# Real-time Simulation and CHiL Testing of Power System Stabilizers on Microcontrollers with Modelica and eFMI

Luigi Vanfretti[1] Christoff Bürger[2] Joseph Pizzimenti[1] Kyle Wilt[1] Hao Chang[1]

[1]Electrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute, USA,
{vanfrl,pizzij,wiltk2,changh7}@rpi.edu
[2]Dassault Systèmes AB, Lund, 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® Standard[2] (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 transition 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 Library[4] (`OpenIPSL`), generated the corresponding eFMI solutions in Dymola, system-integrated them on STM32® microcontrollers with the help of STM32CubeMX[5] and STM32CubeIDE[6], 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 framework[7] 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 Standard[2] (Modelica Association 2024) is an open standard of the Modelica Association®[8]. The official website[2] 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)—are[10] (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 **g**uarded **a**lgorithmic **l**anguage for **e**mbedded **c**ontrol 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 Standard[11], 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/

[9] Originally written by Christoff Bürger.

[10] Parts 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 FMI[12] 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® Premium[13] 26.3.0 and clang-tidy[14] (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).
- **STM32CubeMX[5] 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 STM32CubeH7[15] 1.13.0 software package with hardware abstraction layer (HAL), lower-layer (LL) drivers, and Arm® Cortex®-M Common Microcontroller Software Interface Standard (CMSIS) implementations[16] (cf. Section 5.1).
- **STM32CubeIDE[6] 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).
- **WaveForms[17] 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 Standard[11] 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).

[12] https://fmi-standard.org/

[13] https://www.cppcheck.com/

[14] https://clang.llvm.org/extra/clang-tidy/

[15] https://www.st.com/en/embedded-software/stm32cubeh7.html

[16] STM32CubeH7 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

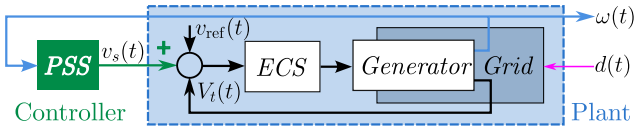



**Figure 1.** Diagram of a power plant's electrical generator equipped with a PSS interconnected with a power grid.

# 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, $v_{\text{ref}}(t)$, 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 subsystem (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, OpenIPSL[4] 3.1.0 is used. OpenIPSL is an open-source Modelica[1] 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 cosimulation, 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:

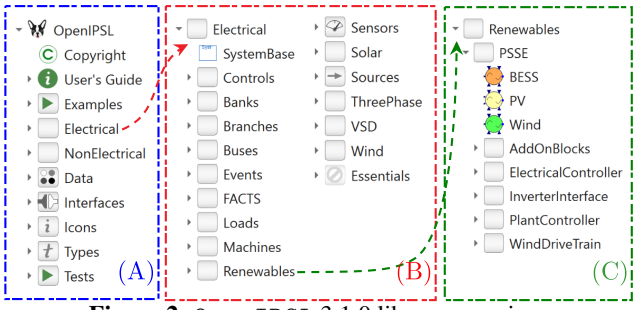



**Figure 2.** `OpenIPSL` 3.1.0 library overview.

- `*.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).

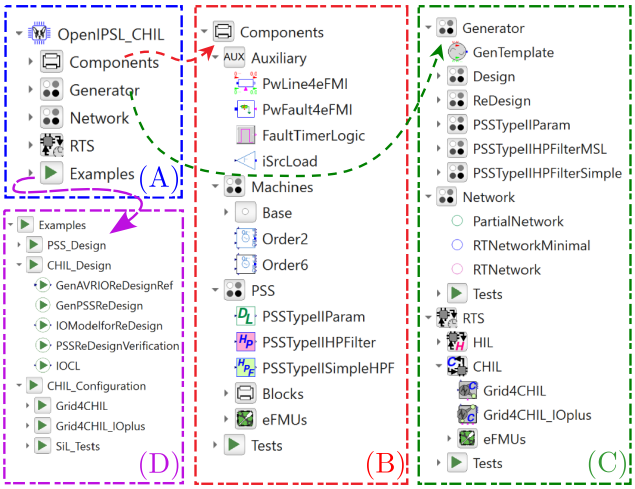



**Figure 3.** `OpenIPSL_CHIL` library overview.

## 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.

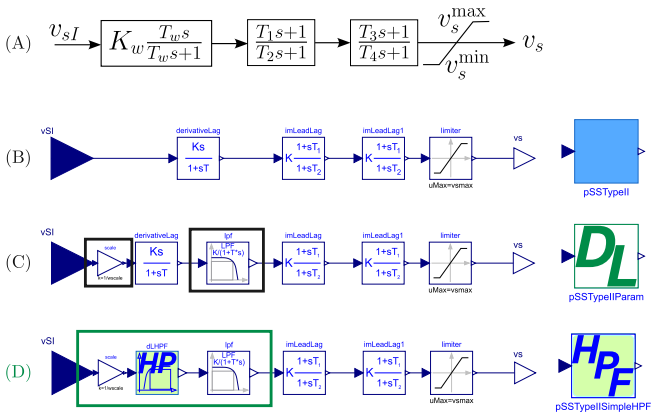



**Figure 4.** PSS variants: (A) Generic block from (Milano 2005), (B) OpenIPSL implementation, (C) Modified OpenIPSL implementation for eFMI export, (D) Final variant with a simplified high pass filter replacing the washout filter.

- `*.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 eFMI 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., $\omega$ 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 `vSI` 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/wscale`, where `wscale=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).

```
block HighPassSimple
  ...
  parameter Real K = Kw*Tw "Gain K = Kw*Tw";
  parameter Real T = Tw    "Time constant T = Tw";
  ...
initial equation
  der(x) = 0;
equation
  der(x) = (u - x)/T;
  y = K/T*(u - x);
end HighPassSimple;
```

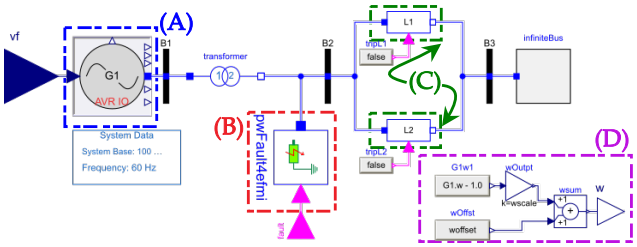



**Figure 5.** The **plant** model: Generator with ECS interconnected with a power grid—`*.RTS.CHIL.Grid4CHIL`, referred to as `GCHIL`.

## 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.

```
// Original OpenIPSL baseMachine:
partial model baseMachine
  ...
  parameter Types.Angle delta0 = CM.arg(
    Vt0 + ((ra + CM.j*xq0) * Z_MBtoSB*I0))
    "Init. val. rotor angle";
  ...
end baseMachine;
```

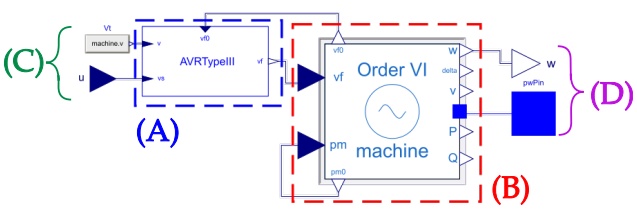



**Figure 6.** The **generator unit** model: Generator with ECS—`*.Generator.ReDesign.GenAVRIOReDesign`.

```
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, $V_t$, 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/lsb. 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, $\Delta w$. (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 $t = [1.5\text{-}1.55]$ 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`.

[19]Note: 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.

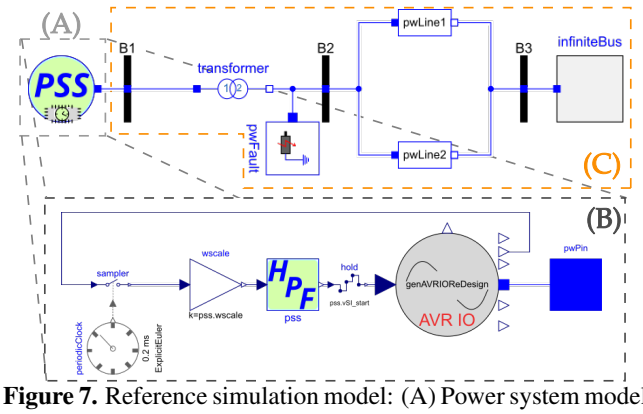



**Figure 7.** Reference simulation model: (A) Power system model instantiation of the generator unit with PSS, (B) Generator unit model with clocked PSS, (C) Power system being interfaced to the generator unit with PSS.

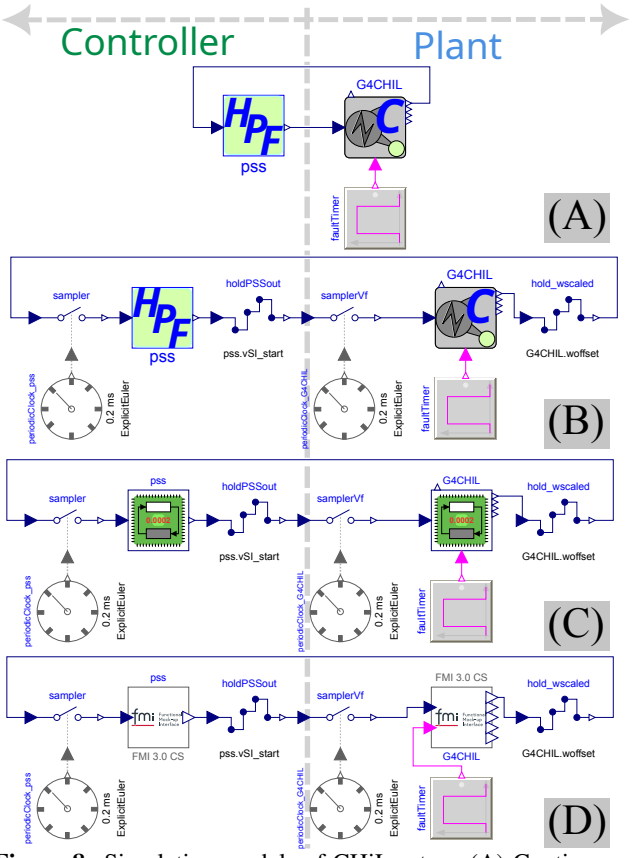



**Figure 8.** Simulation models of CHiL setup: (A) Continuous-time representation, (B) Clocked representation, (C) Clocked with eFMUs, (D) Clocked with source code FMUs based on generated eFMI production code.

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 ($3.0160 \times 10^{-5}$ 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 ($1.6949 \times 10^{-4}$ pu) remains below the $\approx 0.0008$ 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-

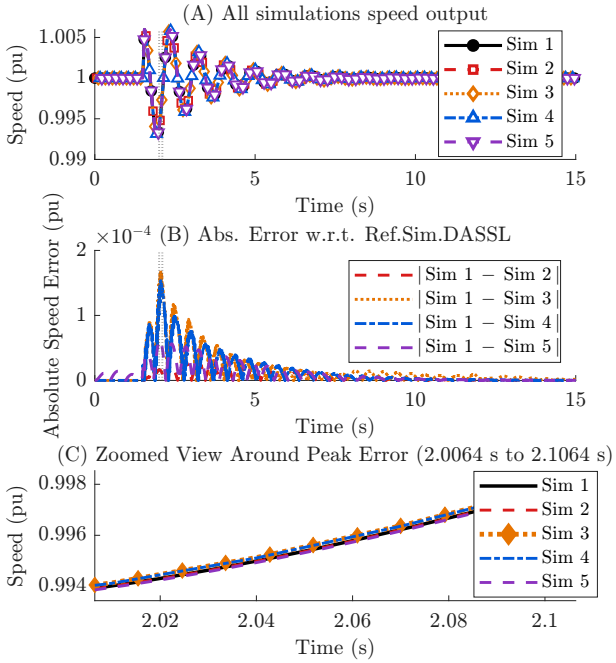



**Figure 9.** MiL Simulations: (A) `w` for all simulations (see Figure 6 (D)), (B) $\forall i \in \{2,3,4,5\}$ : $|$`Ref.Sim.DASSL` − Case $i$ $|$, (C) Zoomed view around peak error.

**Table 1.** Generator speed error relative to `Ref.Sim.DASSL`.

| Case | max $\|\varepsilon\|$ (pu) | RMS $\varepsilon$ (pu) | $t_{\max}$ (s) |
|---|---|---|---|
| 2 | $3.0160 \times 10^{-5}$ | $9.2302 \times 10^{-6}$ | 3.2156 |
| 3 | $1.6949 \times 10^{-4}$ | $3.0662 \times 10^{-5}$ | 2.0564 |
| 4 | $1.5353 \times 10^{-4}$ | $2.7456 \times 10^{-5}$ | 2.0564 |
| 5 | $6.1989 \times 10^{-5}$ | $1.6466 \times 10^{-5}$ | 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 tensorflows of embedded neural networks modeled using the `NeuralNetworks` package of the `eFMI` library, cf. (Kamp et al. 2025)).
- *Software Production Engineering configurations:* Preconfigured 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.

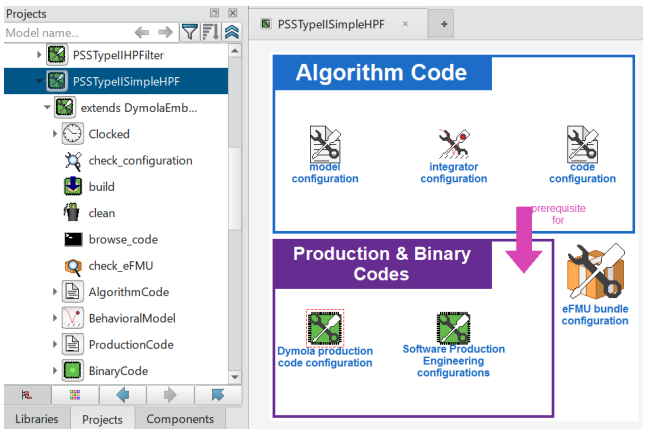



**Figure 10.** `DymolaEmbedded` eFMU configuration for the PSS **controller** of Figure 4 (D). Double-clicking on diagram-view components opens the respective configuration. Calling functions of the *extends*-entry in the package browser executes the respective action, e.g., building GALEC or production 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.

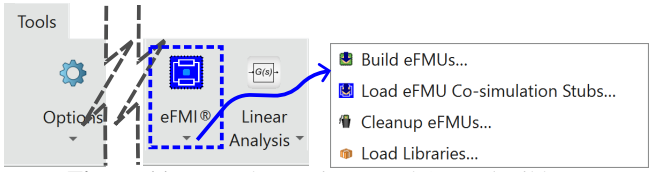



**Figure 11.** *eFMI*-button in Dymola's *Tools*-ribbon.

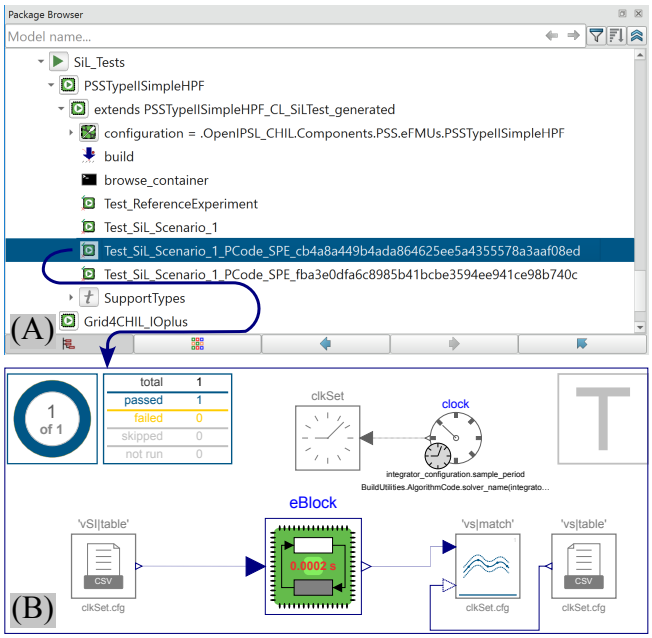



**Figure 12.** (A) SiL experiment package derived from the MiL experiment of Figure 7, (B) SiL test model for the 32-bit floating-point precision production code generated for the PSS **controller** of Figure 4 (D).

## 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 Jenkins[21]), 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 $\approx$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: $\approx$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).

[20] https://www.youtube.com/watch?v=wnk5YbEgTjM

[21] https://www.jenkins.io/

### 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 Standard[11] 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 repository[7]; 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,

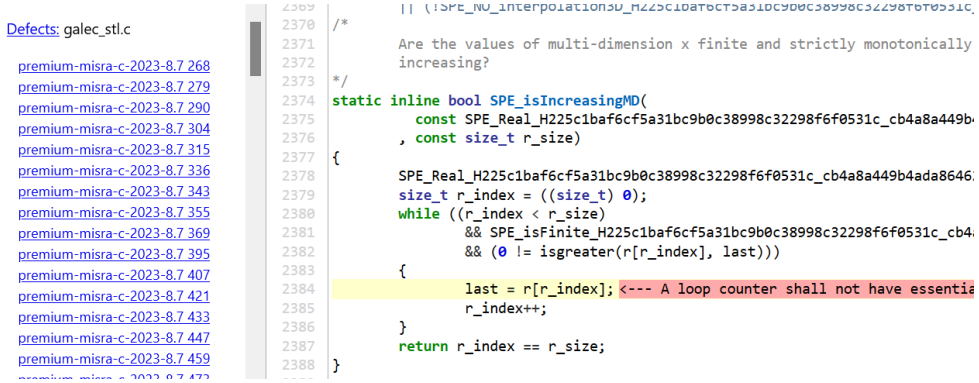



**Figure 13.** Excerpt of the Cppcheck Premium report for the 32-bit floating-point precision production code generated for the PSS **controller** of Figure 4 (D).

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 Pygments[22], 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 FMI[12] 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 managed[23] 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 ($\omega$) 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 potentiometer[24]. 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.

[23] STM32CubeMX proposes a catalog of locally installed and online available STM32CubeH7 versions suited for the target microcontroller.

[24] Although not further discussed for space considerations, the variable-load variant has been successfully deployed and tested.

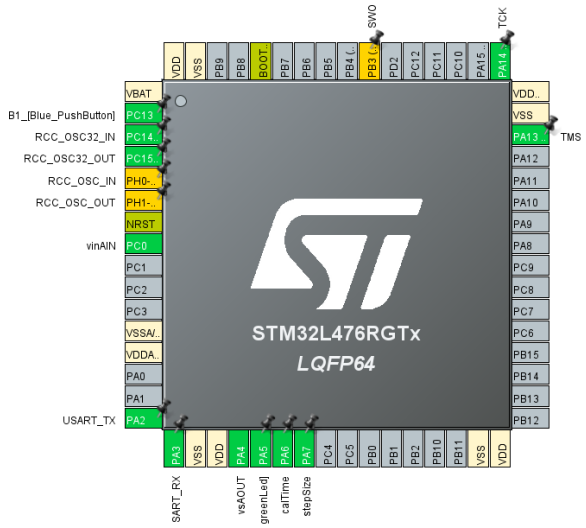



**Figure 14.** STM32CubeMX pin configuration for the NUCLEO-L476RG used as the **controller**.

**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` files per peripheral". Additionally, the target IDE was set to STM32CubeIDE under Project Manager → Project → Toolchain/IDE.

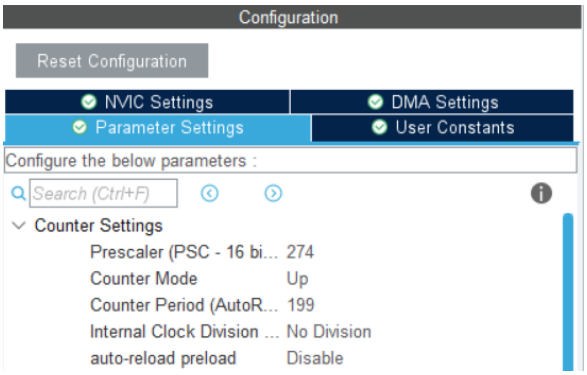



**Figure 15.** SMT32CubeMX timer configuration for the NUCLEO-H723ZG used as the **plant**.

**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:

```
/* USER CODE BEGIN Includes */
#include "block.h"
/* USER CODE END Includes */

/* USER CODE BEGIN PTD */
#define CONCAT_INNER(a, b) a##b
#define CONCAT(a, b) CONCAT_INNER(a, b)

/* Shortcut names for hash-based eFMU API: */
#define MODEL_HASH \
  H225c1baf6cf5a31bc9b0c38998c...

typedef CONCAT(BlockState_, MODEL_HASH) \
  ModelPSS;
typedef CONCAT(SPE_Real_, MODEL_HASH) \
  PSS_Real;
typedef CONCAT(SPE_ErrorSignal_, MODEL_HASH) \
  PSS_ErrorSignal;

#define PSS_NONE_ERRORSIGNAL \
  CONCAT(SPE_ERRORSIGNAL_NONE_, MODEL_HASH)
#define PSS_UNSPECIFIED_ERRORSIGNAL \
  CONCAT(SPE_ERRORSIGNAL_UNSPECIFIED_ERROR_,
      MODEL_HASH)

#define PSS_Startup \
  CONCAT(Startup_, MODEL_HASH)
#define PSS_DoStep \
  CONCAT(DoStep_, MODEL_HASH)
#define PSS_Recalibrate \
  CONCAT(Recalibrate_, MODEL_HASH)
#define PSS_Reinitialize \
  CONCAT(Reinitialize_, MODEL_HASH)
/* 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**:

```
/* USER CODE BEGIN PV */
/* Sampling request counter & PSS on/off button: */
static volatile uint8_t pending_samplings;
static volatile bool use_pss;
/* USER CODE END PV */
```

Note that both variables are `volatile` since they shall be changed by interrupt routines[25]; 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

[25] There 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_ERRORSIGNAL;
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_ERRORSIGNAL;
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_ERRORSIGNAL != 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` $\geq$ 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 FPv4-SP-D16 floating-point unit is used; for the NUCLEO-H723ZG, its FPv5-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 3[26], 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/

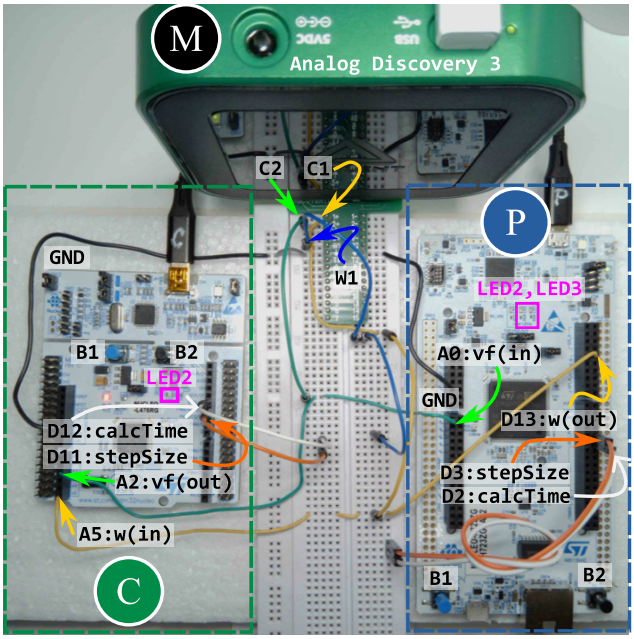



**Figure 16.** CHiL experimental setup wired to record measurement data from experiments using the Analog Discovery 3.

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 WaveForms[17] 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

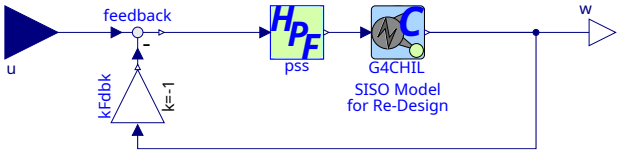



**Figure 17.** Closed-loop model with input and output interfaces to compare with frequency response tests.

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 ($\pm$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

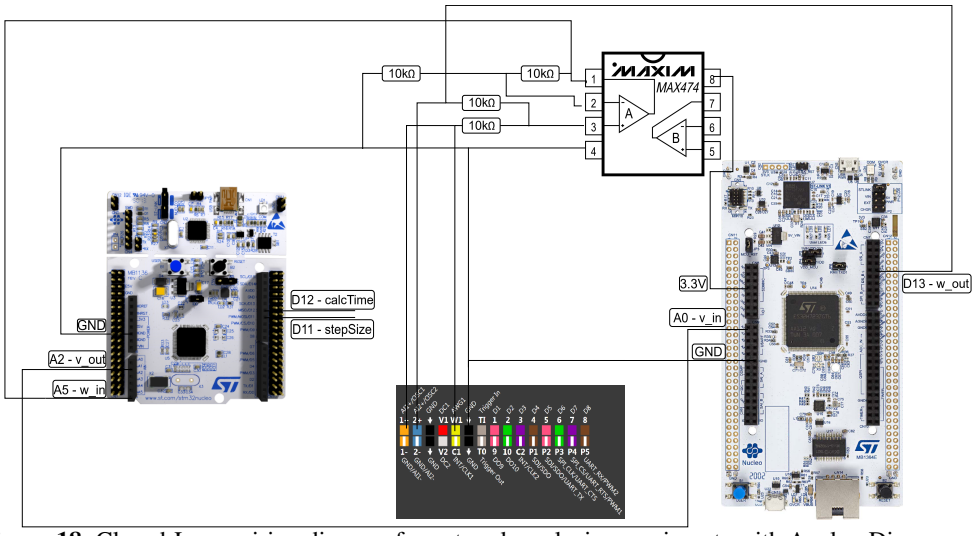



**Figure 18.** Closed-Loop wiring diagram for network analysis experiments with Analog Discovery 3.

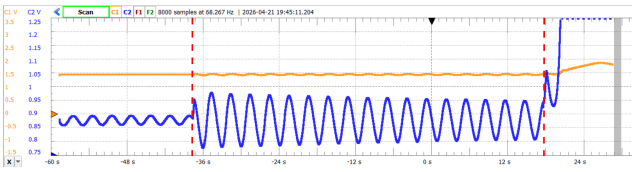


**Figure 19.** Experiment with the PSS disengaged. Orange: PSS output voltage (pu) from the Nucle-L476RG, Blue: scaled speed from the NUCLEO-H723ZG. Red dashed: fault applied.

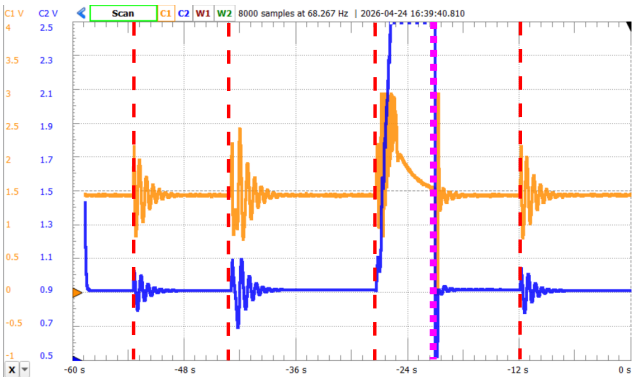



**Figure 20.** Experiment with PSS engaged. Orange: PSS output voltage (pu) from the NUCLEO-L476RG, Blue: scaled speed from the NUCLEO-H723ZG. Red dashed: fault applied. Magenta dotted: reset of both boards.

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 $\pm 0.2$ pu speed deviation, whereas the uncompensated plant became unstable when subjected to deviations greater than $\pm 0.1$ 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

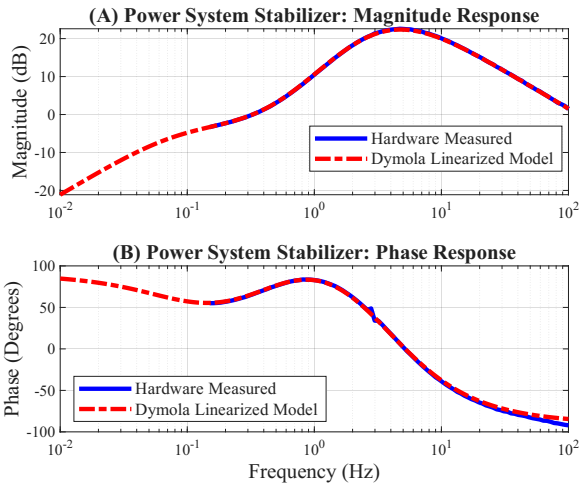



**Figure 21.** Frequency response of the controller. Blue: NUCLEO-L476RG running the eFMU, Red: Dymola linearized model.

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-

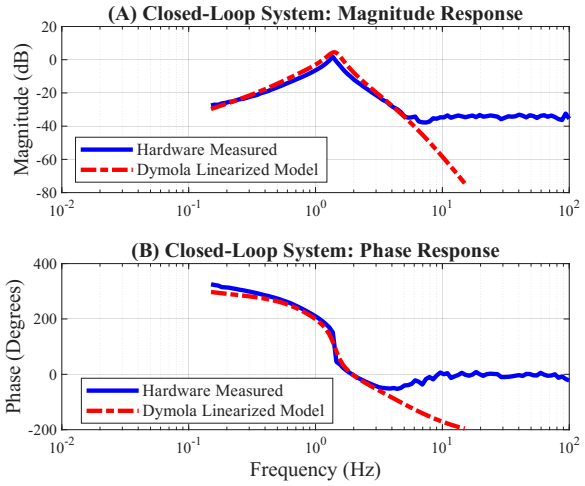



**Figure 22.** Frequency response of the closed-loop test. Blue: hardware measurements, Red: Dymola linearized model.

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

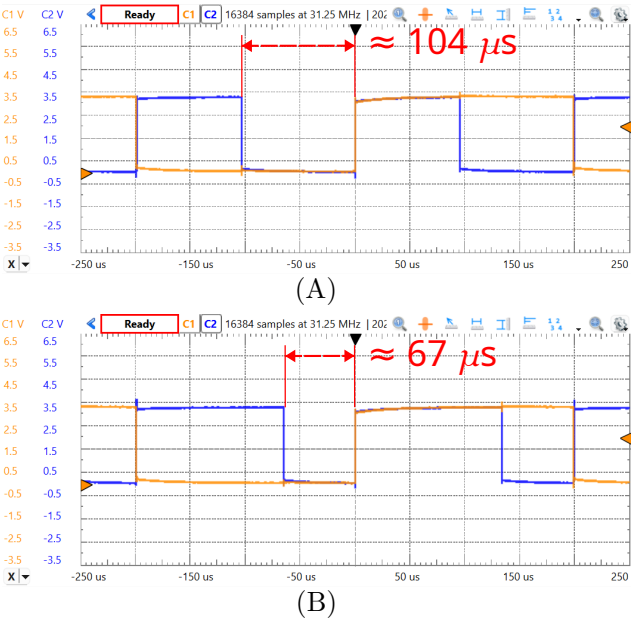



**Figure 23.** Computation time (blue) and sampling timer (orange) of the NUCLEO-H723ZG executing the 32-bit floating-point precision eFMI production code (*Release* build) of the **plant** under (A) steady state, (B) sustained fault injection.

**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 ≈52% headroom in steady state (≈96$\mu s$ computation time) as shown in (A) and ≈33.5% headroom (≈133$\mu s$ 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 (≈239$\mu$ computation time, i.e., ≈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 ≈9% headroom (≈180$\mu$ computation time) in case of sustained fault. We consider ≈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 ≈21% overrun without sustained fault and ≈39.5% overrun with.

For the **controller**, the *Release* build has ≈92.17% headroom (≈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 ≈82.83% (≈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-source[7] 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 Standard[11] 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 **3D**EXPERIENCE® platform[27] 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 AB[13] 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.

[27] https://www.3ds.com/3dexperience/

# 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

[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 Matsson (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 (RTUCON)*, pp. 1–6. DOI: 10.1109/RTUCON62997.2024.10830830.

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 Cordón, 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.

Ultsch, 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.