate model features at specific points in time, thereby allowing the user to create simulation scenarios. In the employed CHiL test configuration, no single “master clock” is available that could provide a common absolute time reference for both `controller` and `plant` (in fact, a dependency of embedded code on absolute time is typically a strong error indication). Furthermore, the fault event should, ideally, be initiated externally by the user so that the `controller` response can be evaluated in a semi-interactive manner. Hence, two new models in `OpenIPSL_CHIL.Components.Auxiliary`, `*.PwFault4eFMI` and `*.PwLine4eFMI` replace the implicit `time`-based behavior switch with explicit Boolean inputs. The interface of the new fault block therefore has a Boolean to trigger the fault externally, while in the case of `L1` and `L2` this option is disabled by setting the corresponding Boolean input to `false`.

Finally, the components labeled (D) in Figure 5 represent the final signal conditioning stage of the `plant` model. Recall that the `plant` system will run on a NUCLEO-

H723ZG embedded development board. This hardware utilizes a 12-bit Digital-to-Analog Converter (DAC) operating within the 0–3.3 V range, resulting in a fixed hardware resolution of ≈ 0.0008 V/lb. The w output of the generator unit is the speed in per unit (pu) and its nominal value is 1.0 pu in steady state. In a simplistic implementation, the value for w may be directly applied to the DAC; consequently, 1.0 pu would result in 1.0 V and the effective resolution of w would be limited to ≈ 0.0008 pu. During disturbances, speed deviations are typically small, often not exceeding the third decimal place (e.g., a shift from 1.0 to 1.006 pu). Because this maximum expected deviation is so close to the board’s fixed quantization floor, a raw signal would lack the fidelity required for accurate control and would suffer from significant quantization error. To mitigate this, the following signal conditioning steps are implemented: (i) *Offset Removal*: The 1.0 pu offset is removed from `G1.w` to isolate the speed deviation, Δw . (ii) *Signal Scaling*: The resulting deviation is scaled by a factor of `wscale = 10`. This process “stretches” the small transient response across a larger portion of the board’s digital dynamic range, effectively increasing the Signal-to-Quantization-Noise Ratio. (iii) *Reconstruction Offset*: The `woffset` is set to 1.0. This “lifts” the processed signal into the center of the DAC’s operating window, providing sufficient margins for the signal to vary bidirectionally, preventing signal saturation or clipping when close to the 0 V or 3.3 V rails during severe transients.

3.5 MiL tests

To validate the models described in the previous section, we built typical offline system simulation experiments and experiments that mimic the CHiL setup. This allows us to study the expected behavior of the entire system in Figure 1, using multiple model variants: from a typical offline system representation, as in Figure 7, to more granular representations, as shown in Figure 8 (A)–(D), which aim to reproduce the physical CHiL setup. All these model-in-the-loop (MiL) experiments can be found under `OpenIPSL_CHIL.Examples.CHIL_Configuration.Grid4CHIL`.

In Figure 7, a representation of how the system would present itself in the “real-world” is shown. The figure shows within the hatched polygons that (A) and (C) represent the system model, and (A) is expanded in (B). In (B) it can be seen that the PSS `controller` model that we want to deploy on embedded hardware has been represented with the purpose-built variant in Figure 4(D). Moreover, because we aim to model an embedded system, a clocked partition running at 0.2 ms with an explicit Euler solver is created by adding a `periodicClock` and `sampler` to the input signal w and a `hold` to the output of the `pss` model. Observe that the gain `wscale` is applied before connecting the sampled signal to the PSS, this is because the input signal w comes directly from the `generator unit` (expanded in Figure 6) and *not* the pre-conditioned

signal w in Figure 5 (D) that is used to represent the **plant** in CHiL. This architecture was used in this first model as it aims to represent the deployment of the digital **controller** in a “real-world” power plant, and *not* in CHiL testing. As can be observed in Figure 7 when comparing it with Figure 5, all models implementing the remainder of the power system are represented with unmodified components from OpenIPSL, which use continuous-time representations.

Next, to actually model the CHiL setup, the model variants in Figure 8 are created. In (A) a continuous-time variant of our CHiL setup is created, where the **pss** (**controller**) output is connected to the input of G4CHIL (**plant**), and the output of G4CHIL is connected to the input of the **pss**¹⁹. Note that G4CHIL includes a **faultTimer**, which allows us to model the pressing of a push button that applies the fault within the model for $t = [1.5-1.55]$ s.

To transform the continuous-time model in Figure 8 (A) into a digital control representation, both the **controller** and **plant** are modeled within their own clocked partitions, as shown in Figure 8 (B). This is achieved similarly to what was done in Figure 7, i.e., by including a sampler, hold, and periodic clock to each partition running at 0.2 ms.

At this point, it is informative to analyze the behavior of the CHiL setup representations in Figures 7 and 8 (A)-(B). To this end, we create the following simulation cases (tolerances are set to 0.0001 for all cases):

1. Ref.Sim.DASSL: Model in Figure 7 simulated with DASSL and Interval=0.0002.
2. Ref.Sim.ExpEuler: Model in Figure 7 simulated with Explicit Euler fixedstepsize=0.0002.
3. Ref.Setup: Model in Figure 8 (A) with DASSL and Interval = 0.0002.
4. Ref.Setup.ExpEuler: Model in Figure 8 (A) with Explicit Euler fixedstepsize=0.0002.

¹⁹Note: In the physical CHiL setup (see Figure 16), the embedded boards are connected through patch wires on a breadboard. Breadboard capacitance, and signal integrity and delay due to the patch cables are irrelevant for this application and do not need to be modeled.

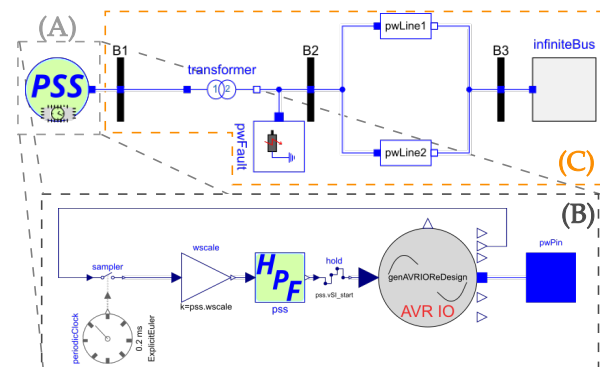


Figure 7. Reference simulation model: (A) Power system model instantiation of the generator unit with PSS, (B) Generator unit model with clocked PSS, (C) Power system being interfaced to the generator unit with PSS.

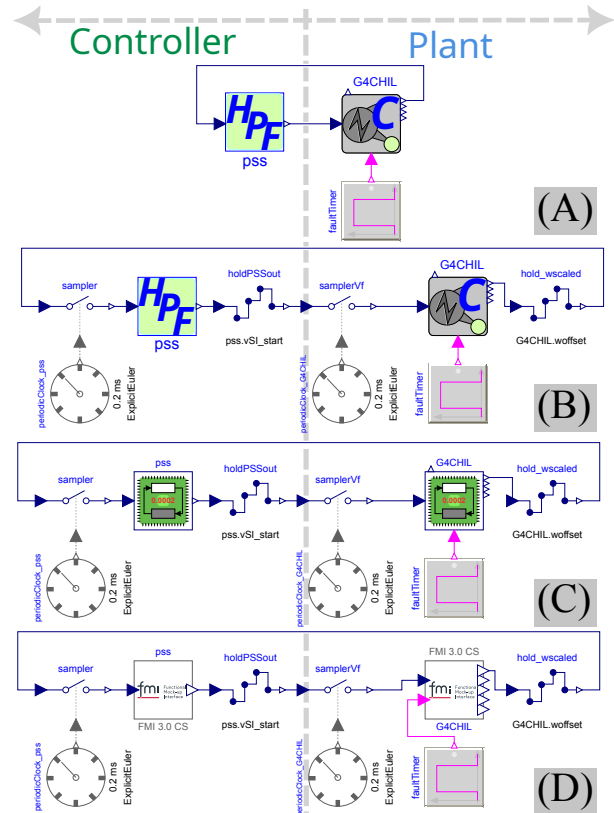


Figure 8. Simulation models of CHiL setup: (A) Continuous-time representation, (B) Clocked representation, (C) Clocked with eFMUs, (D) Clocked with source code FMUs based on generated eFMI production code.

5. Ref.Setup.Clocked: Model in Figure 8 (B) with DASSL and Interval = 0.0002.

The simulation results are summarized in Figure 9 and Table 1. As shown in Figure 9 (A), the cases are visually indistinguishable. The error statistics in Table 1 reveal that the transition to the fixed-step Explicit Euler solver (Sim 2) introduces a minimal baseline penalty (3.0160×10^{-5} pu), while the structural refactorings required for eFMI compatibility account for the additional deviation in Sim 4. High transient fidelity is maintained under fault stress; peak errors occur during the post-fault recovery phase (near $t = 2.05$ s) rather than during the fault interval itself, confirming the numerical stability of the eFMI-targeted model variants. Furthermore, the RMS errors are consistently an order of magnitude lower than the maximum deviations, indicating that these peaks represent brief numerical transients rather than persistent systemic biases. Crucially, as the maximum observed deviation (1.6949×10^{-4} pu) remains below the ≈ 0.0008 pu hardware quantization floor of the target STM32’s 12-bit DAC, these numerical differences are masked by the hardware’s resolution limits and have no impact on the physical control performance.

The subsequent step to further refine the level of detail in our representation of the CHiL configuration is to generate eFMUs for *both* the **controller** and the **plant**, and then employ these in place of the **pss** and **G4CHIL** mod-

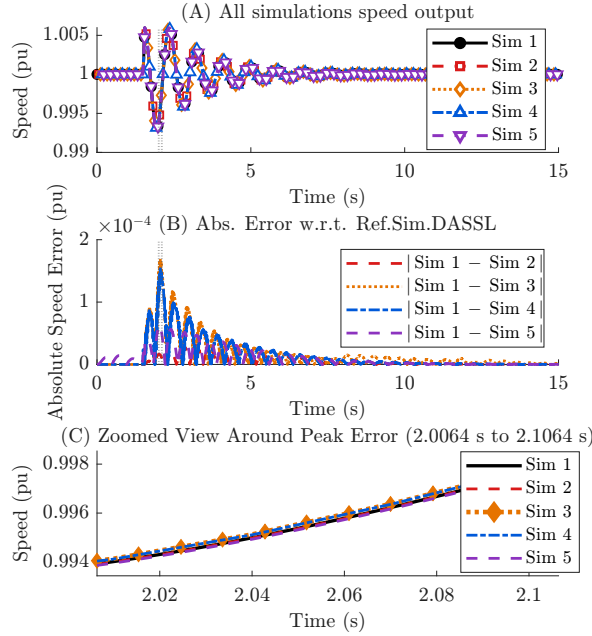


Figure 9. MiL Simulations: (A) w for all simulations (see Figure 6 (D)), (B) $\forall i \in \{2, 3, 4, 5\} : |\text{Ref. Sim. DASSL} - \text{Case } i|$, (C) Zoomed view around peak error.

Table 1. Generator speed error relative to Ref. Sim. DASSL.

Case	$\max \varepsilon $ (pu)	RMS ε (pu)	$t_{\max}$ (s)
2	3.0160×10^{-5}	9.2302×10^{-6}	3.2156
3	1.6949×10^{-4}	3.0662×10^{-5}	2.0564
4	1.5353×10^{-4}	2.7456×10^{-5}	2.0564
5	6.1989×10^{-5}	1.6466×10^{-5}	2.0310

els as illustrated in Figures 8(C) and (D) to conduct SiL tests of the generated eFMI production code.

4 Generation & SiL testing of eFMUs

To generate eFMUs for the PSS and power system variants that have been designed and MiL-tested in Section 3, Dymola offers the `DymolaEmbedded` library. It features a convenient user interface—graphical in the diagram and parameter editor views, yet also fully scriptable in `*.mos` files like any other Modelica library in Dymola—for configuring eFMU generation and conducting SiL tests of the resulting ISO/IEC 9899 (C11, C17, or C24 compliant) production code.

The production code of exported eFMUs leverages on the GALEC guarantees sketched in Section 2.1; in particular, Dymola’s whole range of symbolic equation system optimizations are automatically conducted to find an algorithmic, inline-integrated eFMI GALEC solution for models subject to eFMU export.

4.1 eFMU configurations

To configure eFMU export for a Modelica model, a new package extending `DymolaEmbedded.EmbeddedConfiguration` has to be created. Each such extending-package represents an eFMU configuration and inherits configuration

options and generative facilities as shown in Figure 10. The available configuration options are as follows:

- *eFMU bundle configuration*: Optional name of the directory that shall contain the generated eFMU. If not explicitly set, the full qualified package name of the eFMU configuration is used.
- *Model configuration*: Model subject to eFMU export (mandatory).
- *Integrator configuration*: Fixed sampling period (mandatory) and optional solver method for inline-integration. The only mature solver currently supported is the explicit Euler method; implicit Euler and Rosenbrock methods (Olsson et al. 2017) are still at the prototype stage for eFMU export.
- *Code configuration*: Optional options to obfuscate generated GALEC code, constant-fold all parameters to reduce code size (parameters are hard-coded in the generated code rather than exposed through the GALEC block interface; dependent parameters are removed, preventing any later recalibration), and keep multi-dimensional entities intact rather than reducing them to scalars (e.g., used to preserve tensor-flows of embedded neural networks modeled using the `NeuralNetworks` package of the eFMI library, cf. (Kamp et al. 2025)).
- *Software Production Engineering configurations*: Pre-configured production code generator settings. Defines the floating-point precisions for which production codes shall be generated (by default for 32-bit and 64-bit)—these are the eFMU Production Code containers to generate. Also defines for which instruction set architectures (by default for x86, Microsoft® Windows® 32- and 64-bit) statically linked libraries shall be generated for each Production Code container—these are the eFMU Binary Code containers used for SiL simulation within Dymola (cf.

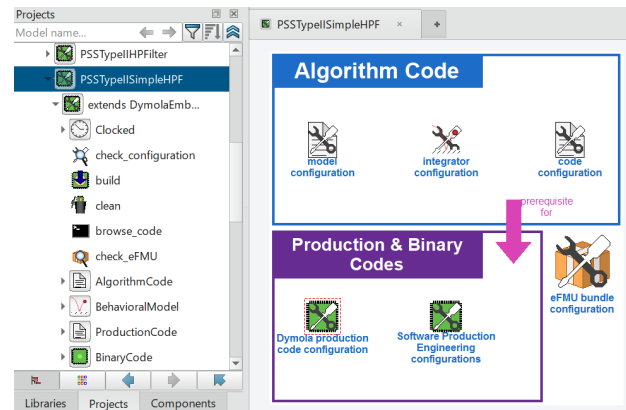


Figure 10. DymolaEmbedded eFMU configuration for the PSS controller of Figure 4 (D). Double-clicking on diagram-view components opens the respective configuration. Calling functions of the `extends`-entry in the package browser executes the respective action, e.g., building GALEC or production code or analyzing code quality with clang-tidy or Cppcheck Premium.

Section 4.2).

- *Dymola production code configuration*: Deprecated embedded code export, predating Dymola's eFMI support.

For the PSS and power system variants designed in Sections 3.3 and 3.4, an explicit Euler with a 0.2 ms sampling period is set as the integrator configuration. In addition, the **controller** of Figure 4 (D) and the **plant** of Figure 5 are chosen as model configurations, respectively. For both, code obfuscation is deactivated.

The generative facilities (i.e., inherited functions) provided by eFMU configurations are:

- `build()`: Generate eFMU with Algorithm, Production and Binary Code containers. Optionally also export SiL-stubs (cf. Section 4.2) and source code FMUs for each generated Production Code container (cf. Section 4.4). The root-level `build()` function exists for convenience; each eFMU container can be individually generated within the `build()` functions of the respective sub-packages (e.g., `ProductionCode.build()`).
- `browse_code()`, `clean()`: Open the eFMU in the default file manager or shell (e.g., File Explorer and `cmd` in Microsoft Windows); delete all generated artifacts.
- `ProductionCode.check_codes()`: Analyze Production Code containers with clang-tidy and Cppcheck Premium (cf. Section 4.3) for code quality and MISRA C:2023 compliance.
- `build_FMUs()`, `build_Simulink_imports()`, and `build_Arduino_sketches()` of `ProductionCode`: Export Production Code containers as source code co-simulation FMUs, Simulink® C Function blocks or Arduino® sketches (cf. Section 4.4), respectively.
- `BehavioralModel.build_tests()`: Reuse existing Modelica experiments to derive SiL-tests for the production codes of the eFMU (cf. Section 4.2).

By default (i.e., if arguments are not interactively changed), the above generative eFMU facilities work with the options of their respective eFMU configuration (i.e., the `DymolaEmbedded.EmbeddedConfiguration` modifiers of their call context). Hence, it is not necessary to reconfigure eFMU export repeatedly when switching models or libraries; instead, configurations are stored together with their corresponding eFMU configuration package.

Note that being ordinary Modelica packages, eFMU configurations (their options and functions) can be scripted in Modelica *.mos scripts. This enables convenient configuration of variants and batch processing. Likewise, an *eFMI*-button in Dymola's *Tools*-ribbon (see Figure 11) enables the user to build all eFMUs of a whole package hierarchy, or load the SiL-stubs of existing eFMUs.

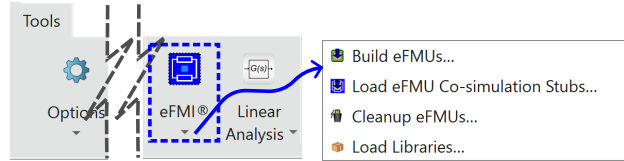


Figure 11. *eFMI*-button in Dymola's *Tools*-ribbon.

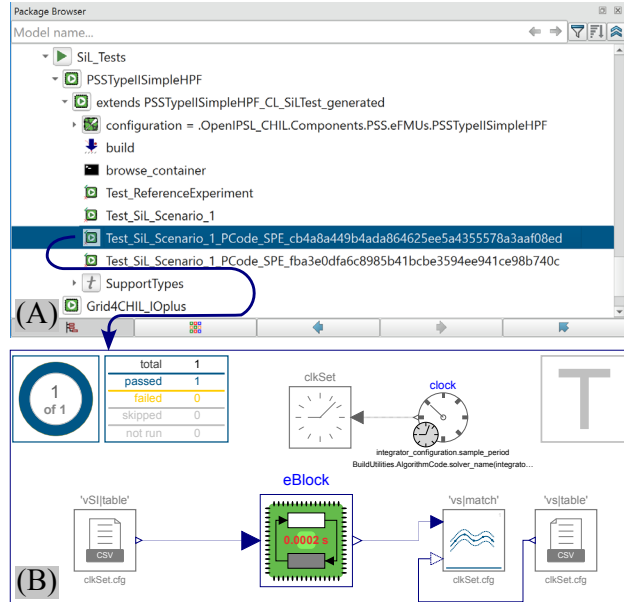


Figure 12. (A) SiL experiment package derived from the MiL experiment of Figure 7, (B) SiL test model for the 32-bit floating-point precision production code generated for the PSS **controller** of Figure 4 (D).

4.2 Modelica-based SiL tests

SiL experiment packages: To validate the functional correctness of the production codes generated for **controller** and **plant** models, we reused the MiL experiments of Section 3.5 as a basis to define respective SiL tests. To that end, Dymola's eFMI facilities support the generation of SiL experiment packages backed by an eFMI Behavioral Model container derived from an existing Modelica experiment; the `BehavioralModel.build_tests()` function of an eFMU configuration (cf. Section 4.1) can be used to generate such SiL experiment packages. Given any Modelica experiment model, `build_tests()` extracts—based on the *model configuration* of the eFMU configuration in whose context it is executed—all instances of the model subject to eFMU code generation. Each instance defines a test scenario.

Figure 12 (A) shows the SiL experiment package (SiLTest) generated from the MiL experiment of Figure 7 for the **controller** of Figure 4 (D). The `Test_ReferenceExperiment` model exploits the MiL experiment of Figure 7 to record reference trajectories. The `Test_SiL_` models are for testing the various production codes of the eFMU. The `build()` function simulates `Test_ReferenceExperiment` and generates the Behavioral Model container backing the SiL ex-

periment package, in particular its comma separated value (CSV) files with the reference trajectories and a manifest file with meta-information about the test scenarios, their in- and outputs, their respective units, etc. The `browse_container()` function just opens the backing Behavioral Model container in the default file manager (e.g., File Explorer in Microsoft Windows).

SiL test models: Figure 12 (B) shows the SiL test model for the 32-bit floating-point precision production code for the **controller** of Figure 4 (D). The `eBlock` component is a SiL-stub generated for the production code. Such stubs are Modelica models with the same interface as the source model of an eFMU configuration, but their implementation is done by referencing the eFMU's production code by means of Modelica's external C code interface (the concept is similar to the Modelica stubs Dymola generates for imported FMUs). The `'vsI|table'` and `'vs|table'` components are input tables providing the reference trajectories for the PSS. The `'vs|match'` block simply compares the actual `vs` output of the production code (`eBlock.vs`) with the expected reference trajectory considering user-defined absolute and relative tolerances. The used tolerances can be defined for all scenarios, or specific ones, in the root diagram view of generated SiL experiment packages; they are the only additional user input required. Otherwise, the whole SiL experiment package can be generated from the MiL experiment of Figure 7.

All components, except the `eBlock`, are from the `Testing`²⁰ Modelica library, freely available with Dymola. `Testing` is a professional library to exploit Modelica experiments for testing purposes, automate regression tests (e.g., with Jenkins²¹), and provide convenient test results and plots. It is used for the continuous integration tests of the commercial Modelica library portfolio of Dassault Systèmes.

SiL test results: The results of the SiL tests for our eFMUs are satisfactory. For the **controller** of Figure 4 (D), the `vs` output of the 32-bit floating-point precision production code has a maximum deviation of ≈ 0.000365324 for the test scenario of Figure 7 (note that the reference trajectories are recorded based on 64-bit floating-point precision simulations); the 64-bit production code has a maximum deviation of $\approx 1.38778e - 16$. The `w` output of the **plant** of Figure 5 has deviations as follows: ≈ 0.000161166 for 32-bit floating-point precision production code and $\approx -1.77636e - 14$ for 64-bit.

Note that we will system integrate the 32-bit production codes in Section 5 since the **controller** STM32 board (NUCLEO-L476RG) supports hardware acceleration only for 32-bit floating-point precision and 64-bit floating-point precision production code for the **plant** turned out to execute significantly slower on its STM32 board (NUCLEO-H723ZG) than the 32-bit floating-point precision code,

leaving to little headroom with a very high risk of overruns in case of merely minor future model changes (cf. Section 6.2 for details).

4.3 Production code quality analyses

To validate that eFMU production code generated by Dymola is of very high quality, Dymola provides preconfigured code quality checks with `clang-tidy`¹⁴ and `Cppcheck Premium`¹³; the latter is particularly useful to validate MISRA C:2023 compliance. The preconfigurations are as verbose, complete and strict as possible.

Clang-tidy configuration: For `clang-tidy` all checks are enabled except the following checks: (1) *readability-identifier-length* (disabled because names of source Modelica models shall be reused, but in physics often very short variable names are used like `v` for voltage or `w` for rotational speed), (2) *cppcoreguidelines-avoid-magic-numbers* and *readability-magic-numbers* (physics equations can contain magic numbers), (3) *altera-unroll-loops* and *altera-id-dependent-backward-branch* (loop unrolling shall be done by the compiler, not the user; the implementation of multi-dimensional GALEC functions in generated production code is universal—their loops depend on the passed size arguments but these are literals at the call site, which is why we can assume compilers are capable of automatic optimization), and (4) *altera-struct-pack-align* (code shall be portable and alignment automatically handled by the compiler).

Cppcheck Premium configuration: For `Cppcheck Premium` the check-level is exhaustive, including style and information analyses; even inconclusive analyses are reported. Further MISRA C:2023 and SEI CERT C Coding Standard¹¹ checks are enabled. For the implementation of GALEC built-in functions, a dedicated library configuration is provided to help `Cppcheck` identify built-in function API violations (e.g., if passed pointers to multidimensional variables must already be allocated and/or initialized).

Code analyses results: The results of the production code analyses for our examples can be found in the public repository⁷; for each production code container in the `logs/check-code/clang-tidy` and `Cppcheck` subdirectories respectively. In the following discussion, we consider the 32-bit floating-point precision code for the **controller** of Figure 4 (D).

First of all, `clang-tidy` (Clang Tools 19.1.5) reports that 64-bit floating-point precision literals are assigned to 32-bit variables (*bugprone-narrowing-conversions* and *cppcoreguidelines-narrowing-conversions* violations); this is a shortcoming of Dymola's production code generator, which fails to add proper `F` suffixes, like in `1.0F / instance->wscale`. Note however, that the actual literals stem from the source model where they are in 64-bit precision. The narrowing when generating 32-bit floating-point precision production code is intended, but we fail to make it explicit to avoid the reports. Further,

²⁰<https://www.youtube.com/watch?v=wnk5YbEgTjM>

²¹<https://www.jenkins.io/>

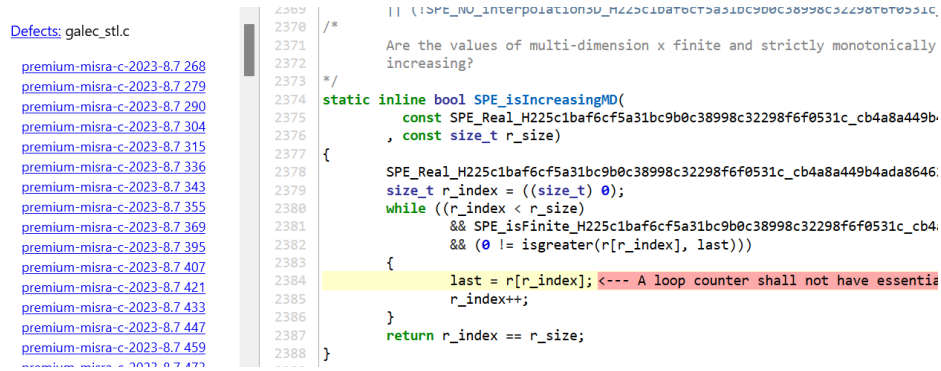


Figure 13. Excerpt of the Cppcheck Premium report for the 32-bit floating-point precision production code generated for the PSS **controller** of Figure 4 (D).

clang-tidy reports that some local variables are not initialized (*cppcoreguidelines-init-variables* violations), which, after manual inspection, are clearly false-positive reports. There are no further reports by clang-tidy.

The Cppcheck Premium 26.3.0 reports are much more interesting. They are syntax highlighted using Python Pygments²², as shown in Figure 13. There are 204 reported issues, most are systematic false-positives considering our application. There are 159 violations of MISRA C:2023 Rule 8.7 (advisory), *"Functions and objects should not be defined with external linkage if they are referenced in only one translation unit"*; all are about GALEC built-in functions, whose implementation is intended as shared library—hence, can be ignored. There are 23 violations of SEI CERT C DCL06-C, *"Use meaningful symbolic constants to represent literal values"*; they stem from magic-numbers in physics equations—hence, can be ignored too. There are 3 violations of SEI CERT C INT30-C, *"Ensure that unsigned integer operations do not wrap"*, which we know to be false-positives because the respective `index - 1` expressions can become at most 0 due to indexing starting at 1 in eFMI GALEC. All other reported issues fall into one of these categories, like a `typedef` or objects intended for library usage (MISRA C:2023 Rule 2.3 & 8.9 (advisory) violations). The only interesting MISRA C:2023 violation reported is for Rule 14.1 (required), *"A loop counter shall not have essentially floating type"*, which has been reported for a function deciding if an array only contains strictly increasing values (used to define an interpolation abscissa); this report is a false-positive the Cppcheck team meanwhile fixed in Cppcheck Premium 26.6.0 based on our findings.

4.4 Production code export

In general, Dymola-generated production code can be integrated into an embedded system with little effort, as will be illustrated for the STM32 ecosystem in Section 5.2. Nevertheless, Dymola supports dedicated production code export for popular embedded ecosystems. Export/import thereby means to use the C code of an eFMU Production Code container as is, add any required wrapper code and

meta-information to satisfy target ecosystem API requirements, and import the code such that the further (in particular compilation) infrastructure of the target ecosystem can be used for system integration on actual hardware. Dymola supports export of production code for (the respective eFMU configuration facilities are listed in Section 4.1):

- **FMI®**: Export as FMI¹² 3.0 source code co-simulation FMU with `fixedInternalStepSize`. The only code added to exported production code, besides the mandatory `modelDescription.xml`, is the required implementation of `fmi3Functions.h`, which is lean, avoiding as much overhead of the generic FMI interface as possible while at the same time introducing sanity checks to ensure the FMU is used as intended (in particular called with its fixed sampling period or a proper multiple thereof). eFMU production code exported as source code FMU enables the use of eFMI on many dedicated real-time platforms with source code FMU import; the tooling of such platforms will take care of compiling and deploying the production code. Examples are ConfigurationDesk® (dSPACE), TwinCAT® (Beckhoff®), or Silver® (Synopsys®).

It is noteworthy that the eFMU production code based FMUs exported by Dymola are significantly smaller than FMUs exported with its normal source and binary code co-simulation FMU export. For the **controller** of Figure 4 (D) the eFMU production code backed source code FMU is ≈31 kB, its compiled binary code version for Microsoft Windows 11 64-bit on Intel® x86-64 is ≈176 kB, whereas the model directly exported as source code FMU is ≈5.41 MB and binary code FMU is ≈2.84 MB (in both cases exported with protected variables excluded, only black box, FMI 3.0 co-simulation, and inline integration method explicit Euler). Likewise for the **plant** of Figure 5: ≈38 kB for the eFMU backed source code FMU, ≈220 kB for the eFMU backed binary code FMU, ≈5.49 MB for the normal source code FMU, and ≈2.93 MB for the normal binary FMU.

²²<https://pygments.org/>

- *Simulink*: Generate a MATLAB[®] script, that when executed on a Simulink C Function block instantiates it to execute the respective production code. The GALEC block-interface (and therefore in-, outputs and parameters of the production code) are properly typed with native Simulink embedded real-time target (ERT) types such that unnecessary type conversions can be avoided; as is any copying of multidimensions. The resulting block can be used, and combined in Simulink like any other block. ERT code generators will take care of proper system integration of the C Function block, but otherwise use its backing eFMU production code as is (i.e., skip code generation for it).
- *Arduino*: Exports production code as Arduino sketch, adding the required `*.ino` file with initialization and main loop as well as commented code where to access inputs, outputs, and parameters. Exported sketches are ready for further development and deployment via the Arduino IDE, or any other Arduino compatible tooling.

For all export/import options the high-quality standards of eFMU production code as generated by Dymola are preserved, including being free of additional library dependencies besides common C Standard Library routines (mostly `math.h`).

5 STM32 integration of eFMUs

The [controller](#) and [plant](#) eFMUs generated in Section 4 still need to be system integrated on physical microcontrollers. The STM32 (STMicroelectronics[®]) ecosystem has been used as the target here since it is widely-used, low-priced, and well-documented with good development and software support. Being mainly a hardware manufacturer—and as often the case in the embedded domain—STMicroelectronics' viewpoint starts from the hardware and not the model. System integration starts with the selection of one of the target microcontrollers in STM32CubeMX, where peripherals like GPIO pins, DACs, and timers are configured, with template code ultimately being generated, initializing and provisioning all configured peripherals based on the hardware abstraction layer (HAL), lower-layer (LL) drivers, and Arm[®] Cortex[®]-M CMSIS provided by the used STM32CubeH7 software package managed²³ in STM32CubeMX. The result of STM32CubeMX template generation is a C development project for STM32CubeIDE, whose `main()` execution routine (`int main(void)` function in `main.c`) is ready for adding the actual user-code interacting with the configured peripherals. In our case, eFMI system integration therefore means to integrate the 32-bit floating-point precision production codes of the [controller](#) and [plant](#) eFMUs into the `main()` execution routines of their respective template codes generated by STM32CubeMX. Note the plural; [controller](#) and [plant](#) have their individual

STM32CubeIDE projects.

A direct STM32 system integration not using tooling like STM32CubeMX and STM32CubeIDE would be possible too, but that requires professional knowledge we consider rare in our development process, which is driven by the modeling community; it also would be a laborious task not paying off for sporadic use-cases.

5.1 Board configuration with STM32CubeMX

Hardware configuration: Two STM32 Nucleo-64/144 boards were chosen for deploying the eFMU production codes developed in Section 4. A NUCLEO-L476RG was used to deploy the [controller](#), where the system clock of 80 MHz was sufficient to support real-time evaluation of the model. A NUCLEO-H723ZG was selected to run the [plant](#). The much higher 550 MHz system clock is preferable for performing the fast matrix algebra required by the corresponding production code. Both the NUCLEO-L476RG and the NUCLEO-H723ZG have (at least) one analog input (ADC) and one analog output (DAC) for mapping the input and output signals.

Pin configurations: The analog and digital I/O must be mapped to prevent peripheral collisions and minimize latency during the 5 kHz execution loop (i.e., 0.2 ms sampling period). Digital LEDs are employed to signal real-time violation and production code errors, whereas digital output pins are enabled to examine execution times through an oscilloscope by means of low/high voltage toggling at each sampling point and a separate high-voltage signal whenever eFMU production code is in execution. Digital inputs, such as the onboard push-button for the manually controlled [plant](#) fault condition trigger, were also mapped in the STM32CubeMX configuration. For example, for the NUCLEO-H723ZG used for the [plant](#), the primary analog input (the voltage output of the PSS [controller](#) board) was assigned to GPIO pin `PA3`, corresponding to ADC1 channel 15; and the calculated speed output (ω) was assigned to GPIO pin `PA5`, which connects directly to the DAC1 output channel. A similar approach was taken for the [controller](#) I/O configuration, whose NUCLEO-L476RG pin configuration is shown in Figure 14.

We also prototyped more advanced plant models supporting variable loads via an additional input that can be interactively changed with a potentiometer²⁴. In that case, the additional analog input has been deliberately mapped to GPIO pin `PC0` because it can physically route to an entirely independent analog-to-digital converter, ADC2 (specifically `ADC123_INP10`). Distributing the analog input signals across separate, dedicated ADCs avoids the need for complex “Scan Mode” channel switching or Direct Memory Access (DMA) pipelines. This hardware-level isolation ensures deterministic, low-latency sampling within the tight 0.2 ms timer interrupt.

²³STM32CubeMX proposes a catalog of locally installed and online available STM32CubeH7 versions suited for the target microcontroller.

²⁴Although not further discussed for space considerations, the variable-load variant has been successfully deployed and tested.

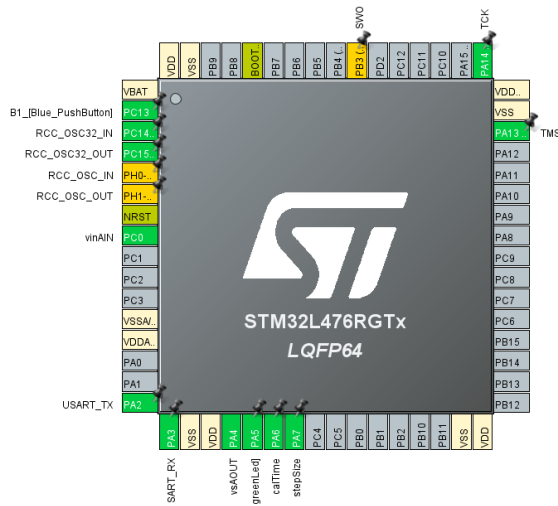


Figure 14. STM32CubeMX pin configuration for the NUCLEO-L476RG used as the **controller**.

Clock configurations: In Dymola, the sampling period of eFMU configurations is explicitly set (see Section 4.1). For the STM32CubeMX configurations, the clock dividers and pre-dividers must be calculated to match the exact sampling period set in the eFMU. The timer’s clock rate (its frequency) is determined by the following equation:

$$\text{frequency} = \frac{\text{APB Clock}}{(\text{ARR} + 1)(\text{PSC} + 1)}$$

where ARR stands for Auto-Reload Register and PSC stands for the timer's prescaler. The APB Clock is the system clock that has been pre-divided for the peripherals (such as timers). The limitations of the ARR and prescaler will be displayed in the STM32CubeMX GUI.

For example, the embedded board that is used as the [plant](#) is required to sample every 0.2 ms (i.e., at 5 kHz frequency). To this end, the NUCLEO-H723ZG, which has a 550 MHz system clock, is configured to have an APB pre-divider of 2, an ARR of 199, and a PSC of 274. That results in the desired frequency:

$$\text{frequency} = \frac{550/2 \text{ MHz}}{(199+1)(274+1)} = 5 \text{ kHz}$$

These clock configuration parameters are set in the timer peripheral menu of STM32CubeMX. For the [plant](#) board, TIM2 was used with the parameters found above, as shown in Figure 15.

The timer configuration for the **controller** is likewise, except that TIM1 is used instead of TIM2 and that its APB Clock, ARR, and PSC values are different to achieve a 0.2 ms sampling period because the NUCLEO-L476RG has a 80 instead of a 550 MHz Arm Cortex-M processor.

Code generation configurations: To maintain a clean file structure—in particular for the generated `main()` execution routine in `main.c`—the code generator of STM32CubeMX was set to generate peripheral initialization as a pair of `.c/.h` files per peripheral. The respective setting is in Project Manager → Code Generator → “Generate peripheral initialization as a pair of `.c/.h`

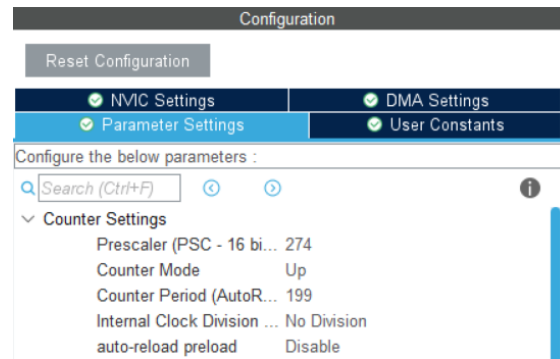


Figure 15. SMT32CubeMX timer configuration for the NUCLEO-H723ZG used as the **plant**.

files per peripheral”. Additionally, the target IDE was set to STM32CubeIDE under Project Manager → Project → Toolchain/IDE.

Generated template code overview: The template code generated by STM32CubeMX serves as the foundational bare-metal layer for the microcontrollers. Upon boot, the code initializes the ARM Cortex-M processor and configures the system clock tree to drive the NUCLEO-H723ZG at 550 MHz and the NUCLEO-L476RG at 80 MHz. The GUI shows which specific GPIO pins can be mapped to ADC and DAC channels, and initializes the hardware timers using the ARR and PSC values calculated to ensure that the timer interrupts fire at the exact 5 kHz rate required by each of the sampling periods configured in Section 4.1. In addition, initialization code for the HAL is generated. Because the eFMU’s production code is system integrated into the generated template code, it is critical that all custom logic, like eFMU function calls, is placed strictly within the designated `/* USER CODE BEGIN / and / USER CODE END */` tags; otherwise, code regeneration during subsequent hardware reconfigurations would overwrite the manually added production code integration.

5.2 Programming, compilation, and deployment with STM32CubeIDE

eFMU production code inclusion: Rather than manually copying the production codes generated for the [controller](#) and [plant](#) into their respective STM32CubeIDE projects, the Production Code containers of the respective eFMUs were symbolically linked using relative paths based on project environment variables provided by STM32CubeIDE (e.g., `PARENT-3-PROJECT_LOC` for the level-3 parent directory of the project root directory). This allows the [controller](#) and [plant](#) Modelica source models to be iteratively tweaked and re-exported from Dymola as eFMUs without requiring manual production code re-import in their respective STM32CubeIDE projects (just refreshing the projects is sufficient)—an important feature for iterative, rapid prototyping cycles from a Modelica model to deployable embedded code. Besides

symbolic linking, the eFMU production codes also have to be added to the list of include and source file directories such that STM32CubeIDE takes care of their compilation and binary code generation.

Manual system integration in main.c: Manual coding to system integrate the eFMU production codes is only required in the main.c files generated by STM32CubeMX. The required steps are very similar for the **controller** and **plant** and are explained in the following by the example of the **controller**.

First, the eFMU production code must be included via `#include "block.h"` to enable **controller** initialization, recalibration, and sampling via the eFMI GALEC block-interface routines `Startup()`, `Recalibrate()`, and `DoStep()`. Note that the implementation generated by Dymola uses hashed names, such that name clashes when several eFMUs are compiled and linked together are avoided. This is not required in our example—after all **controller** and **plant** are individual STM32CubeIDE projects deployed on individual boards—for which reason we can introduce simplified names via `#define` preprocessor macros that ease the readability of subsequent system integration code:

```
1 /* USER CODE BEGIN Includes */
2 #include "block.h"
3 /* USER CODE END Includes */
4
5 /* USER CODE BEGIN PTD */
6 #define CONCAT_INNER(a, b) a##b
7 #define CONCAT(a, b) CONCAT_INNER(a, b)
8
9 /* Shortcut names for hash-based eFMU API: */
10 #define MODEL_HASH \
11     H225c1baf6cf5a31bc9b0c38998c...
12
13 typedef CONCAT(BlockState_, MODEL_HASH) \
14     ModelPSS;
15 typedef CONCAT(SPE_Real_, MODEL_HASH) \
16     PSS_Real;
17 typedef CONCAT(SPE_ErrorSignal_, MODEL_HASH) \
18     PSS_ErrorSignal;
19
20 #define PSS_NONE_ERROR_SIGNAL \
21     CONCAT(SPE_ERROR_SIGNAL_NONE_, MODEL_HASH)
22 #define PSS_UNSPECIFIED_ERROR_SIGNAL \
23     CONCAT(SPE_ERROR_SIGNAL_UNSPECIFIED_ERROR_,
24             MODEL_HASH)
25
26 #define PSS_Startup \
27     CONCAT(Startup_, MODEL_HASH)
28 #define PSS_DoStep \
29     CONCAT(DoStep_, MODEL_HASH)
30 #define PSS_Recalibrate \
31     CONCAT(Recalibrate_, MODEL_HASH)
32 #define PSS_Reinitialize \
33     CONCAT(Reinitialize_, MODEL_HASH)
34 /* USER CODE END PTD */
```

Next, two global variables are introduced to capture sampling requests of the timer interrupt (`pending_samplings`) and to engage or disengage the **controller** interactively (`use_pss`); the latter will be used in Section 6.2 to validate the stabilizing effects of the **controller**:

```
1 /* USER CODE BEGIN PV */
2 /* Sampling request counter & PSS on/off button: */
```

```
3 static volatile uint8_t pending_samplings;
4 static volatile bool use_pss;
5 /* USER CODE END PV */
```

Note that both variables are `volatile` since they shall be changed by interrupt routines²⁵; the `pending_samplings` variable is changed whenever the timer interrupt triggers and `use_pss` whenever the B1 onboard button shown in Figure 16 is pressed. The interrupt routines we need to adapt are predefined by the STM32 HAL; `HAL_TIM_PeriodElapsedCallback` for the timer and `HAL_GPIO_EXTI_Callback` for the onboard button:

```
1 /* USER CODE BEGIN 4 */
2 void HAL_TIM_PeriodElapsedCallback(
3     TIM_HandleTypeDef *htim)
4 {
5     /* Check for sampling timer interrupt: */
6     if (TIM1 == htim->Instance)
7     {
8         if (((uint8_t) 255) > pending_samplings)
9         {
10             ++pending_samplings;
11         }
12         HAL_GPIO_TogglePin(
13             stepSize_GPIO_Port,
14             stepSize_Pin);
15     }
16 }
17
18 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
19 {
20     static uint32_t last_override = ((uint32_t) 0);
21
22     /* Check for blue button (PC13) interrupt: */
23     if (B1_Pin == GPIO_Pin)
24     {
25         const uint32_t current_time = HAL_GetTick();
26         /*
27          * Debounce; only accept press if 250 ms
28          * passed since last:
29          */
30         if (((uint32_t) 250)
31             < (current_time - last_override))
32         {
33             use_pss = !use_pss;
34             last_override = current_time;
35         }
36     }
37 }
38 /* USER CODE END 4 */
```

The checks on Lines 6 and 23 are based on our STM32CubeMX **controller** configuration for the timer (TIM1) and onboard button (B1_Pin) peripherals, and ensure the action is only taken if the respective peripheral caused the interrupt. In case of the timer, we simply increase the `pending_samplings` on Line 10 (ensuring it saturates instead of overflowing); in case of interactive **controller** toggling we just invert `use_pss` on Line 33 (incorporating a debounce time to avoid toggle-cascading).

We also flip a digital output on Lines 12-14 whenever the timer ticks. The resulting `stepSize` pulse signal—whose fixed-length is the sampling period—is useful for

²⁵There is no need for semaphores because the single core ARM Cortex-M processor guarantees atomic access for `uint8_t` and `bool`.

the computation time analyses of Section 6.2 since it provides a reference-grid for oscilloscope-based plots of the time required by the eFMU production code for computing individual samplings (cf. `calTime` of the control-loop listing below).

ARM Cortex-M interrupt routines have defined priority levels. Only routines of higher priority can interrupt others; pending interrupts of lower or equal priority are delayed until higher priority interrupts are complete. The logic of interrupt routines therefore should be concise to avoid interrupt overruns. For that reason, our timer interrupt only registers another sampling request in `HAL_TIM_PeriodElapsedCallback` instead of conducting the actual sampling (which would, in case of an overrun, silently block a following timer tick). The sampling code itself is within the control-loop of the interruptible `main()` execution routine:

```
1 /* USER CODE BEGIN WHILE */
2 while (true)
3 {
4     /* USER CODE END WHILE */
5
6     /* USER CODE BEGIN 3 */
7     /* Conduct sampling if sampling timer ticked: */
8     if (((uint8_t) 0) < pending_samplings)
9     {
10        /* Always reset the error signals: */
11        error_signals = PSS_NONE_ERROR_SIGNAL;
12
13        /* Computation time for external profiling: */
14        HAL_GPIO_WritePin(
15            calTime_GPIO_Port,
16            calTime_Pin,
17            GPIO_PIN_SET);
18
19        /* Input processing (convert AC to volt): */
20        HAL_ADC_Start(&hadc1);
21        if (HAL_OK == HAL_ADC_PollForConversion(
22            &hadc1,
23            POLL_TIMEOUT))
24        {
25            const uint32_t adc_raw =
26                HAL_ADC_GetValue(&hadc1);
27            pss.vSI =
28                ((PSS_Real) adc_raw) * ADC_TO_VOLTS;
29        }
30        else
31        {
32            error_signals |= PSS_UNSPECIFIED_ERROR_SIGNAL;
33        }
34        HAL_ADC_Stop(&hadc1);
35
36        /* Compute sampling step: */
37        PSS_DoStep(&pss);
38        error_signals |= pss.ErrorSignals;
39
40        /* Check if switch is pulling to GND: */
41        const PSS_Real target_pu = use_pss
42            ? pss.vs
43            : CONSTANT_PU;
44
45        /*
46         * Output processing (scaling, clamping,
47         * converting volt to AC):
48         */
49        PSS_Real dac_val =
50            (target_pu + OFFSET) * VOLTS_TO_DAC;
51        if (DAC_MIN > dac_val)
52        {
53            dac_val = DAC_MIN;
54        }
```

```
55     else if (DAC_MAX < dac_val)
56     {
57         dac_val = DAC_MAX;
58     }
59     HAL_DAC_SetValue(
60         &hdac1,
61         DAC_CHANNEL_1,
62         DAC_ALIGN_12B_R,
63         ((uint32_t) dac_val));
64
65     /* Computation time for external profiling: */
66     HAL_GPIO_WritePin(
67         calTime_GPIO_Port,
68         calTime_Pin,
69         GPIO_PIN_RESET);
70
71     --pending_samplings;
72 }
73
74 /* Set LED 2 for sampling errors or overruns: */
75 if ((PSS_NONE_ERROR_SIGNAL != error_signals)
76     || (((uint8_t) 1) < pending_samplings))
77 {
78     error_timestamp = HAL_GetTick();
79     HAL_GPIO_WritePin(
80         LD2_GPIO_Port,
81         LD2_Pin,
82         GPIO_PIN_SET);
83 }
84
85 /*
86  * Unset LED 2 if on due to past errors
87  * (but always hold for at least 30000 ms):
88  */
89 if (((uint32_t) 0) != error_timestamp)
90     && (((uint32_t) 30000)
91         < (HAL_GetTick() - error_timestamp))
92 {
93     HAL_GPIO_WritePin(
94         LD2_GPIO_Port,
95         LD2_Pin,
96         GPIO_PIN_RESET);
97     error_timestamp = ((uint32_t) 0);
98 }
99 }
100 /* USER CODE END 3 */
```

Note that the control-loop is greedy (while (true) on Line 2) instead of going to sleep until events happen; such “busy waiting” is typical for hard real-time code. It consists of three main parts: (1) On Lines 8-72, actual sampling—i.e., the use of eFMU production code to compute new outputs and internal control states for the given inputs and parameters—is conducted whenever the timer interrupt indicated a sampling point (`pending_samplings ≥ 1`). (2) On Lines 75-83 errors—either (a) signaled by the block-interface functions of the eFMU production code (`error_signals` variable set on Lines 37-38) or due to real-time violations, because of (b) non-timely input provision (timeout of `HAL_ADC_PollForConversion` on Lines 21-23) or (c) sampling computation overruns when the timer ticks again while a previous sampling still is processed (`pending_samplings > 1` check on Line 76)—are detected and indicated by the onboard LED 2. (3) On Lines 89-98, LED 2 is reset if there have been no errors within the last 30 s.

Note that sampling on Lines 8-72 is mostly about input and output provisioning with respective timeout detec-

tion and AC/voltage conversions with scaling and clamping according to voltage ranges. The actual sampling logic is completely encapsulated in the `eFMI DoStep()` implementation of the eFMU production code (called on Line 37), which takes as argument a struct holding the current **controller** state (`pss` holding the values of tunable and dependent parameters, in- and outputs, and states). The struct type is provided by the eFMU production code as is a respective `eFMI Startup()` implementation for its initialization. The only other noteworthy sampling actions are (1) to indicate the actual computation time with the `calTime` pulse signal (set to high-voltage when sampling starts on Lines 14-17 and reset on Lines 66-69 when it ends), (2) engage or disengage—not disable; just whether its output shall be used—the **controller** on user request on Lines 41-43, and, of course, (3) reduce the sampling request counter on Line 71 when finished.

Compiler and linker settings: STM32CubeIDE provides various options for compilation and binary code generation. Our **controller** and **plant** projects each provide two predefined build configurations: A *Debug* and a *Release* build. Both share the following common settings (unmentioned options are left default):

- **MCU/MPU Toolchain:** The GNU Tools for STM32 (14.3.rel1) are used for compilation and binary code generation.
- **MCU/MPU Settings:** Use of the reduced C runtime library (`-specs=nano.specs`) and hardware-implemented floating-point ABI (`-mfloat-abi=hard`). For the NUCLEO-L476RG, where the **controller** is deployed, its FPUv4-SP-D16 floating-point unit is used; for the NUCLEO-H723ZG, its FPUv5-D16 floating-point unit is used.
- **MCU/MPU GCC Assembler:** The preprocessor does not search system directories (`-nostdinc`) to avoid inclusion or configuration by external—third party set—macros.
- **MCU/MPU GCC Compiler:** Reports are verbose (`-v`). ISO C18 (`-std=c18`) is used as language standard to verify portability and compliance of the eFMU production code generated by Dymola. All warnings are enabled (`-Wall`, `-Wextra`, `-pedantic`, `-Wmissing-include-dirs`, `-Wswitch-default`, `-Wswitch-enum`, and `-Wconversion`). As expected from the production code analyses of Section 4.3, the eFMU production codes are free of any warnings except the 32-bit floating-point narrowing conversions already explained; interestingly, the driver/standard library implementation of STM32 has many warnings.
- **MCU/MPU GCC Linker:** Reports are verbose (`-v`). Standard libraries are not default included (`-nodefaultlibs`) and a minimal implementation for system calls is used (`-specs=nosys.specs`)

to verify that the eFMI production code generated by Dymola is self-contained and without external library dependencies besides basic C Standard Library dependencies.

The *Debug* and *Release* builds only differ in applied optimizations and available debug support:

- **MCU/MPU GCC Assembler:** The debug level is maximum (`-g3`) for the *Debug* build and none for the *Release* build; likewise for the MCU/MPU GCC Compiler settings, where additionally the `-DDEBUG` preprocessor directive is defined for the *Debug* build but not for the *Release* build.
- **MCU/MPU GCC Compiler:** The optimizations are none (`-O0`) for the *Debug* build and most (`-O3`) for the *Release* build. Stack usage analysis (`-fstack-usage`) and cyclomatic complexity (`-fcyclomatic-complexity`) are enabled for the *Debug* build but not for the *Release* build.

6 RT simulation and CHiL testing

All analyses in this section have been conducted with *Release* builds if not mentioned otherwise explicitly (cf. Section 5.2).

6.1 CHiL experimental setup

Schematics and wiring: To perform CHiL tests, Figure 16 shows how the microcontrollers have been wired together along with an Analog Discovery 3²⁶, a multi-functional test and measurement device (labeled (M)). On the left hand side, the NUCLEO-L476RG, labeled (C), serves as the **controller** running the PSS eFMU (from the model in Figure 4 (D)), while on the right hand side the NUCLEO-H723ZG, labeled (P), acts as the **plant** running the other eFMU (from the model in Figure 5).

The speed output signal `w` of the **plant**, in Figure 5, is now physically obtained from Figure 16(P) (the NUCLEO-H723ZG) on pin D13. It is connected with a yellow wire to both the measurement channel C1 of the Analog Discovery Board 3 (M) and to pin A5 of the **controller**, the NUCLEO-L476RG, labeled (C) in Figure 16. Similarly, the output signal `vf` of the **controller** in Figure 4 (D)) is obtained from the NUCLEO-L476RG in Figure 16(C) from pin A2. It is then connected with green wires to both the measurement channel C2 of the Analog Discovery Board 3 and to pin A0 of the **plant**, the NUCLEO-H723ZG.

Interactive fault injection, stabilizer toggling, and resetting: In traditional offline Modelica simulations, events like faults, reinitialization, or enabling or disabling a controller are typically triggered by absolute time components (e.g., the `faultTimer` in our offline simulation models of the CHiL setup, shown in Figure 8). In a continuous HiL environment, absolute time based events are

²⁶<https://digilent.com/shop/analog-discovery-3/>

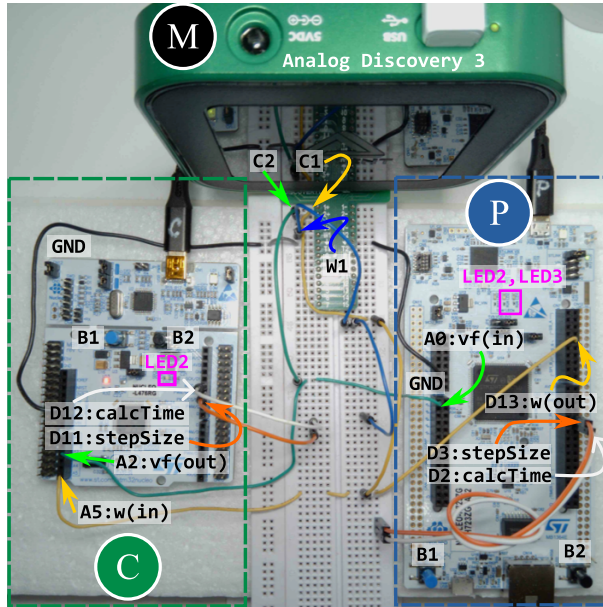


Figure 16. CHiL experimental setup wired to record measurement data from experiments using the Analog Discovery 3.

problematic however; there might be no well-defined, synchronized global clock for all devices, multiple consecutive test runs require synchronous resetting of all boards, and interactive testing is hampered. For these reasons, we decided for interactive, onboard button based, event triggering.

Manual **plant** faults can be triggered by pushing the blue “User Button” on the NUCLEO-H723ZG, which is labeled B1 on the bottom right of Figure 16. Pressing the button pulls the GPIO pin high, instantly activating the fault dynamics of the **plant** eFMU, while releasing it clears the fault, allowing for interactive, repeatable time-domain testing.

In case of the **controller**, the corresponding “User Button” is configured to engage or disengage the eFMU itself. When the **controller** button B1, on the left of Figure 16, is pressed, it will “turn-on” the PSS, giving an output driven by the eFMU, and pressing it again will “turn-off” the PSS, giving a constant output.

Both boards, the L476RG and the H723ZG, can be individually reset using the buttons labeled B2, on the left hand side for the **controller** and on the bottom right hand side for the **plant**.

Analog Discovery 3 oscilloscope and network analysis: The Analog Discovery 3 interfaces with Digilent’s WaveForms¹⁷ software package. This software is capable of acquiring oscilloscope measurements through two analog channels (see C1 and C2 in Figure 16), allowing real-time analysis of the analog signals used by microcontrollers during experiments. Additionally, the Analog Discovery 3 includes a waveform generator, allowing for test signal injection into the Nucleo-64/144 boards so that their functionality could be tested independently. The WaveForms software also includes a network analysis tool, employing

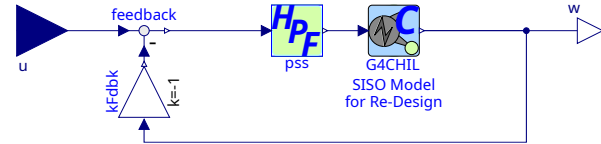


Figure 17. Closed-loop model with input and output interfaces to compare with frequency response tests.

both the oscilloscope and waveform generator hardware features. This tool allows for the frequency dependent amplitude and phase responses of the circuit under test to be captured easily. This is achieved by connecting the waveform generator, W1 in Figure 16, and one analog probe, C1, to the input of the system, with the second analog probe, C2 connected to the output of the system. Through this network analysis function, frequency response plots were generated and compared with the Bode plots generated by linearizing the Modelica models.

Full closed-loop network analysis setup: Because the experimental CHiL setup includes two embedded boards in closed-loop, it is desirable to verify the frequency response characteristics of this closed-loop system as well. Figure 17 shows the Dymola model that was used to compare with the hardware-based closed-loop test.

To obtain measurements of the closed-loop frequency response without violating the electrical constraints of the STM32 hardware, a custom signal injection circuit was required. The STM32’s Analog-to-Digital Converter (ADC) operates on a single-supply 3.3 V logic level and cannot physically process negative voltages. Injecting a standard 0 V offset AC sweep directly into the ADC would result in severe signal clipping and non-linear distortion. As shown in the schematic in Figure 18, a non-inverting summing amplifier was constructed using a MAX474 operational amplifier. The adder takes the nominal steady-state speed output of the **plant**, which rests at a flat 1.0 V DC, and sums it with the small AC signal (± 50 mV, 0 V offset) generated by the Analog Discovery 3 from pin W1. This outputs a safely biased signal fluctuating between 0.95 V and 1.05 V into the **controller**’s ADC. By measuring the injected stimulus on channel C1 and the **plant**’s speed output on channel C2, the true closed-loop transfer function was empirically captured.

6.2 Results and analyses

Time-domain analyses: To establish a baseline, the system was first tested having the PSS eFMU functions disengaged. Figure 19 displays the time domain plot of the generator’s speed deviation (blue trace) when subjected to a fault without engaged **controller**. In this case, the **plant** essentially behaving as an undamped pendulum, exhibiting sustained electromechanical oscillations, which are further excited following a later fault triggering making the **plant** unstable.

The system’s response with the tuned **controller** active in the control loop is shown in Figure 20. The injection of

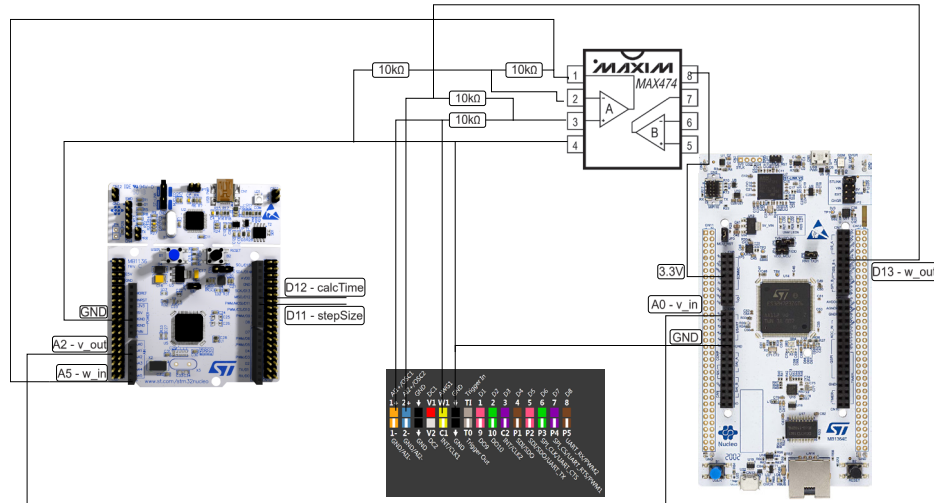


Figure 18. Closed-Loop wiring diagram for network analysis experiments with Analog Discovery 3.

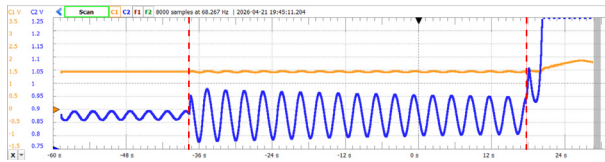


Figure 19. Experiment with the PSS disengaged. Orange: PSS output voltage (pu) from the Nucleo-L476RG, Blue: scaled speed from the NUCLEO-H723ZG. Red dashed: fault applied.

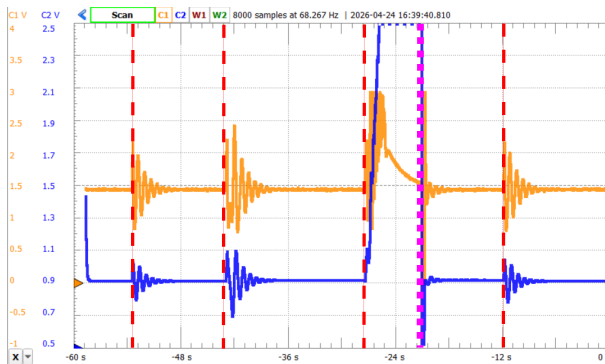


Figure 20. Experiment with PSS engaged. Orange: PSS output voltage (pu) from the NUCLEO-L476RG, Blue: scaled speed from the NUCLEO-H723ZG. Red dashed: fault applied. Magenta dotted: reset of both boards.

the **controller** voltage (shown in orange) provides immediate active damping. The generator speed (shown in blue) damps to a steady state within seconds. Furthermore, the active **controller** allowed the physical hardware simulation to recover successfully from a ± 0.2 pu speed deviation, whereas the uncompensated **plant** became unstable when subjected to deviations greater than ± 0.1 pu.

For both experiments in Figures 19 & 20, excessively large faults were applied to the system (the **plant** B1 button was held down for long periods); evident where the generator speed would become unstable, rapidly increasing, and eventually saturating. This is because the **controller** has a limited operational window of generator

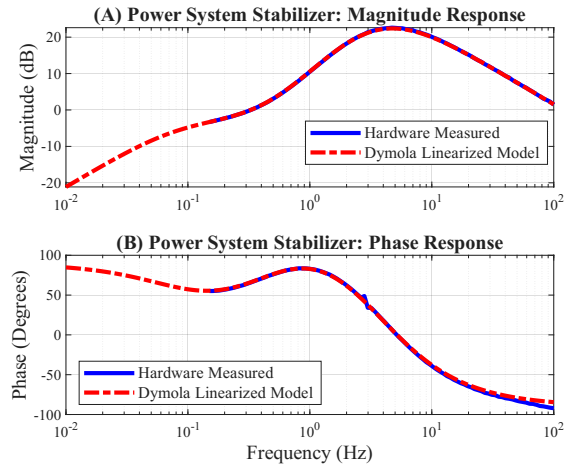


Figure 21. Frequency response of the **controller**. Blue: NUCLEO-L476RG running the eFMU, Red: Dymola linearized model.

speeds it can stabilize.

Frequency-domain analyses: The WaveForms network analyzer was configured to perform logarithmic frequency sweeps from 0.15 to 100 Hz. A signal amplitude of 50 mV was selected for the waveform generator to ensure that the **plant** AVR limits were never reached during the test.

First, an open-loop frequency response experiment was conducted on the **controller**. Figure 21 shows the Bode plot obtained by analyzing the linear model generated from Dymola (by linearizing the model in Figure 4(D)) in a red dashed line, along with the measured Bode plot measured from the hardware (blue line). The excellent agreement of both traces confirms that the **controller** simulation is running as expected. If the timer's time step on the NUCLEO-L476RG was not configured to exactly match the time step set in Dymola's eFMU export, the entire graph would maintain the same shape but shift horizontally along the frequency axis.

The closed-loop experiment was then conducted. Fig-

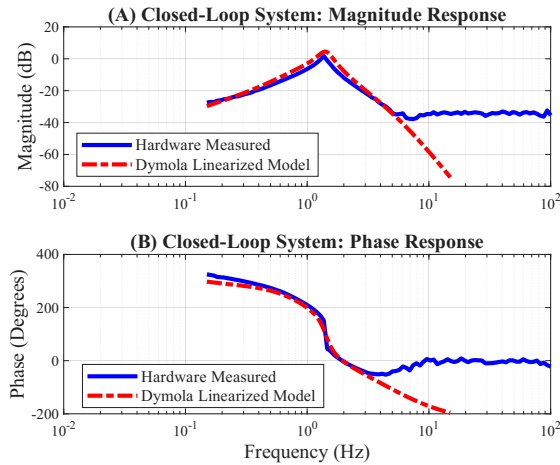


Figure 22. Frequency response of the closed-loop test. Blue: hardware measurements, Red: Dymola linearized model.

ure 22 shows the Bode plot obtained by linearizing the model in Figure 17 in Dymola, and the one measured during the experiment. Although there is no perfect alignment in this case, they match closely enough to indicate that the eFMU is producing results that are highly consistent with the offline simulation. Measurements at ≈ 5 Hz appear to flatten according to Figure 22 (B), albeit with a jagged nature due to noise. This is likely due to the Analog Discovery 3 hitting its measurement noise floor, where it cannot accurately resolve the frequency response at that level of sensitivity anymore, and the limitations of the used onboard ADCs and DACs.

Computation time analyses: A critical requirement of the CHIL tests is to validate that there are no real-time violations (called overruns), i.e., the **controller** and **plant** always finish computing a sampling before a subsequent timer interrupt triggers, requesting the next sampling cycle to be evaluated. To verify this, onboard LED 2 on either board is used (cf. Figure 16); whenever an overrun is detected, it turns on green for 30 s for the **controller** or yellow for 5 s for the **plant**. Note, that LEDs are also used for indicating error signals from the eFMU production codes (like when the embedded linear solver failed due to a singularity). Since the NUCLEO-L476RG of the **controller** only has LED 2 available for user programs, it is shared for overrun detection and error signals; thus, if it turns on it could be either an overrun or an error signal. For the NUCLEO-H723ZG of the **plant**, LED 3 (red) is used to indicate error signals.

Although LEDs are helpful for diagnostic purposes, they do not provide any information to which degree real-time requirements are satisfied (if they are); but the headroom for potential additional computations is a good indication if the chosen board is a good fit considering also some security-buffer in case future model changes may cause reasonable additional computational-effort. Therefore, digital GPIO pins were configured to turn on high at the exact start of sampling and to turn off low immediately upon completion (labeled D12 in Figure 16 for the

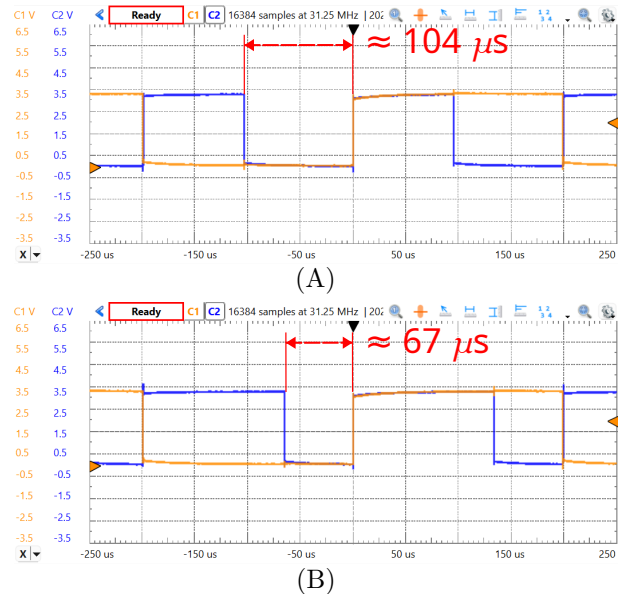


Figure 23. Computation time (blue) and sampling timer (orange) of the NUCLEO-H723ZG executing the 32-bit floating-point precision eFMI production code (*Release* build) of the **plant** under (A) steady state, (B) sustained fault injection.

controller and D2 for the **plant**). By monitoring the respective pin of either board with the oscilloscope, the true computation time, `calcTime`, can be physically measured. To be able to plot the computation time relative to the sampling period of that board, timer interrupts are also toggling a separate digital GPIO pin, the `stepSize` (labeled D11 in Figure 16 for the **controller** and D3 for the **plant**).

For the *Release* build of the **plant** the oscilloscope measurements in Figure 23 show that its computation time is well within the 0.2 ms time window between two samplings, with $\approx 52\%$ headroom in steady state ($\approx 96\mu\text{s}$ computation time) as shown in (A) and $\approx 33.5\%$ headroom ($\approx 133\mu\text{s}$ computation time) during sustained fault as shown in (B). The low-cost NUCLEO-H723ZG with its 550 MHz ARM Cortex-M7 can be used as real-time platform for OpenIPSL power generator simulations.

As can be seen comparing Figure 23 (A) and (B), there are significant computational differences whether faults are injected or not. The GALEC code of the **plant** is dominated by a system of linear equations $Ax = b$, with the respective GALEC built-in function call `solveLinearEquations(A, b)`. If no fault is injected, many elements of the A -matrix and b -vector become 0, i.e., A and b are sparse. The implementation of `solveLinearEquations` in the generated production code avoids unnecessary computations in case of 0-elements. As a consequence, computation time drops significantly if no fault is injected. For that reason, our real-time capability validation in Figure 23 (B) is under sustained fault injection regardless of the **plant** becoming unstable beyond the point of recovery.

Note that the *Debug* build of the **plant** overruns in case

of sustained fault injection ($\approx 239\mu$ computation time, i.e., $\approx 19.5\%$ overrun); it is not suited for hard real-time application on the NUCLEO-H723ZG, stressing the requirement for good SiL testing support as demonstrated in Section 4.2 enabling offline debugging in case of functional errors without real-time constraints. For completeness, we also investigated the 64-bit floating-point precision production code of the [plant](#); the *Release* build is only just real-time suited with $\approx 9\%$ headroom ($\approx 180\mu$ computation time) in case of sustained fault. We consider $\approx 9\%$ headroom to tight however, to be on the safe side anticipating future minor model changes. Of course, the *Debug* version of the 64-bit floating-point precision [plant](#) is out of question with already $\approx 21\%$ overrun without sustained fault and $\approx 39.5\%$ overrun with.

For the [controller](#), the *Release* build has $\approx 92.17\%$ headroom ($\approx 15.67\mu$ s computation time), regardless whether faults are injected in the [plant](#) or not. In the case of the [controller](#), also the *Debug* build can be deployed with a comfortable headroom of $\approx 82.83\%$ ($\approx 34.35\mu$ s computation time). Ample headroom for the [controller](#) is not only beneficial for further, more advanced, stabilizer models, but also crucial in the safety-critical domain, where extensive automatic compiler optimizations like `-O3` in case of the *Release* build might be considered potentially unsafe for which reason only the `-O0` non-optimized code of the *Debug* build can be deployed.

7 Conclusions and future work

We presented a MBSE workflow that streamlines power system stabilizer design and deployment on low-cost microcontrollers leveraging the eFMI Standard. By introducing the `OpenIPSL_CHIL` library, we established a “single source of truth” from physics-based, acausal Modelica models to production C code, clearly separating physics modeling from embedded development. All models, eFMUs, and C projects are provided open-source⁷ as an extensible workbench framework ultimately empowering researchers to rapidly prototype grid controllers with guaranteed traceability and significantly reduced capital expenditure.

Our systematic MiL, SiL, and CHiL design processes do not only guide development from off-line, continuous simulation models toward safety-critical, hard real-time suited, sampled embedded application, but also enable validation of intermediate designs of the various abstraction levels, significantly reducing the risk of late-stage issues with hard to trace sources of error. Verifying functional correctness via SiL testing provides strong confidence for deployment; once SiL tests pass, HiL system integration mainly involves checking memory fit and real-time execution on the target.

The mostly automatized MBSE process thereby supports iterative development cycles from Modelica models to deployable embedded code, enabling rapid prototyping for target switches and remodeling. The presented

plant, for example, had to be redeployed on several targets due to real-time violations and IO pin restrictions before finally settling with the NUCLEO-H723ZG; and likewise the controller design underwent several iterations for signal-noise handling and real-time suited parameter-fitting before CHiL results have been satisfactory. And instead of using the built-in ADCs and DACs of the selected microcontrollers as well as the power supplies provided onboard, we could improve signal-to-noise ratios by using dedicated external ADC and DAC components with careful design without further microcontroller upgrades; such hardware setup redesign would have to be reflected in the model (e.g., filter configurations, parameters, etc), confirming the need of an MBSE-supported iterative development process along the lines of eFMI.

Besides systematic functional testing, we also validated the non-functional quality of the production code generated by the used eFMI tooling. Production code analyses confirm MISRA C:2023 and SEI CERT C Coding Standard¹¹ compliance; execution time measurements confirm real-time capability; and the rigorous error signaling of eFMI GALEC confirms the absence of numerical issues.

Last but not least, our approach dramatically reduces capital expenditure. It reduces software licenses to only Dymola (with *Dymola Source Code Generation* option) while using free development tools for the STM32 microcontrollers. And although eFMI was not designed for real-time plant simulation, we show that it has strong potential for that use, eventually replacing proprietary real-time platforms that typically cost $\gg \$50,000$ with $< \$50$ microcontrollers. This came as a surprise to us. Our original plan was to use one of the established real-time platforms, like from dSPACE, for the power generator plant-model (and only deploy the PSS controller-model on a microcontroller); we only tried eFMI-based deployment of the plant-model out of curiosity how far we can get with Dymola’s eFMI tooling, but soon figured this may indeed work out!

Future research will concentrate on both the modeling and eFMU integration sides. We intend to work on scaling our current proof-of-concept to cover a broader spectrum of PSS models, and on creating templates that support a variety of embedded hardware boards from the STM32 family. An important objective is to exploit the eFMI recalibration to enable online parameter tuning, thereby allowing the PSS to adapt its behavior to real-time grid conditions. Finally, we plan to showcase the integration of eFMUs (exported as source code FMUs or the eFMI production code directly system integrated) into industrial real-time platforms from dSPACE, Typhoon HIL, and OPAL-RT Technologies.

Acknowledgment

The authors would like to thank:

- Fabio Ballari and the CATIA® Champions program at Dassault Systèmes® for providing complimentary ac-

cess to the 3DEXPERIENCE® platform²⁷ and its Systems Software Production Engineer (SOP-OC) capabilities, which enabled the early development of this work.

- Dr. Christopher R. Laughman at Mitsubishi Electric® Research Laboratories for an unrestricted gift grant to Rensselaer Polytechnic Institute® that funded the embedded hardware behind this project.
- Daniel Marjamäki and Magnus Janagård of Cppcheck Solutions AB¹³ for providing a free license of Cppcheck Premium.

Appendix: Related work

Previous research has transitioned advanced controllers to real-time platforms, such as National Instruments hardware-based wide-area controllers in (Vanfretti, Jónsdóttir, et al. 2018), (Zhang et al. 2021) and (Schoenwald et al. 2017). Moreover, recent industrial trials by Red Eléctrica and RTE on HVDC interconnectors (Renedo et al. 2023) demonstrate that aligning simulation models with actual hardware behavior is now paramount.

Industrial platforms like GE's PhasorController (Golshani et al. 2021) utilize deterministic PLC engines for real-time grid control; however, these solutions typically rely on manual imperative programming rather than the automated code synthesis central to a formal MBSE workflow, which severs the traceability between the physical model and the target implementation. The limitations of manual coding are particularly acute in enabling modern adaptive control strategies; for example, recent Terna research proposes online PSS tuning based on estimation (Lauria et al. 2024), while our previous work demonstrates the need for a continuous controller redesign (Bombois and Vanfretti 2024; Vanfretti and Bombois 2024).

This lack of traceability is similarly evident in the field of power system simulation, where the exchange of models within industry remains limited. The IEEE/CIGRE DLLs Guideline (ICDMS) (CIGRE Working Group B4.82/IEEE 2025) attempts to address model sharing by distributing vendor algorithms as precompiled DLLs; nevertheless, these remain 'black boxes' containing likely hand-written code, continuing to sever the connection to the underlying physics. As such DLLs are not suitable for real-time simulation, CHiL testing remains limited. This is even more challenging when trying to migrate large-scale models, such as the Continental Europe model, to real-time targets, which remains problematic (Ramirez-Gonzalez et al. 2024). However, recent work has championed Modelica and FMI (Modelica Association 2020) to solve this, enabling models from the Open-Instance Power Systems Library (OpenIPSL) to be executed directly on dSPACE hardware (de Castro, Laera, et al. 2023).

In this context, the eFMI Standard (Modelica Association 2024; Lenord et al. 2021) enables a continuum from

offline simulation into the domain of hard real-time and safety-critical embedded software. Unlike opaque DLLs, eFMI guarantees traceability via GALEC, ensuring that production code is systematically generated from a source physics model. Although proven in automotive applications (Ultsch et al. 2021; Kamp et al. 2025; Armugham et al. 2021), its application in power system control is first explored in this paper.

References

- Armugham, Siva Sankar, Oliver Lenord, Kai Werther, Christian Bertsch, and Karthikeyan Ramachandran (2021). "eFMI (FMI for Embedded Systems) in AUTOSAR for Next Generation Automotive Software Development". In: *Symposium on International Automotive Technology*. SAE International. DOI: 10.4271/2021-26-0048.
- Baudette, Maxime, Marcelo de Castro, Tin Rabuzin, Jan Lavenius, Tetiana Bogodorova, and Luigi Vanfretti (2018). "OpenIPSL: Open-Instance Power System Library – Update 1.5 to "iTesla Power Systems Library (iPSL): A Modelica library for phasor time-domain simulations"". In: *SoftwareX* 7, pp. 34–36. DOI: 10.1016/j.softx.2018.01.002.
- Bhattacharjee, Srijita, Luigi Vanfretti, and Fernando Fachini (2024). "Building Power System Models for Stability and Control Design Analysis using Modelica and the OpenIPSL". In: *Proceedings of the American Modelica Conference 2024*, pp. 63–71. DOI: 10.3384/ECP20763.
- Bombois, Xavier and Luigi Vanfretti (2024). "Performance monitoring and redesign of power system stabilizers based on system identification techniques". In: *Sustainable Energy, Grids and Networks* 38, p. 101278. DOI: 10.1016/j.segan.2024.101278.
- Bürger, Christoff (2019). "Modelica language extensions for practical non-monotonic modelling: on the need for selective model extension". In: *Proceedings of the 13th International Modelica Conference. Update (2023-05-09): The selective-model extension mechanism for non-monotonic, inheritance-based model variation proposed in the paper is part of Modelica since version 3.6.*, pp. 277–288. DOI: 10.3384/ecp19157277.
- CIGRE Working Group B4.82/IEEE (2025). *Guidelines for use of real-code in EMT models for HVDC, FACTS and inverter based generators in power systems analysis*. Tech. rep. 958. CIGRE.
- de Castro, Marcelo, Giuseppe Laera, et al. (2023). "Power System Real-Time Simulation using Modelica and the FMI". In: *Proceedings of the American Modelica Conference 2022*, pp. 85–94. DOI: 10.3384/ECP2118685.
- de Castro, Marcelo and Luigi Vanfretti (2025). "OpenIWPI: Open-Instance Wave-Phasor Interface Library for Power System Simulation Studies in Modelica". In: *Proceedings of the 16th International Modelica & FMI Conference*, pp. 223–232. DOI: 10.3384/ecp218223.
- de Castro, Marcelo, Luigi Vanfretti, and Francisco J. Gómez (2019). "A Fundamental Time-Domain and Linearized Eigenvalue Analysis of Coalesced Power Transmission and Unbalanced Distribution Grids using Modelica and the OpenIPSL". In: *Proceedings of the 13th International Modelica Conference*, pp. 617–626. DOI: 10.3384/ecp19157617.
- de Castro, Marcelo, Luigi Vanfretti, Francisco J. Gómez, Srijita Bhattacharjee, and Fernando Fachini (2023). "Version

²⁷ <https://www.3ds.com/3dexperience/>

- [OpenIPSL 2.0.0] — [iTesla Power Systems Library (iPSL): A Modelica library for phasor time-domain simulations]”. In: *SoftwareX* 21, p. 101306. DOI: 10.1016/j.softx.2022.101277.
- Elmqvist, Hilding, Martin Otter, and Sven Erik Mattsson (2012). “Fundamentals of Synchronous Control in Modelica”. In: *Proceedings of the 9th International MODELICA Conference*, pp. 15–26. DOI: 10.3384/ecp1207615.
- ENTSO-E (2018). *Oscillation Event 03.12.2017*. Tech. rep. ENTSO-E. URL: <https://tinyurl.com/ENTSOE2017>.
- ENTSO-E (2025). *Grid Incident in Spain and Portugal on 28 April 2025: ICS Investigation Expert Panel Factual Report*. Tech. rep. ENTSO-E. URL: <https://www.entsoe.eu/publications/blackout/28-april-2025-iberian-blackout/>.
- ENTSO-E (2026). *Final Report on the Grid Incident in Spain and Portugal on 28 April 2025*. Tech. rep. ENTSO-E. URL: <https://www.entsoe.eu/publications/blackout/28-april-2025-iberian-blackout/>.
- ENTSO-E Sub-Group System Dynamics and Protection (2017). *Analysis of CE Inter-Area Oscillations of 1st December 2016*. Tech. rep. ENTSO-E. URL: <https://tinyurl.com/ENTSOE2016>.
- Fachini, Fernando, Hao Chang, Tetiana Bogodorova, and Luigi Vanfretti (2025). “Customized open source renewable energy models validated through PHIL lab experiments”. In: *Renewable Energy* 244, p. 122627. DOI: 10.1016/j.renene.2025.122627.
- Golshani, M., D. Wilson, S. Norris, I. Cowan, Md H. Rahman, and B. Marshall (2021). “Application of Phasor-Based Functionality to HVDC Control in Reduced System Strength”. In: *The 17th International Conference on AC and DC Power Transmission (ACDC 2021)*, pp. 44–49. DOI: 10.1049/icp.2021.2442.
- Kamp, Tobias, Christoff Bürger, Johannes Rein, and Jonathan Brembeck (2025). “Hybrid Simulation Models for Embedded Applications: A Modelica and eFMI approach”. In: *Proceedings of the 16th International Modelica & FMI Conference 2025. Update (2025-09-08): For details about Modelica-based neural network modelling with eFMI. NeuralNetworks and embedded code generation preserving the multidimensional arithmetic of tensor-flows*, cf. https://youtu.be/nkugRv_cwnA, pp. 555–565. DOI: 10.3384/ecp218555.
- Laera, Giuseppe et al. (2023). “Guidelines and Use Cases for Power System Dynamics Modeling and Model Verification using Modelica”. In: *Proceedings of the American Modelica Conference 2022*, pp. 146–156. DOI: 10.3384/ECP21186146.
- Lauria, Davide, Fabio Mottola, Giorgio Giannuzzi, Cosimo Pisani, and S. Tessitore (2024). “Power systems stabilizers online tuning based upon parameters dynamic estimation”. In: *Sustainable Energy, Grids and Networks* 39, p. 101481. DOI: 10.1016/j.segan.2024.101481.
- Lenord, Oliver et al. (2021). “eFMI: An open standard for physical models in embedded software”. In: *Proceedings of 14th Modelica Conference 2021*, pp. 57–71. DOI: 10.3384/ecp2118157.
- Milano, Federico (2005). “An open source power system analysis toolbox”. In: *IEEE Transactions on Power Systems* 20.3, pp. 1199–1206. DOI: 10.1109/TPWRS.2005.851911.
- Mishra, Chetan, Luigi Vanfretti, Mark Baldwin, Jaime Delaree Jr., and Kevin D. Jones (2024). “Analysis of Generator Forced Oscillations During MOD 25 Testing Exploiting Wavelets”. In: *Proceedings of the 57th Hawaii International Conference on System Sciences (HICSS-57)*, pp. 1–10. DOI: 10.24251/HICSS.2024.364.
- Mishra, Chetan, Luigi Vanfretti, Jaime Delaree, and Kevin D. Jones (2024). “Analyzing a Non-Sinusoidal Response from a Real-World Solar PV”. In: *IEEE Transactions on Power Systems* 39.2, pp. 4771–4774. DOI: 10.1109/TPWRS.2024.3350377.
- Modelica Association (2020). *Functional Mock-up Interface for Model Exchange and Co-Simulation – Version 2.0.2*. Tech. rep. Modelica Association.
- Modelica Association (2024). *eFMI Standard: Functional Mock-up Interface for embedded systems – Version 1.0.0 Beta 1*. Tech. rep. Modelica Association.
- Olsson, Hans, Sven Erik Mattsson, Martin Otter, Andreas Pfeiffer, Christoff Bürger, and Dan Henriksson (2017). “Model-based Embedded Control using Rosenbrock Integration Methods”. In: *Proceedings of the 12th International Modelica Conference*, pp. 517–526. DOI: 10.3384/ecp17132517.
- PSS®E 34.2.0 Model Library (2017). Siemens Industry, Inc., Power Technologies International (Siemens PTI).
- Pudasaini, Bikal, Luigi Vanfretti, Chetan Mishra, Jaime De La Ree, and Kevin D. Jones (2026). “Irregular Wind Power Plant Dynamics Induced by Power Factor Command Truncation”. In: *2026 IEEE Green Technologies Conference (GreenTech)*, pp. 1–6. DOI: 10.1109/GreenTech68285.2026.11471549.
- Ramirez-Gonzalez, Miguel, Artjoms Obusevs, Felix Rafael Segundo Sevilla, Santiago Sanchez, and Petr Korba (2024). “Real-Time Simulation of the Initial Dynamic Model of CE in ePHASORSim: Issues and Troubleshooting”. In: *2024 IEEE 65th International Scientific Conference on Power and Electrical Engineering of Riga Technical University (RTU-CON)*, pp. 1–6. DOI: 10.1109/RTU-CON62997.2024.10830830.
- Rebello, Eldrich, Luigi Vanfretti, and Muhammad Shoaib Almas (2020). “Experimental Testing of a Real-Time Implementation of a PMU-Based Wide-Area Damping Control System”. In: *IEEE Access* 8, pp. 25800–25810. DOI: 10.1109/ACCESS.2020.2970988.
- Renedo, Javier, Agustín Díaz-García, Gilles Torresan, Eduardo Lorenzo Cabrera, Antonio Córdón, et al. (2023). “Tests on the POD-P controller of INELFE Spain-France VSC-HVDC interconnector”. In: *CIGRE B4 International SC Meeting and Colloquium*. B4-1145.
- Schoenwald, David A., Brian J. Pierre, Felipe Wilches-Bernal, and Daniel J. Trudnowski (2017). “Design and Implementation of a Wide-Area Damping Controller Using High Voltage DC Modulation and Synchrophasor Feedback”. In: *IFAC-PapersOnLine* 50.1, pp. 67–72. DOI: 10.1016/j.ifacol.2017.08.012.
- Ulsch, Johannes, Julian Ruggaber, Andreas Pfeiffer, Christina Schreppel, Jakub Tobolář, Jonathan Brembeck, and Daniel Baumgartner (2021). “Advanced Controller Development Based on eFMI with Applications to Automotive Vertical Dynamics Control”. In: *Actuators* 10.11, p. 301. DOI: 10.3390/act10110301.
- Vanfretti, Luigi and Xavier Bombois (2024). “Power System Oscillation Monitoring and Damping Control Redesign under Ambient Conditions and Multiple Operating Points”. In: *IFAC-PapersOnLine* 58.12, pp. 268–273. DOI: 10.1016/j.ifacol.2024.08.583.
- Vanfretti, Luigi, G. M. Jónsdóttir, Muhammad Shoaib Almas, Eldrich Rebello, Seyed Reza Firouzi, and Maxime Baudette (2018). “Audur—A platform for synchrophasor-based power

- system wide-area control system implementation”. In: *SoftwareX* 7, pp. 294–301. DOI: 10.1016/j.softx.2018.08.003.
- Vanfretti, Luigi and Christopher R. Laughman (2024). “Power System Modeling for Identification and Control Applications using Modelica and OpenIPSL”. In: *2024 IEEE Conference on Control Technology and Applications (CCTA)*, pp. 525–532. DOI: 10.1109/CCTA60707.2024.10666594.
- Vanfretti, Luigi, Tin Rabuzin, Maxime Baudette, and Muhammad Murad (2016). “iTesla Power Systems Library (iPSL): A Modelica library for phasor time-domain simulations”. In: *SoftwareX* 5, pp. 84–88. DOI: 10.1016/j.softx.2016.05.001.
- Zhang, Chengwen et al. (2021). “Implementation and Hardware-In-the-Loop Testing of A Wide-Area Damping Controller Based on Measurement-Driven Models”. In: *2021 IEEE Power & Energy Society General Meeting (PESGM)*, pp. 1–5. DOI: 10.1109/PESGM46819.2021.9638055.