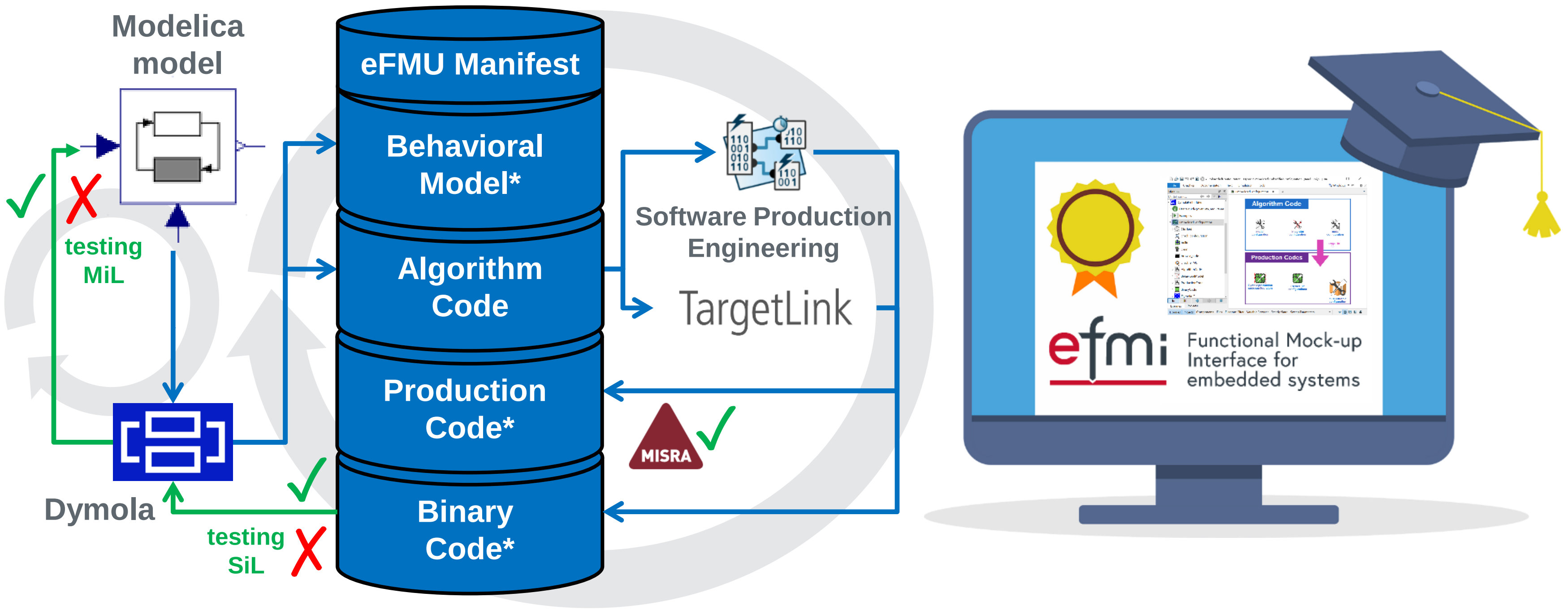


Part 3: Hands-on in Dymola and Software Production Engineering

eFMI®: A beginner's overview and hands-on – 16th International Modelica Conference – 8th of September 2025

Christoff Bürger
Dassault Systèmes
Christoff.Buerger@3ds.com

eFMI® tutorial – Agenda

Part 1: eFMI® motivation and overview (40 min)

Part 2: Running use-case introduction (10 min)

Part 3: Hands-on in Dymola and Software Production Engineering (25 min)

Coffee break (30 min)

Part 3: Hands-on in Dymola and Software Production Engineering (30 min)

Part 4: Advanced demonstrators (20 min)

Part 5 (industry case-study): eFMI based thermal management system (TMS) development for fuel cell electric vehicles (FCEV) (20 min)

Part 6: Outlook and conclusion (5 min)

Tutorial leader:
Christoff Bürger

DASSAULT SYSTEMES

Presenter:
Daeoh Kang

iVH Institute of Vehicle Engineering

eFMI Functional Mock-up Interface for embedded systems

This handout provides a step-by-step guide how to generate and software-in-the-loop (SiL) test an eFMU in Dymola.

Tutorial requirements:

- Own computer with Windows 10 or 11, 64-Bit, x86

You – i.e., every tutorial participant – should have gotten a software bundle with:

- This documentation (`eFMI-Tutorial-Part-3.pdf` in root directory)
- Preinstalled Dymola 2026x Beta 1 (`/Dymola`)
- Preinstalled Software Production Engineering prototype (included in Dymola)
- Workdirectory where eFMUs will be generated and simulation artefacts stored (`/work-directory`)
- Modelica models we actually want to develop; for your reference if something goes wrong (`/reference-models`)
- eFMUs we actually want to build; for your reference if something goes wrong (`/reference-eFMUs`)
- Portable Microsoft Visual C++ and Microsoft Windows SDK required by Dymola (`/portable-MSVC`)
- Portable Java required by Software Production Engineering (`/portable-Java`)
- Portable Cppcheck (`/portable-Cppcheck`) and Python (`/portable-Python`) required for MISRA C:2023 compliance checks of production code
- Licenses of provided software (`/licenses`)

Functional Mock-up Interface for embedded systems

DISCLAIMER

The *Microsoft Visual C++* and *Microsoft Windows SDK* provided in the `/portable-MSVC` directory are subject to licensing of *Microsoft*.

The *Java Development Kit* (*OpenJDK*) provided in the `/portable-Java` directory is subject to licensing of the *Free Software Foundation, Inc*.

The *Python* provided in the `/portable-Python` directory is subject to licensing of the *Python Software Foundation*.

The *Cppcheck* provided in the `/portable-Cppcheck` directory is subject to licensing of *Cppcheck Solutions AB*.

The *Dymola* and *Software Production Engineering* provided in the `/Dymola` directory are subject to licensing of *Dassault Systèmes*.

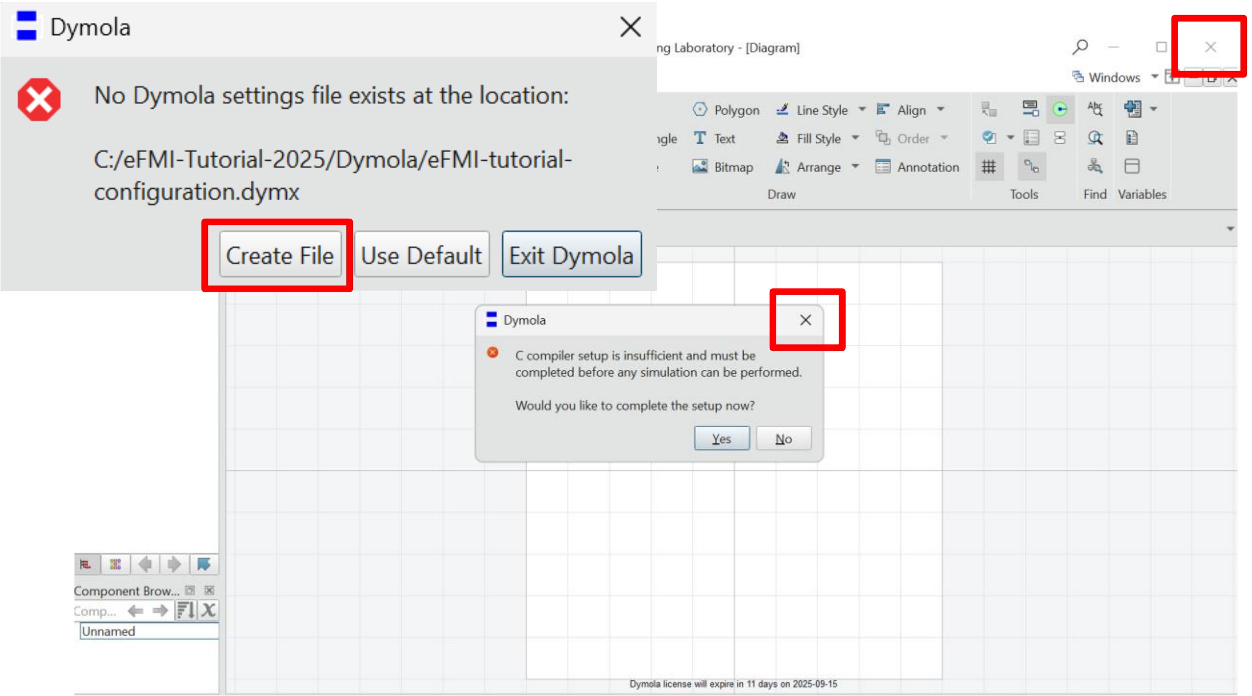
The *Python* libraries and/or scripts *argparse*, *attrs*, *colorama*, *get-pip*, *iniconfig*, *Jinja*, *lxml*, *MarkupSafe*, *packaging*, *pip*, *pluggy*, *Pygments*, *pytest*, *ruff* and *efmpy* are subject to their respective licensing.

BEFORE USING ANY OF ABOVE SOFTWARE, USERS MUST ACCEPT AND AGREE TO THEIR LICENSING

(all licenses can be found in the `/licenses` directory).

THE DISTRIBUTED SOFTWARE IS PROVIDED ONLY FOR USAGE IN THE SCOPE OF THE eFMI® TUTORIAL "*eFMI®: A beginner's overview and hands-on*" OF THE "*16th International Modelica Conference*, 8-10 September 2025"; AND IT IS FOR PARTICIPANTS OF THE TUTORIAL ONLY.

Before getting started, please auto-configure the provided Dymola distribution:



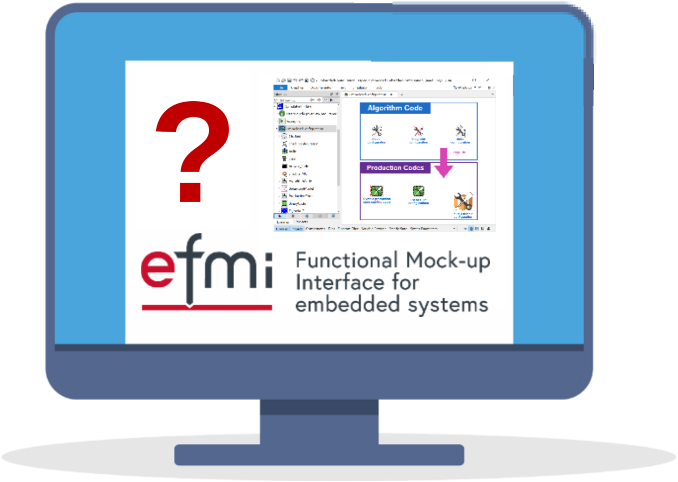
Do the following *once*:

1. Execute `/Dymola/start-Dymola.bat`
2. Create the Dymola settings file
3. Ignore the compiler settings warning (just close the dialog)
4. Immediately close Dymola

From here on, *always* start Dymola via `/Dymola/start-Dymola.bat`.

Feel free to rearrange tabs etc. as you please. Settings are stored. They do *not* interfere with any existing Dymola installation.

Functional Mock-up Interface for embedded systems

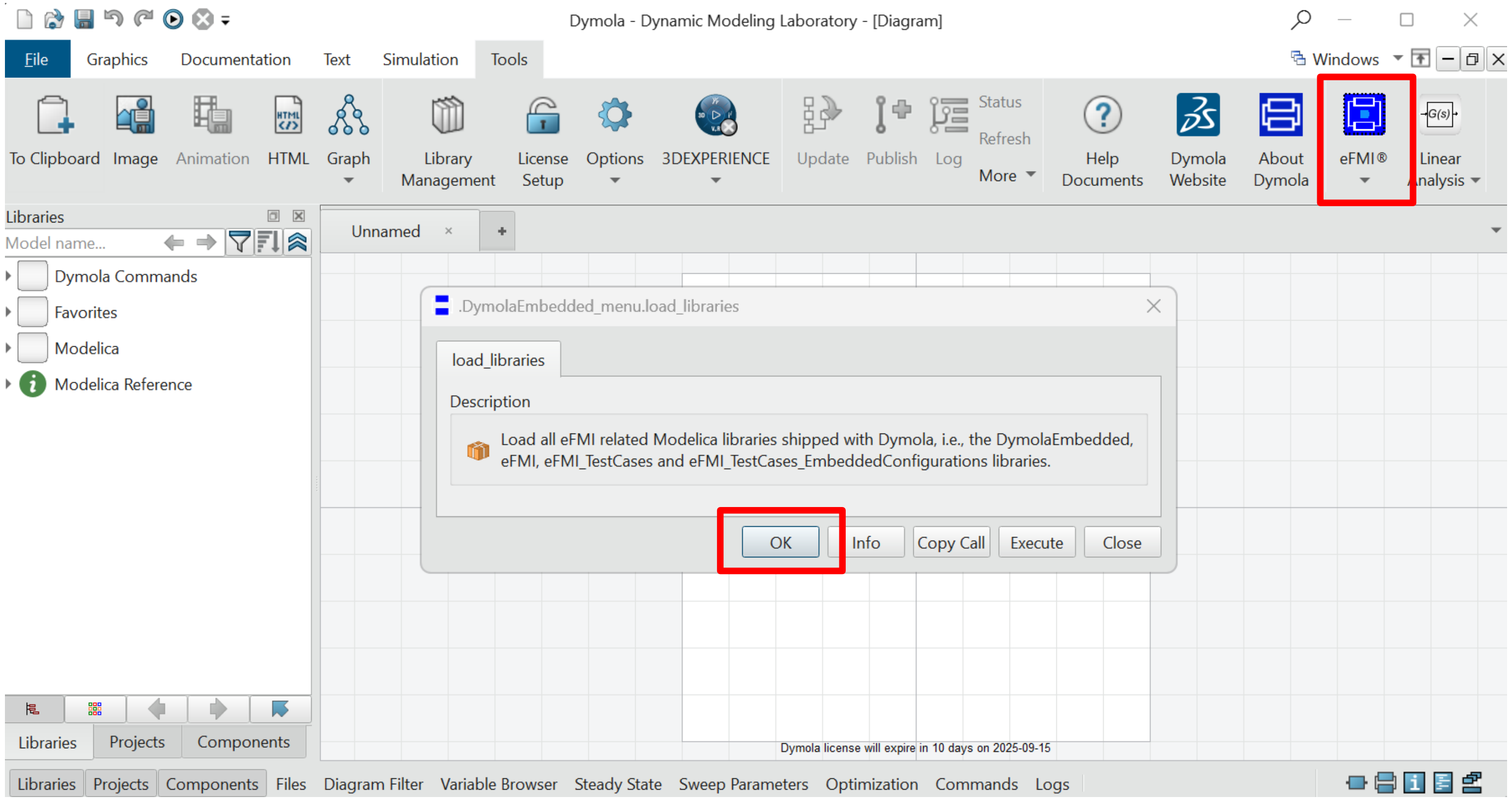


Ok, lets get started!

efmi Functional Mock-up Interface for embedded systems

The user interface for eFMI support in Dymola is provided by means of a Modelica library: `DymolaEmbedded`

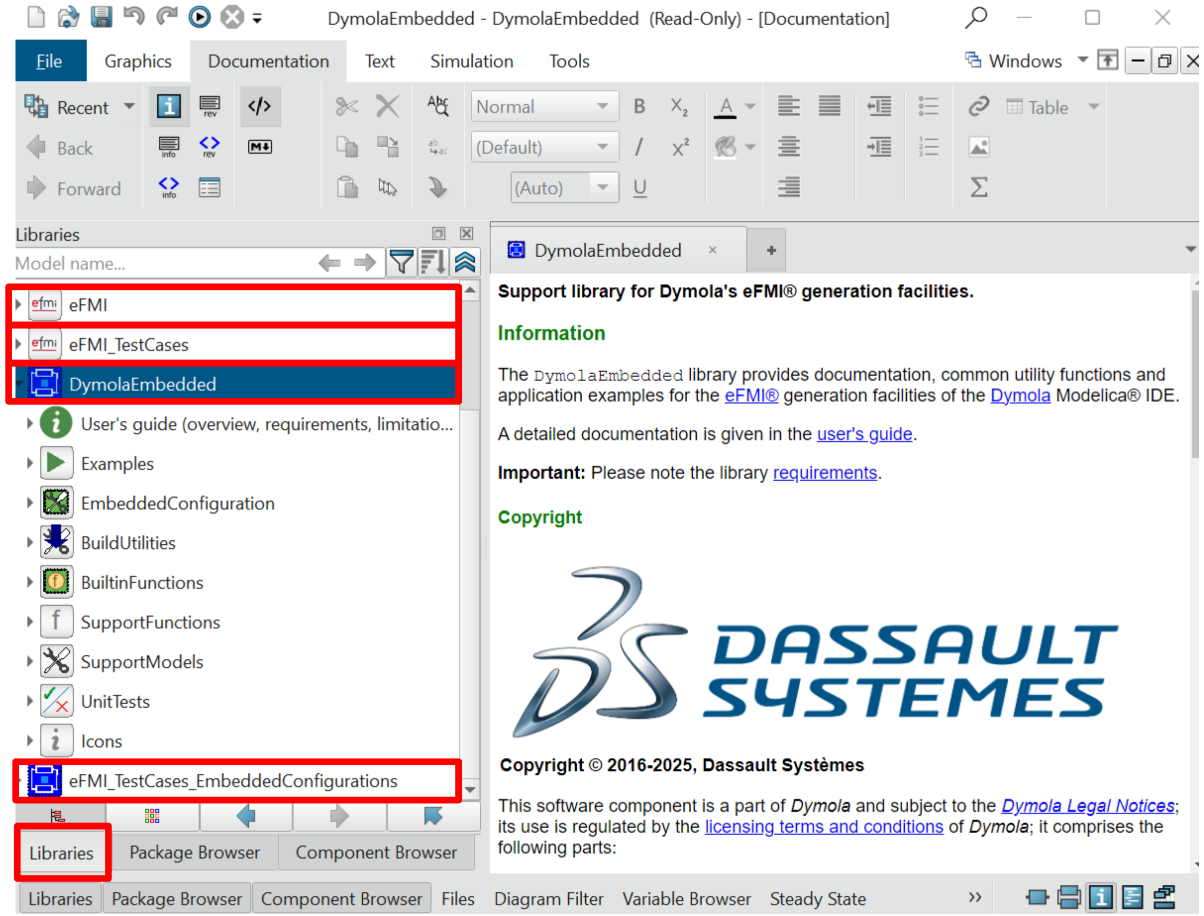
Load `DymolaEmbedded` via the *eFMI* button in the *Tools* ribbon → *Load Libraries…* → *OK*:



Other menu entries permit to build or delete eFMUs for whole package hierarchies and load their co-simulation stubs (this convenience use-cases will become clear throughout the tutorial).

Functional Mock-up Interface for embedded systems

The following libraries are loaded:



`eFMI`:

- Support library to ease adaptation of existing Modelica models for eFMI (mostly about MSL → eFMI table adapters)
- Public domain, © MA, MAP eFMI

`eFMI_TestCases`:

- eFMI application examples used for official cross-checks of eFMI tooling; Modelica tooling agnostic
- Public domain, © MA, MAP eFMI
- Contains our running use-case, M04

`DymolaEmbedded`:

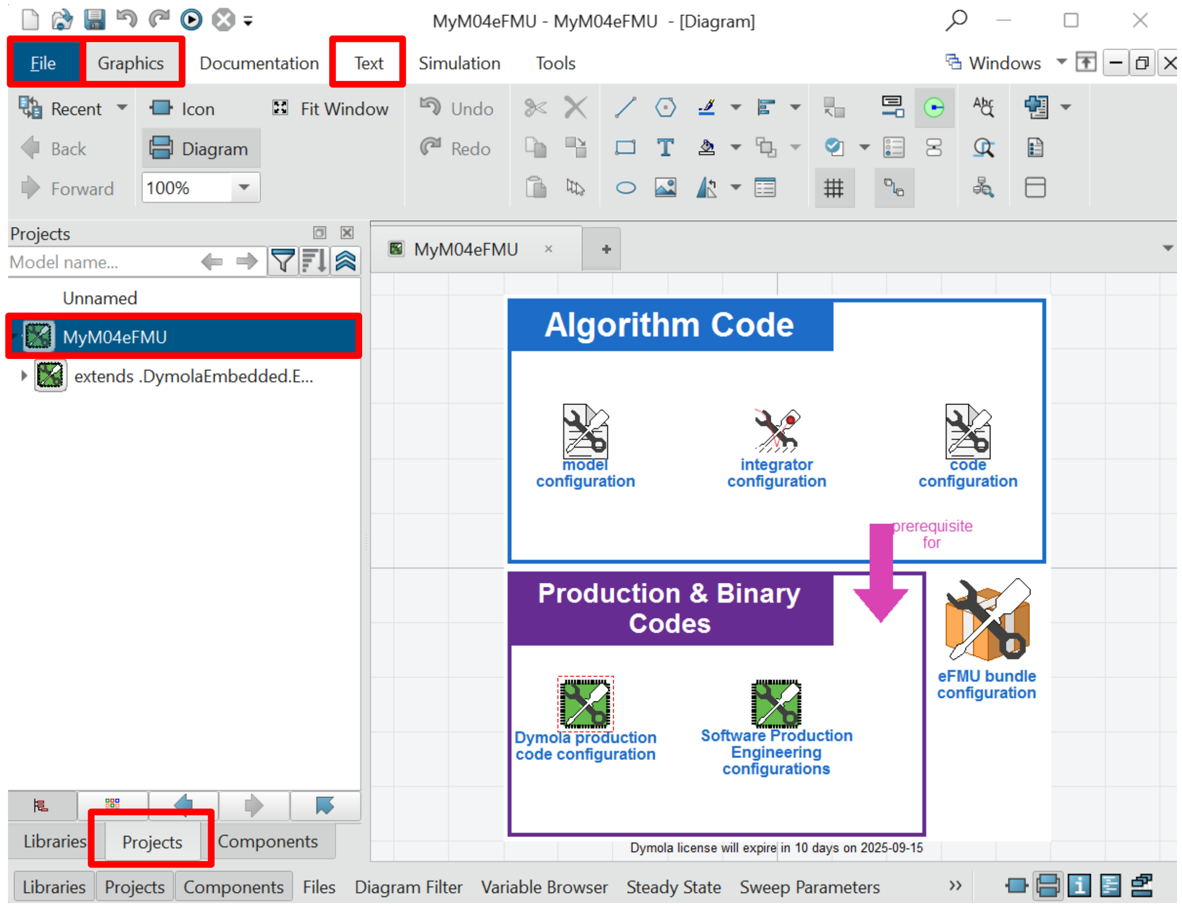
- Interface for Dymola's eFMI facilities
- Provides means to configure eFMU generation & generate various eFMI containers

`eFMI_TestCases_EmbeddedConfigurations`:

- eFMU generation configurations for `eFMI_TestCases`
- Already contains a configuration for M04 (we will develop from scratch in the following)

Functional Mock-up Interface for embedded systems

Create a new eFMU generation configuration for the M04 controller:



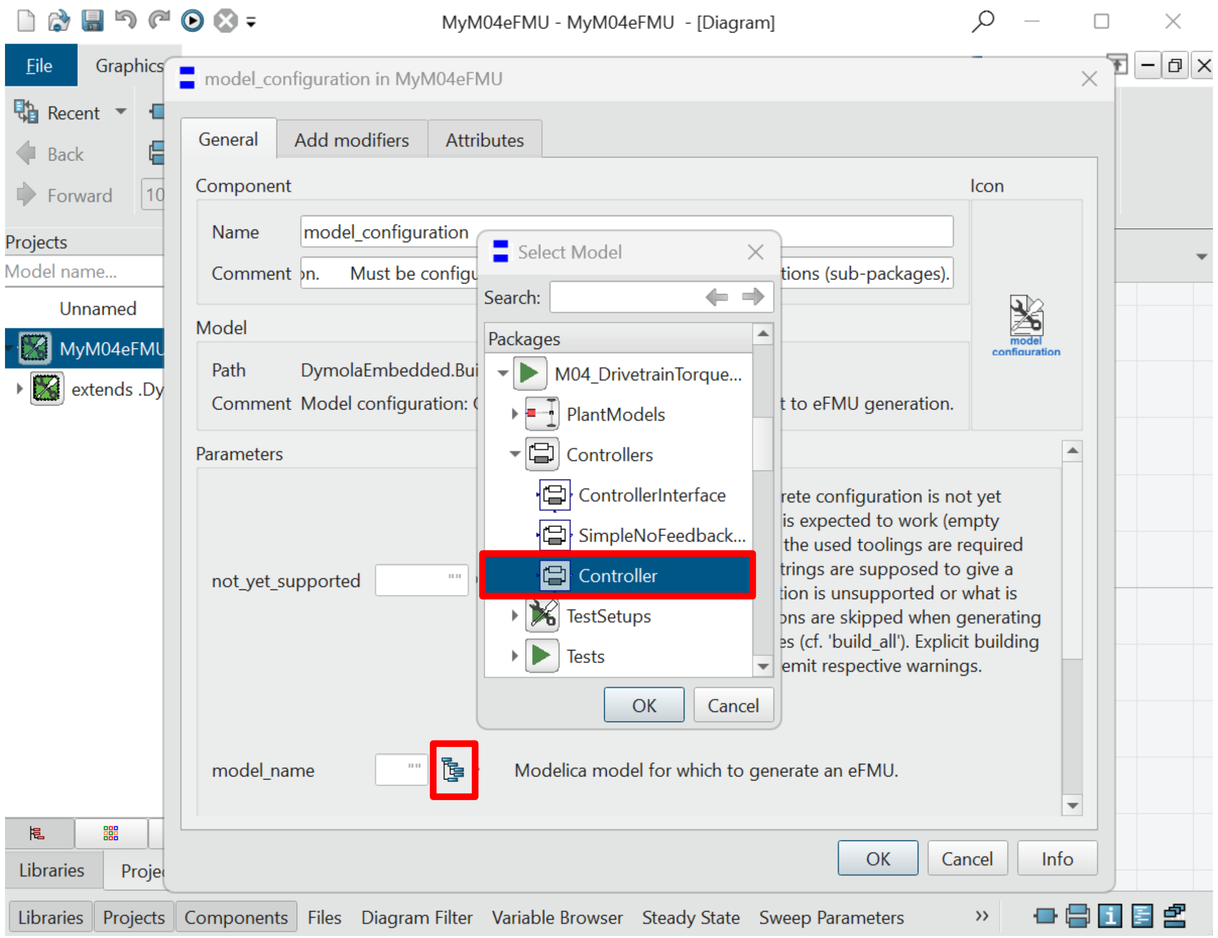
Create package extending `EmbeddedConfiguration`:

1. *File → New → Package*, Name: `MyM04eFMU`
2. New package visible in *Package Browser* & *Projects* (not *Libraries*)
3. Double click `MyM04eFMU`; switch to *Text* ribbon
4. Add **`extends`** `.DymolaEmbedded.EmbeddedConfiguration;`
5. Switch to *Graphics* ribbon

Dymola and Software Production Engineering eFMU code generation can be configured from the diagram layer of `MyM04eFMU.`; it is an eFMU generation configuration.

Functional Mock-up Interface for embedded systems

Create a new eFMU generation configuration for the M04 controller:

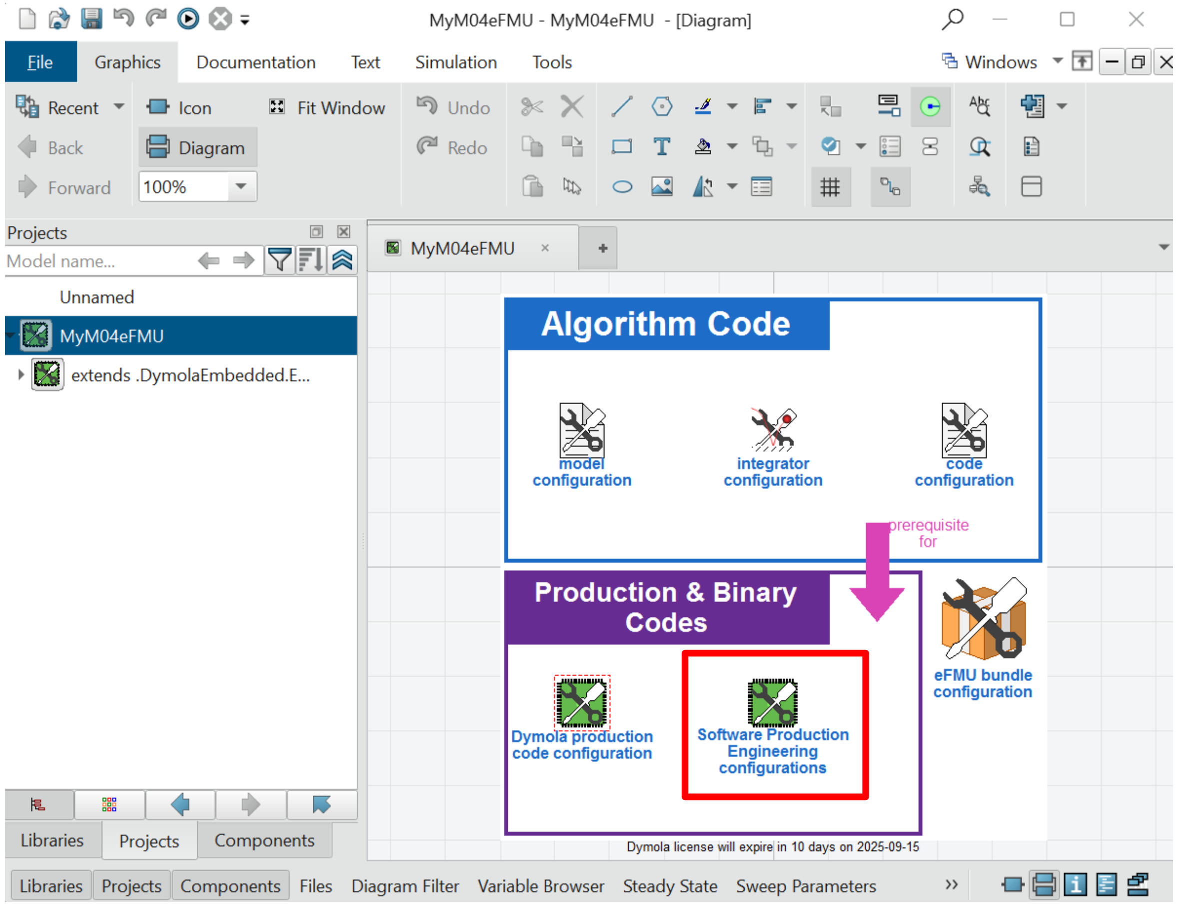


Configure Dymola's GALEC code generation:

1. Double click *model configuration*
   → *model_name*
   → *Edit* (package tree icon)
   → select `eFMI_TestCases`
   `.M04_DrivetrainTorqueControl`
   `.Controllers.Controller`
   → *OK*
   → *OK*
2. Double click *code configuration*
   → *obfuscate*: `None`
   → *OK*
3. Double click *integrator configuration*
   → *sample_period*: `5.0e-4`
   → *solver_method*: `Explicit Euler`
   → *OK*

Functional Mock-up Interface for embedded systems

Create a new eFMU generation configuration for the M04 controller:



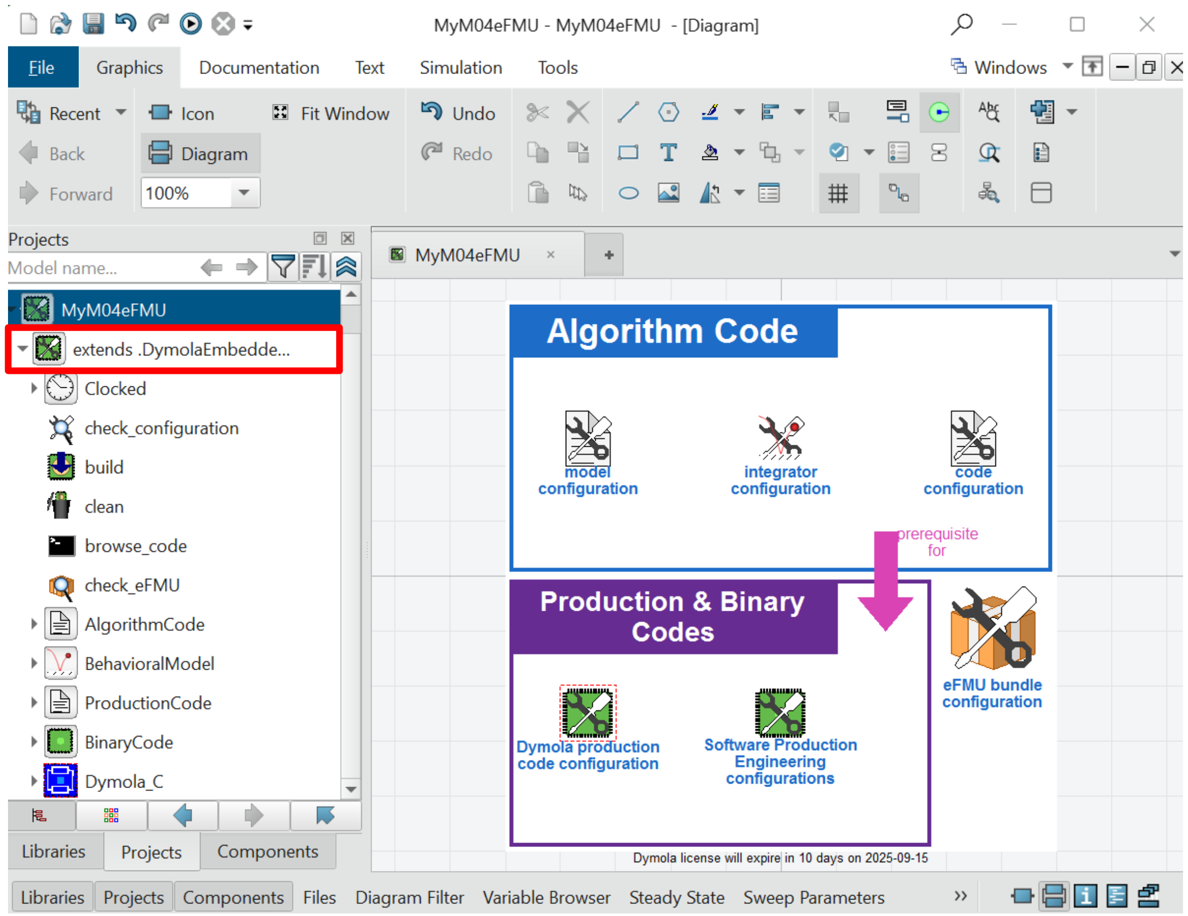
Software Production Engineering is already default configured:

- 32-Bit and 64-Bit floating-point precision production codes
- 32-Bit and 64-Bit x86 ISA binary codes (self-contained static linked libraries)

⇒ 2 Production Code & 4 Binary Code containers

Functional Mock-up Interface for embedded systems

Investigate the eFMU generation configuration `MyM04eFMU` for the M04 controller:

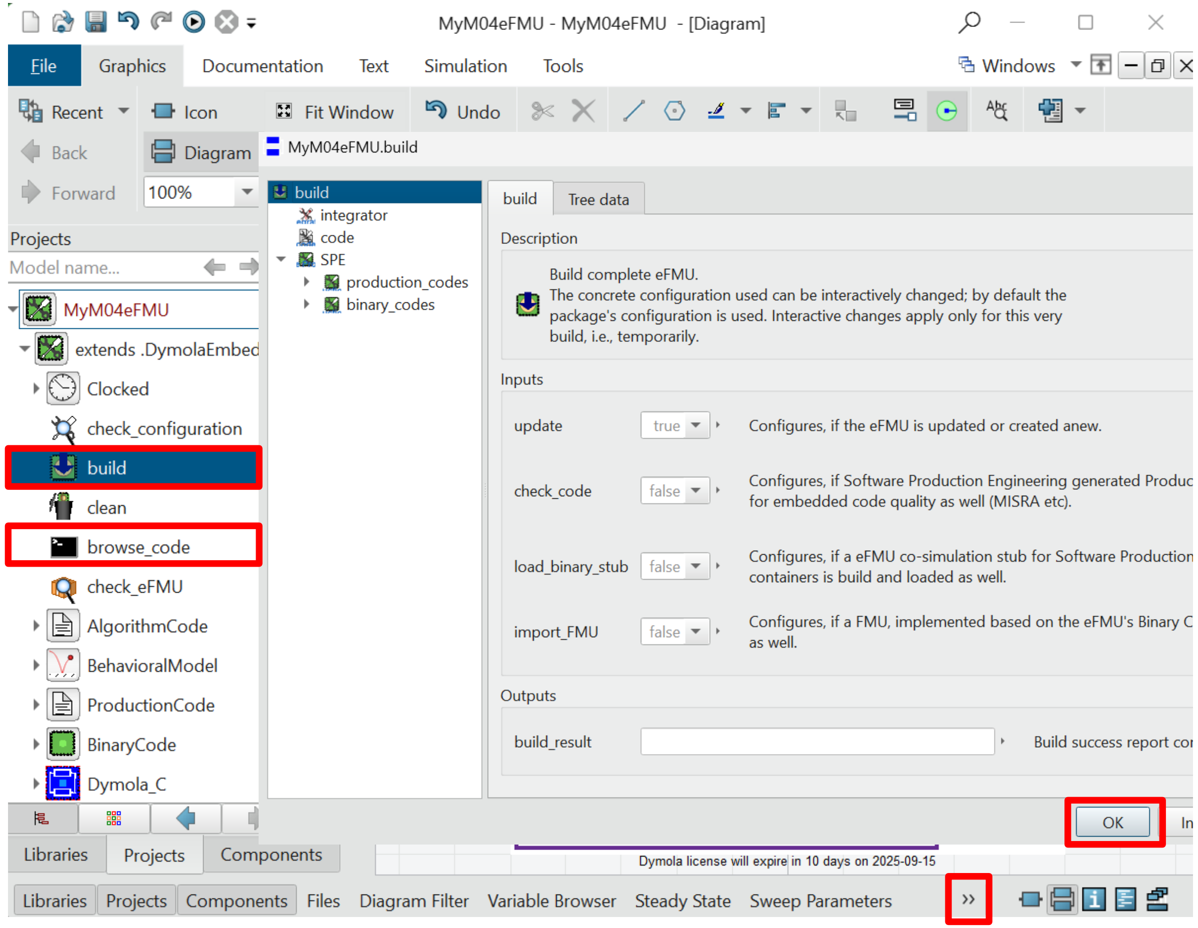


All eFMU build activities are inherited from `DymolaEmbedded.EmbeddedConfiguration`:

- Available via the *extends* entry in the *Package Browser & Libraries* / *Projects* view (depending if configuration is write protected or not)
- Preconfigured with eFMU generation configuration
- Activities grouped according to eFMI container type:
  - **Algorithm Code:** Generate GALEC code
  - **Behavioral Model:** Derive experiment packages to configure test scenarios & tolerances; use experiment packages to generate respective Behavioral Models
  - **Production Code:** Generate & MISRA C:2023 check Software Production Engineering code
  - **Binary Code:** Generate Software Production Engineering binaries & Modelica proxies for co-simulating such; export FMU

Functional Mock-up Interface for embedded systems

Generate the eFMU configured in `MyM04eFMU` for the M04 controller:



Build the eFMU with Algorithm Code, 2x Production Code and 4x Binary Code containers:

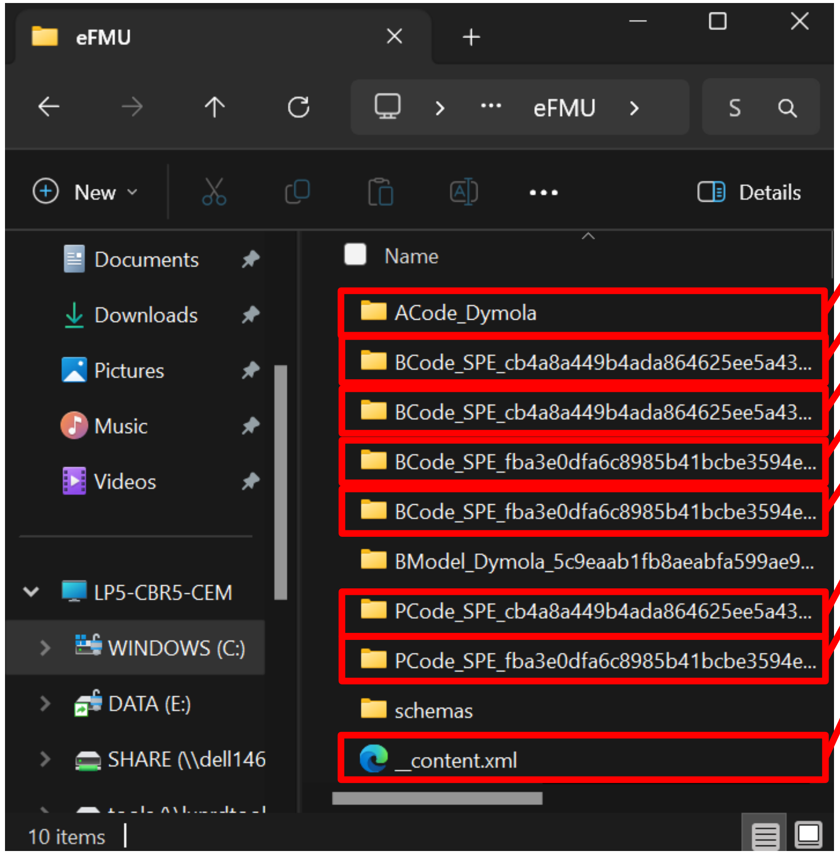
1. Right click `MyM04eFMU.build` in the *Package Browser / Projects* view
   → *Call Function…*
   → *OK*
2. You can check the build log in the *Commands* window

Browse the generated eFMU:

1. Right click `MyM04eFMU.browse_code` in the *Package Browser / Projects* view
   → *Call Function…*
   → *OK*

Functional Mock-up Interface for embedded systems

Investigate the generated eFMU (`MyM04eFMU/eFMU`):



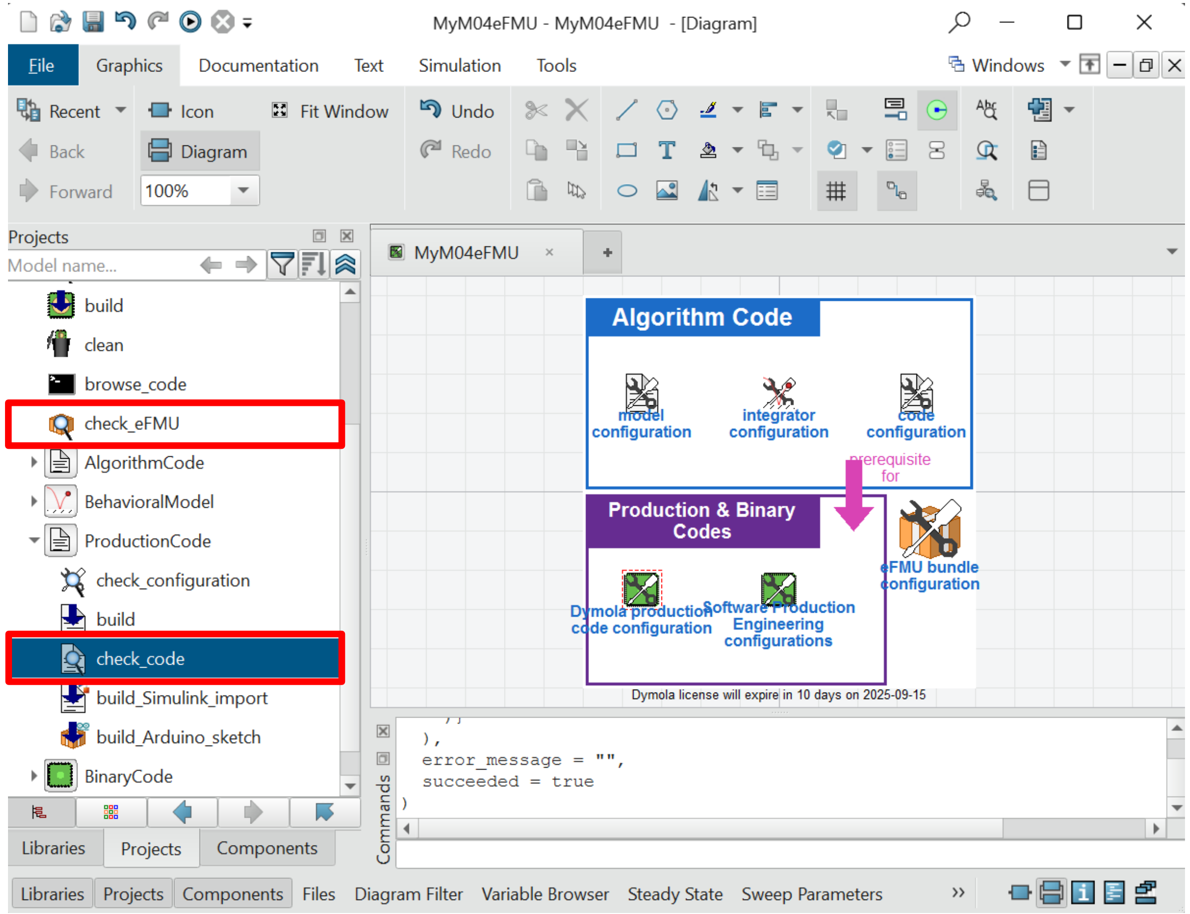
Contained containers:

- **Algorithm Code** with GALEC code
- x86, 32-Bit floating-point precision **Binary Code**
- x64, 32-Bit floating-point precision **Binary Code**
- x86, 64-Bit floating-point precision **Binary Code**
- x64, 64-Bit floating-point precision **Binary Code**
- 32-Bit floating-point precision **Production Code**
- 64-Bit floating-point precision **Production Code**
- **Content manifest** listing all containers

Take some time to investigate the eFMU, e.g.:

- How cross references between manifests work
- Quality of generated GALEC code (self-contained / inlined, error handling of symbolic optimized linear equation systems, local vs. global variables etc)
- …

Check the eFMU and its production codes:



Check MISRA C:2023 compliance of all production codes via Cppcheck:

1. Right click `MyM04eFMU.ProductionCode.check_code` in *Package Browser / Projects* view
   → *Call Function…*
   → *OK*
2. Analyses reports for each production code are provided in your webbrowser (note, that `block.c`, the actual production code, satisfies MISRA)

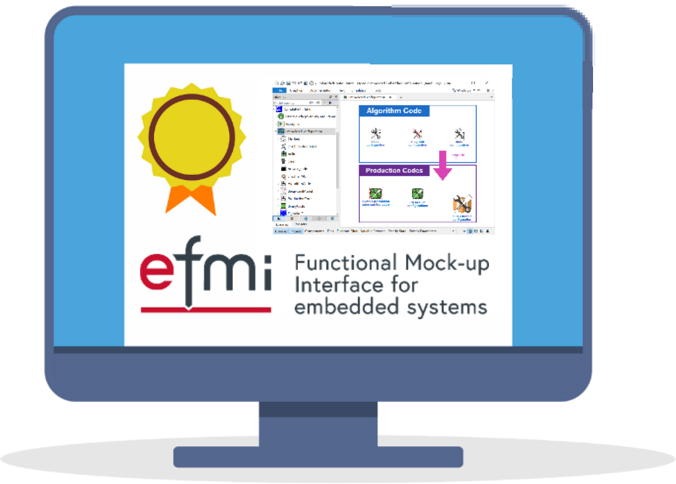
Check eFMU with *eFMI Container Manager* and *eFMI Compliance Checker* (MAP eFMI released tools):

1. Right click `MyM04eFMU.check_eFMU` in the *Package Browser / Projects* view
   → *Call Function…*
   → *OK*

Functional Mock-up Interface for embedded systems

Congratulations, you are halfway through!

See you in the second half of the hands-on after the coffee break!

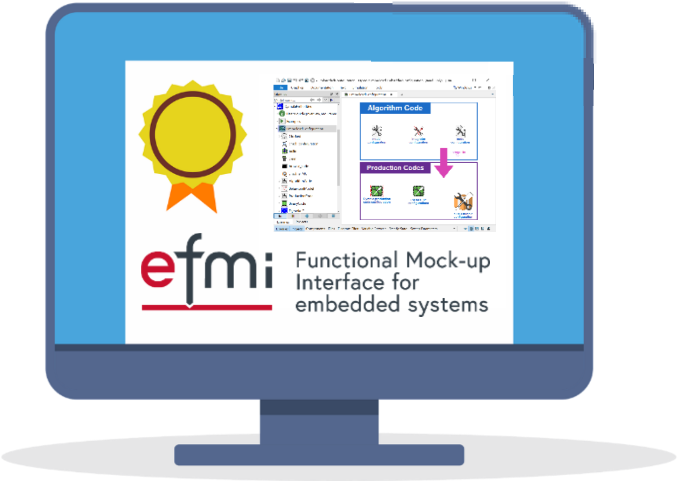


eFMU generation done.

Let’s go on to Behavioral Models & software-in-the-loop (SiL) simulation.

Congratulations, you are halfway through!

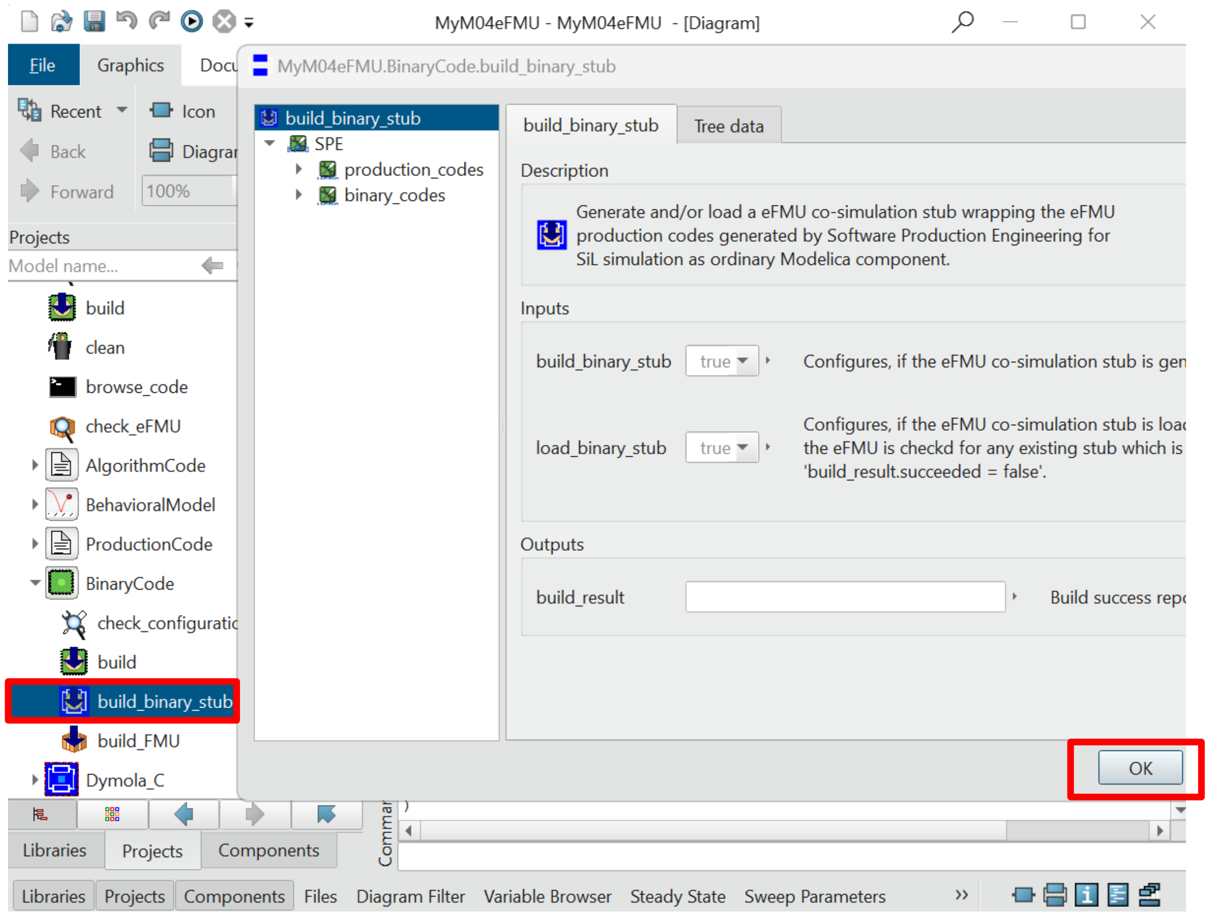
Welcome back to the second half of the hands-on!



eFMU generation done.

Let's go on to Behavioral Models & software-in-the-loop (SiL) simulation.

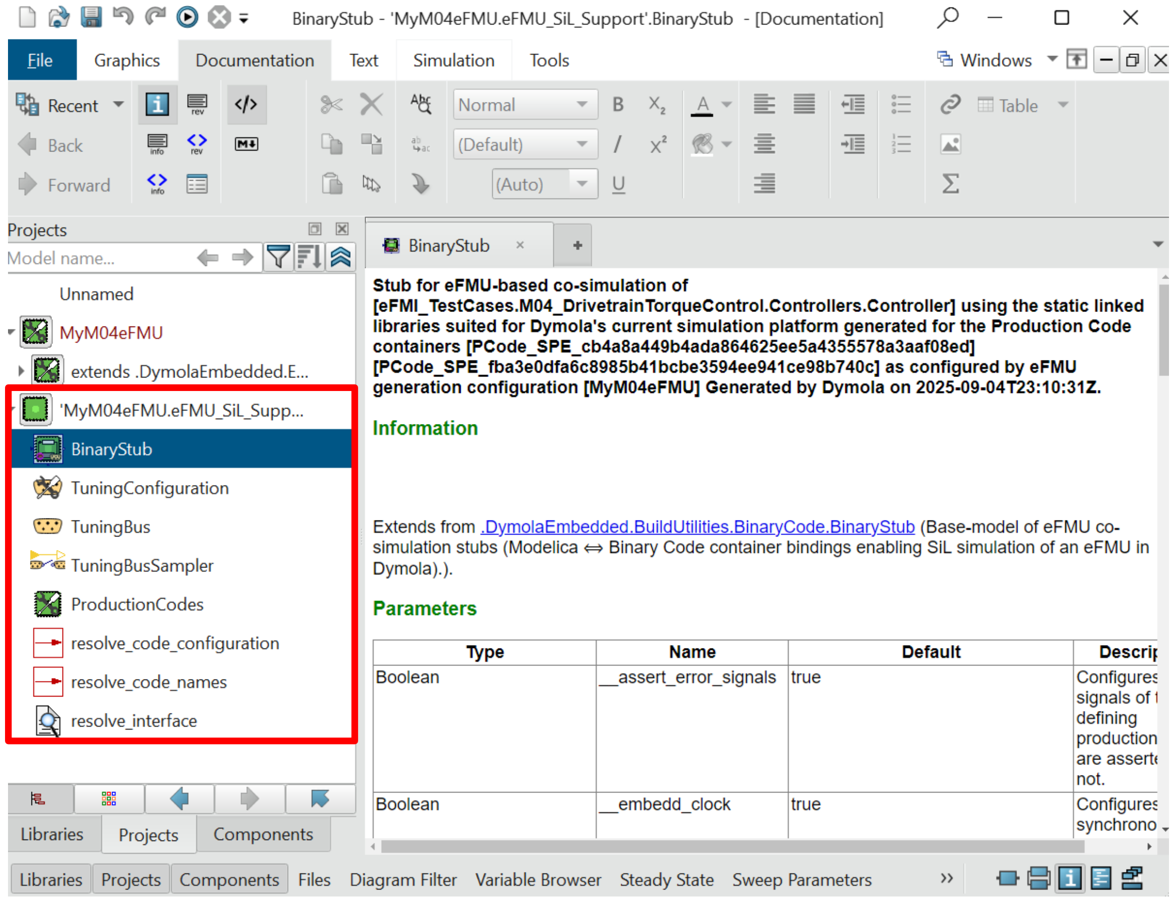
Generate eFMU co-simulation stub:



1. Right click `MyM04eFMU.BinaryCode.build_binary_stub` in *Package Browser / Projects* view
   → *Call Function…*
   → *OK*

A new package `'MyM04eFMU.eFMU_SiL_Support'` is generated. Its `BinaryStub` model is a Modelica proxy to the static linked libraries – and therefore production codes – generated by Software Production Engineering.

Investigate generated eFMU co-simulation stub:

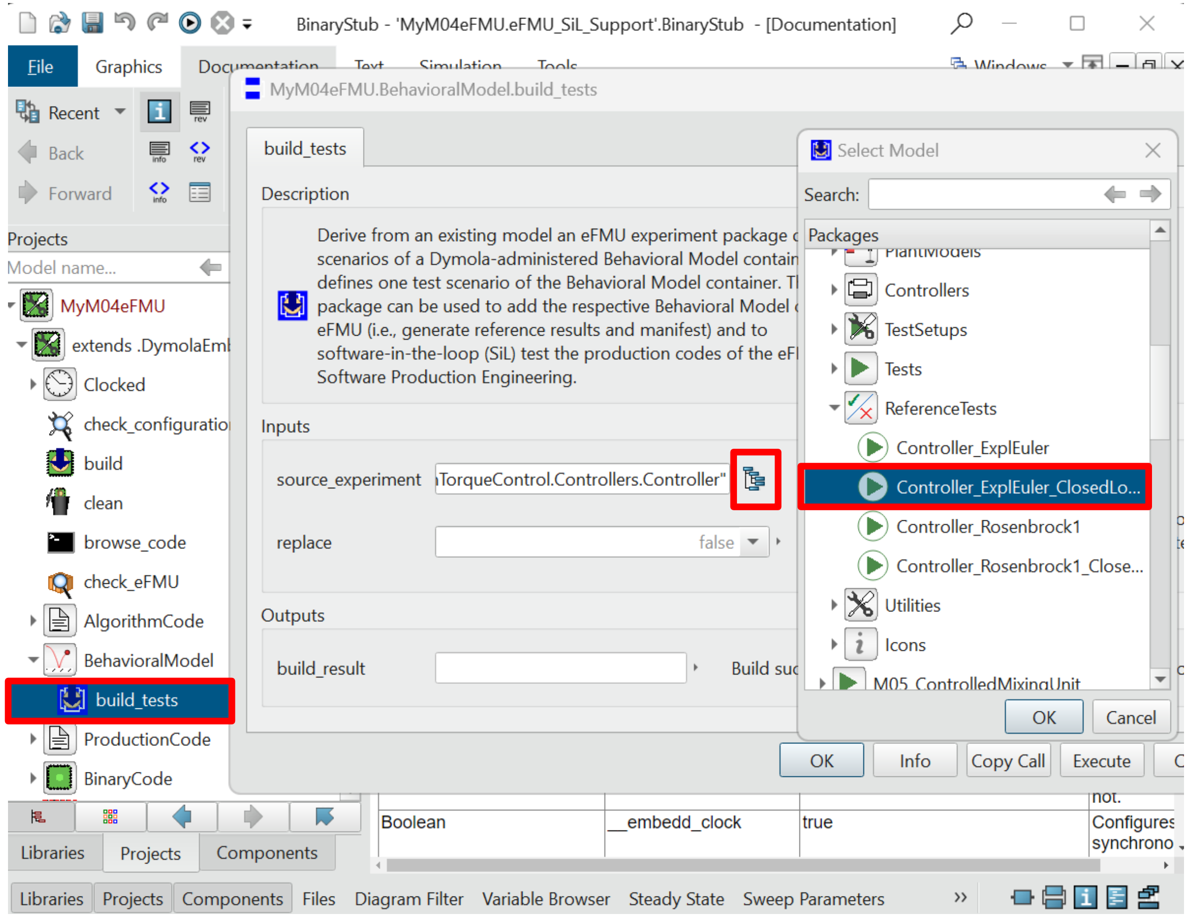


Main characteristics of eFMU co-simulation stubs:

- Support multiple instantiation (each is atomic)
- All production codes available (32-Bit & 64-Bit floating-point precision simulation)
- Support modification, input-dependent initialization, recalibration & reinitialization
- Provide & assert eFMI error signals
- Preserve original model interface (dimensionalities, diagramatic layout of in- & output connectors etc)
- Provide sampling with period of generated eFMU
- ”Just” a production code proxy (no additional equations; no solver required; ”simply” implement GALEC block life-cycle)

Functional Mock-up Interface for embedded systems

Derive experiment package to define test scenarios & generate Behavioral Model container:

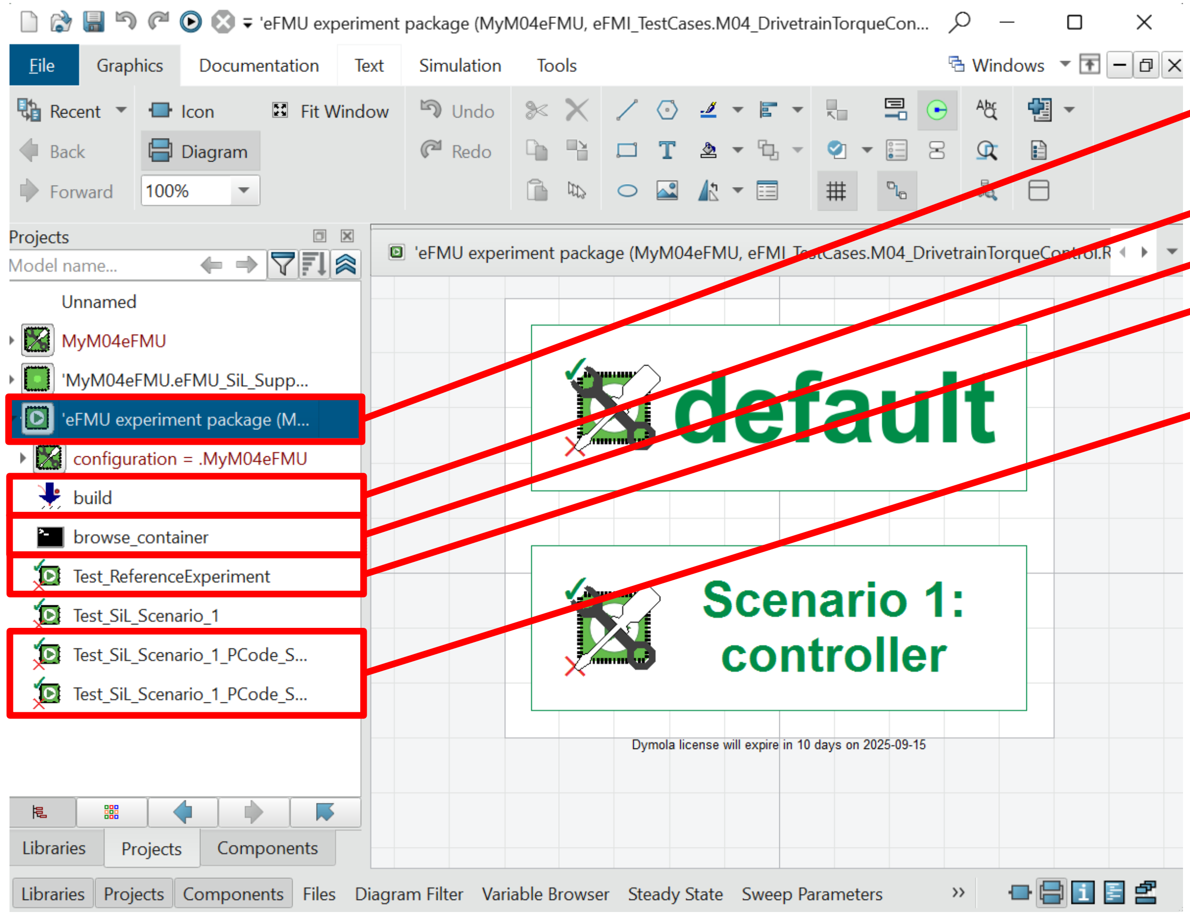


Derive experiment package from existing closed loop experiment:

1. Right click `MyM04eFMU.BehavioralModel.build_tests` in *Package Browser* / *Projects* view
   → *Call Function…*
   → *source_experiment*
   → *Edit* (package tree icon)
   → select `eFMI_TestCases.M04_DrivetrainTorqueControl.ReferenceTests.Controller_ExplEuler_ClosedLoop`
   → *OK*
   → *OK*

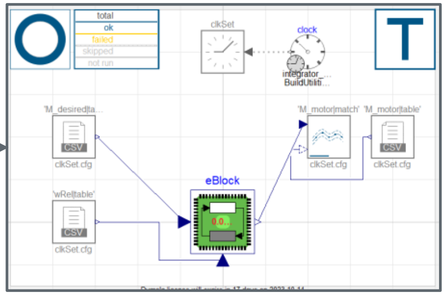
Functional Mock-up Interface for embedded systems

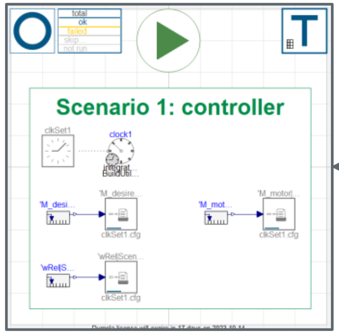
Investigate the derived experiment package:



The generated experiment package contains:

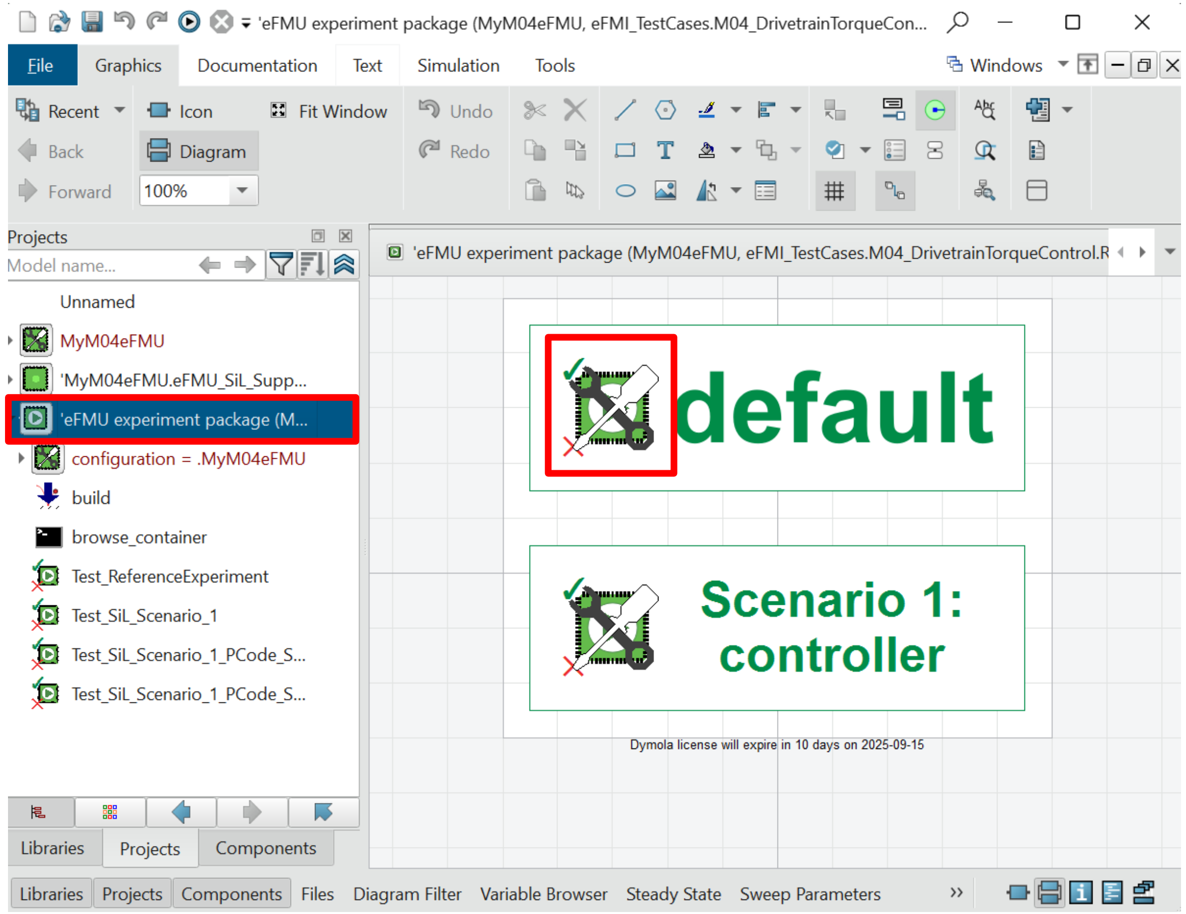
- Records to define absolute & relative tolerances for test scenarios
- Function to generate the Behavioral Model container
- Function to browse the Behavioral Model container
- A single reference experiment to regression test the source experiment and generate reference results
- SiL tests for each production code; one test for each "controller" instance (i.e., test scenario) in the source experiment





Functional Mock-up Interface for embedded systems

Define tolerances for the test scenarios of the experiment package:



Define absolute and relative tolerances for all floating-point precisions and test scenarios (i.e., SiL tests). We can use a default for all scenarios (here only a single):

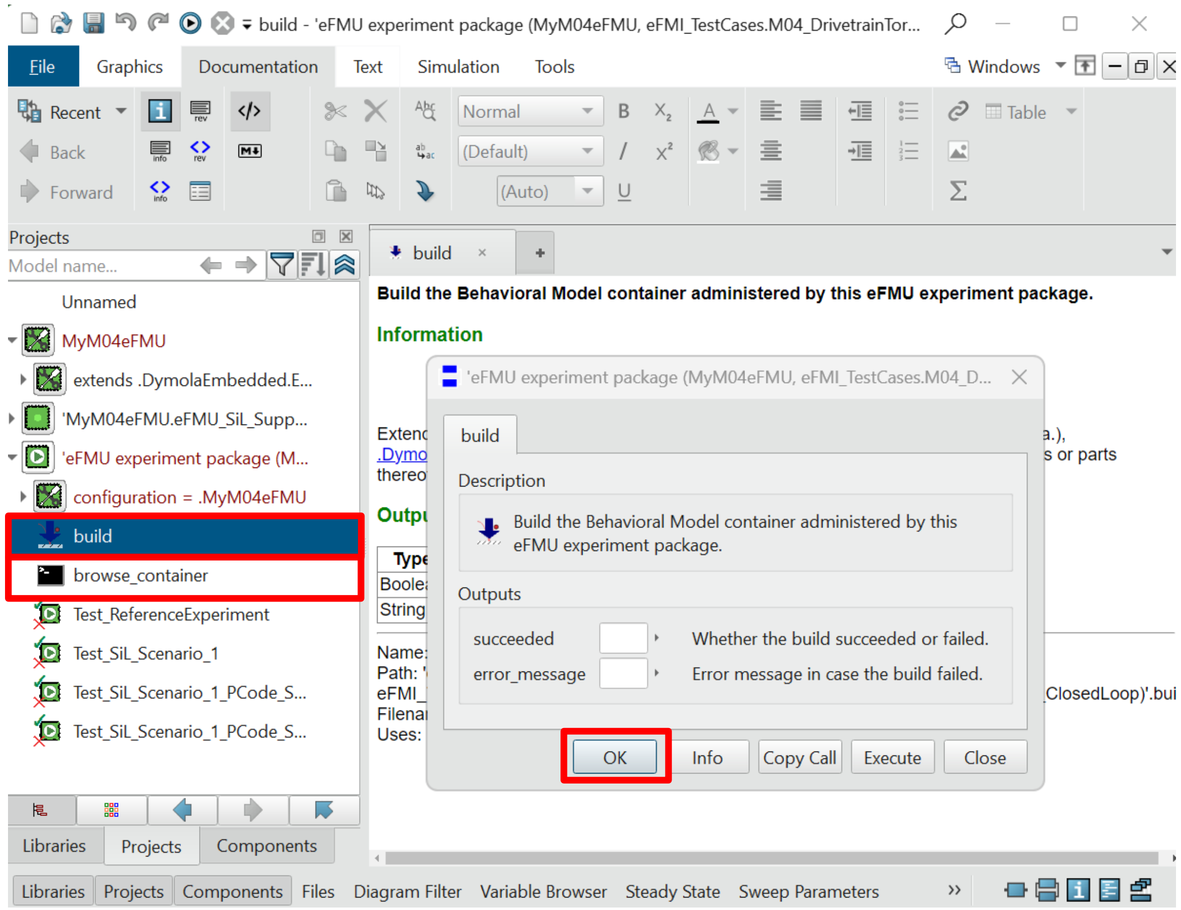
1. Double click `tolerances_default` (labeld **default**) in *Diagram* view of the experiment package
   → set tolerances for `M_motor` output a follows

   ```
   absolute_x32(M_motor=1.0e-3)
   relative_x32(M_motor=1.0e-4)
   absolute_x64(M_motor=1.0e-6)
   relative_x64(M_motor=1.0e-8)
   ```

   → *OK*

Functional Mock-up Interface for embedded systems

Generate Behavioral Model container form the experiment package:



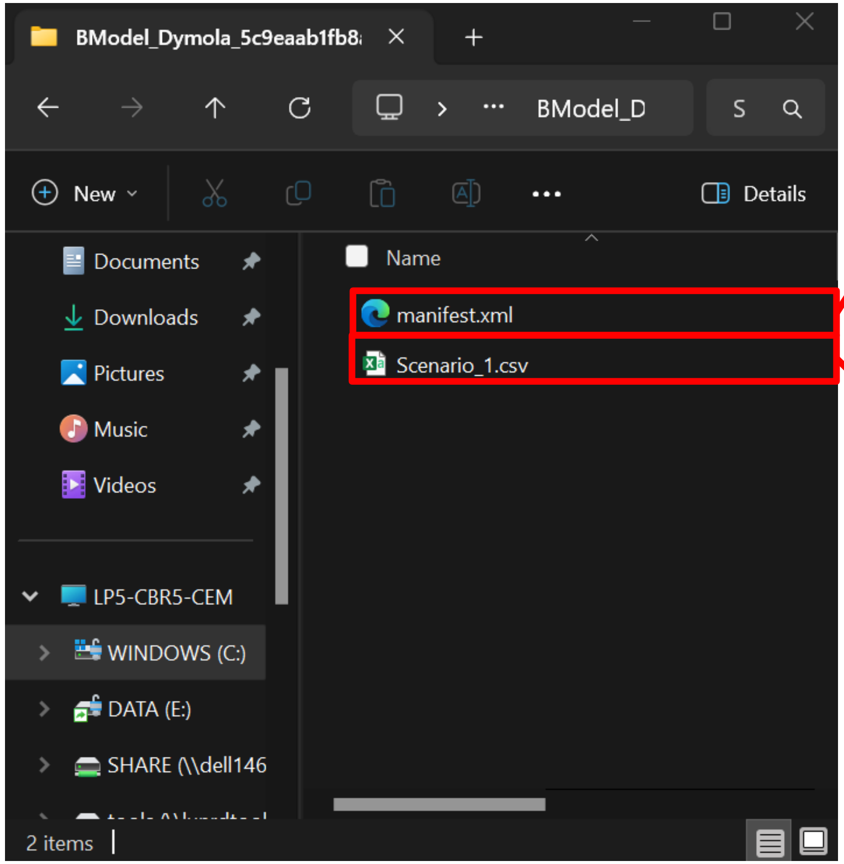
Build the Behavioral Model container with reference results taken from simulation of the reference experiment `Test_ReferenceExperiment`:

1. Right click `build` of experiment package in *Package Browser* / *Projects* view
   → *Call Function…*
   → *OK*

Browse the generated Behavioral Model container:

1. Right click `browse_container` of experiment package in the *Package Browser* / *Projects* view
   → *Call Function…*
   → *OK*

Investigate the generated Behavioral Model container (`BModel_Dymola_5c9eaab1fb8…`):

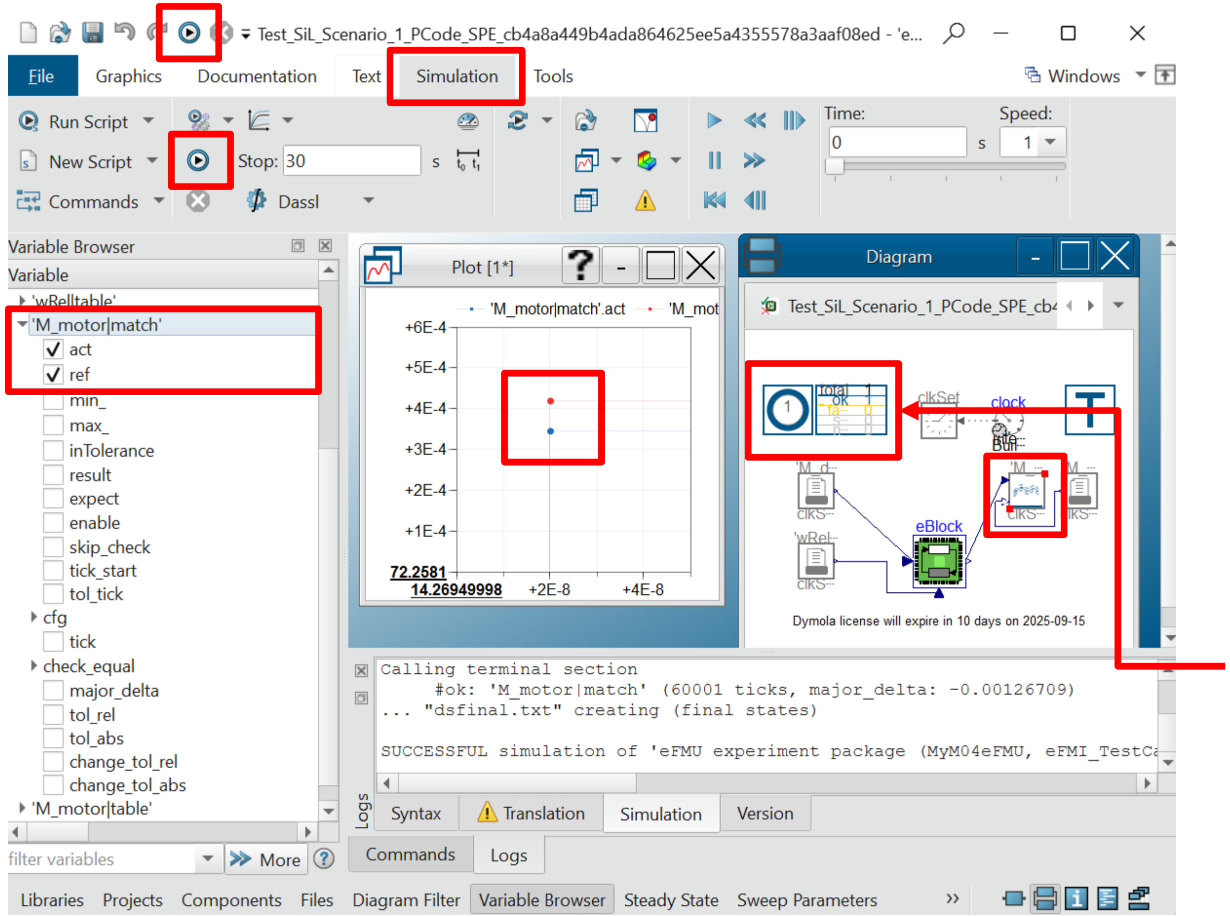


Container content:

- **XML manifest** with
  - Test scenarios
  - Links to Algorithm Code manifest for variable names and types (in-, output, tuneable parameter) & sample period
  - Variables → CSV column name links (multi-dimensions are flattened to individual columns)
  - Tolerances for various floating-point precisions
- **Reference trajectories** in comma separated values (CSV) files (one file per test scenario)

Take some time to investigate the manifest and CSV file.

Conduct SiL test of Software Production Engineering generated production codes:

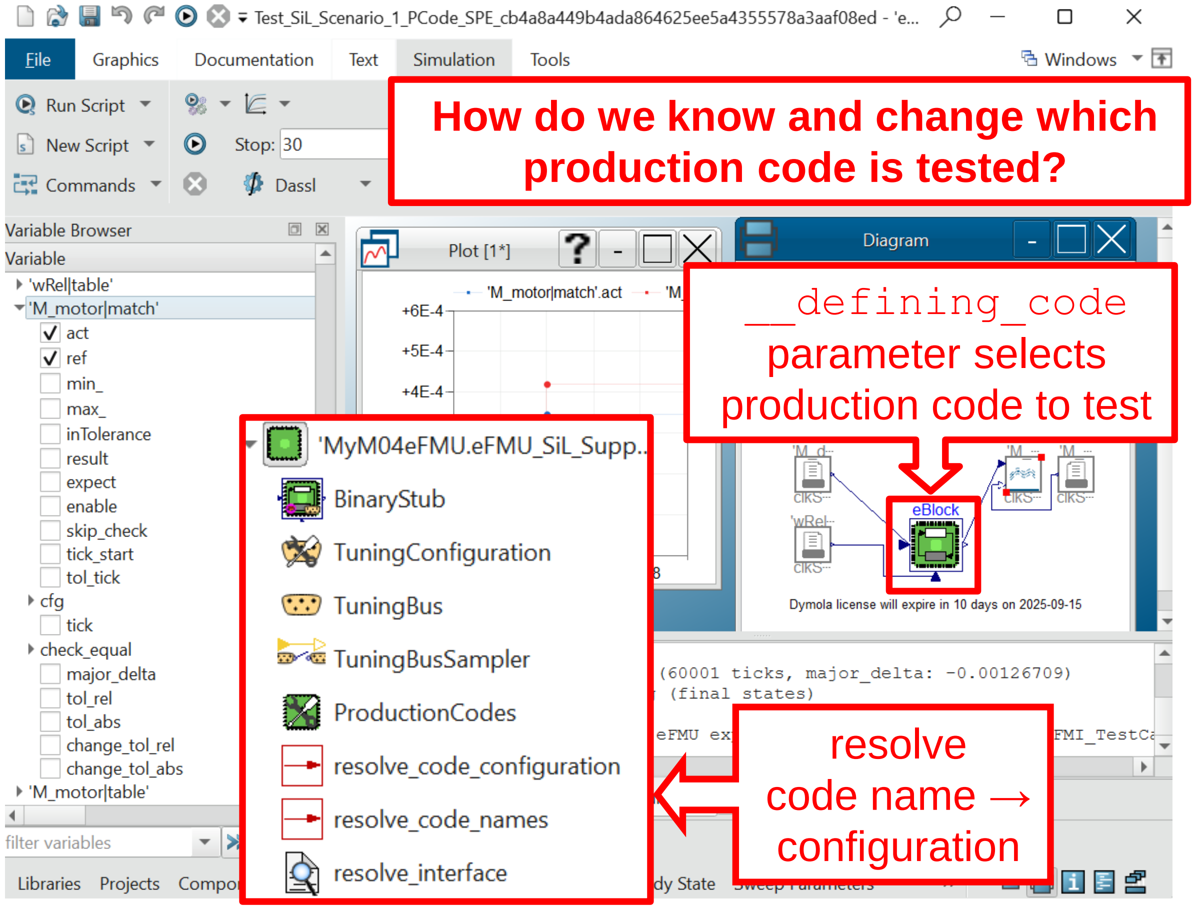


1. Double click `Test_SiL_Scenario_1_PCode_SPE_cb4...` of the experiment package in *Package Browser / Projects* view
2. Switch to *Simulation* ribbon
   → Click *Simulate* button
3. Right click `'M_motor|match'` in diagram plot
   → *Plot Variable*
   → select *act* (actual SiL simulation trajectory)
   → select *ref* (expected reference trajectory)
4. Zoom into *Plot* window to see there are differences

Note, that the test did not fail (see *Logs* window & dashboards). If you tighten tolerances – e.g., change the 32-Bit floating-point precision tolerances to the 64-Bit ones – it will fail.

eFMI Functional Mock-up Interface for embedded systems

Conduct SiL test of Software Production Engineering generated production codes:

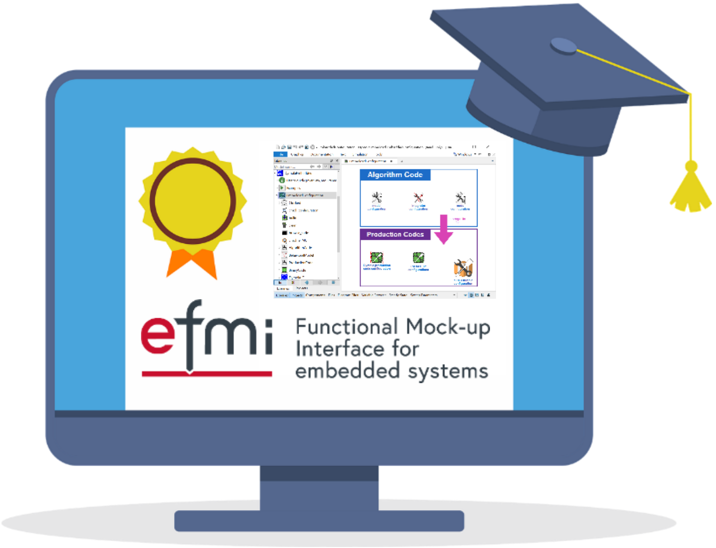


1. Double click `Test_SiL_Scenario_1_PCode_SPE_cb4...` of the experiment package in *Package Browser / Projects* view
2. Switch to *Simulation* ribbon
   → Click *Simulate* button
3. Right click `'M_motor|match'` in diagram plot
   → *Plot Variable*
   → select *act* (actual SiL simulation trajectory)
   → select *ref* (expected reference trajectory)
4. Zoom into *Plot* window to see there are differences

Note, that the test did not fail (see *Logs* window & dashboards). If you tighten tolerances – e.g., change the 32-Bit floating-point precision tolerances to the 64-Bit ones – it will fail.

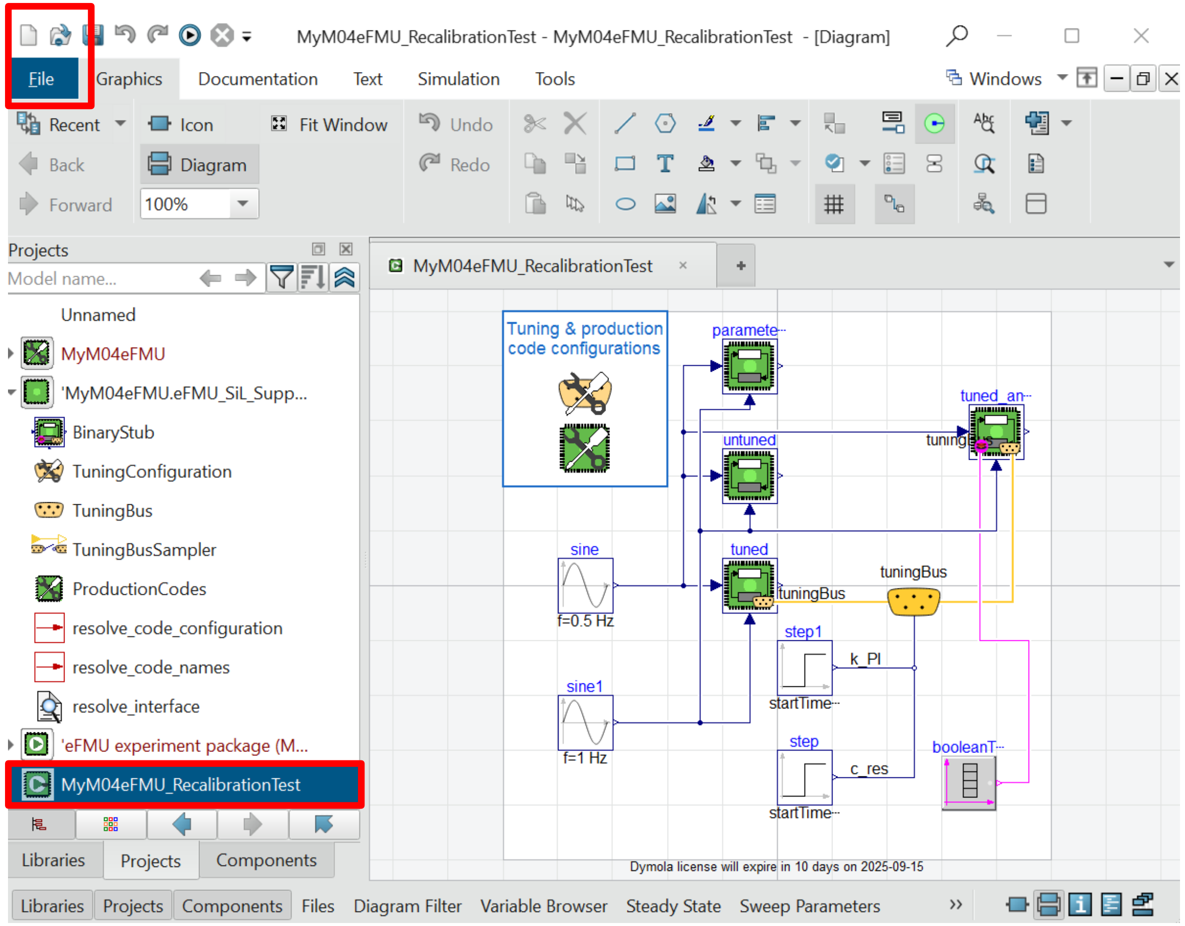
Functional Mock-up Interface for embedded systems

Congratulations, you did it!



Let’s do some advanced SiL stuff, like recalibration and reinitialization.

Load prepared recalibration & reinitialization example for M04 controller:

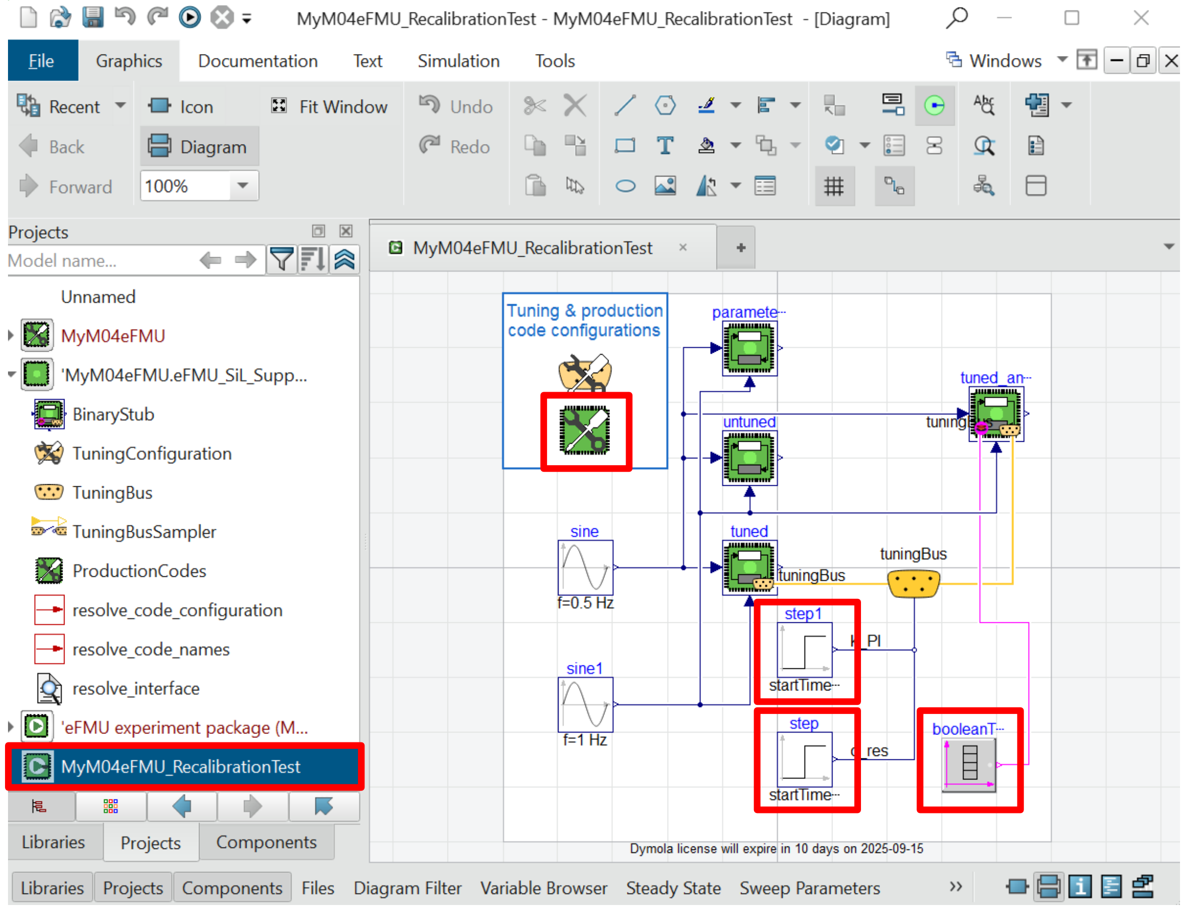


1. Either, drag and drop model `reference-models/Part-3/MyM04eFMU_RecalibrationTest.mo` in *Package Browser / Projects* view or load it via *File* → *Open* → *Load…*

The model has 4x M04 controller instances (eFMU co-simulation stub instances):

1. `untuned`: not modified, recalibrated nor reinitialized
2. `parameterized`: modified `c_res` & `k_PI` parameters, but not recalibrated nor reinitialized
3. `tuned`: unmodified, but via `tuningBus` runtime recalibrated `c_res` & `k_PI` parameters
4. `tuned_and_reinitialized`: like 3, but additionally at runtime reinitialized

Investigate recalibration & reinitialization example for M04 controller:



All 4 controllers use the same production code for simulation (`__defining_code` modification set by the global record parameter in the upper left of the diagram).

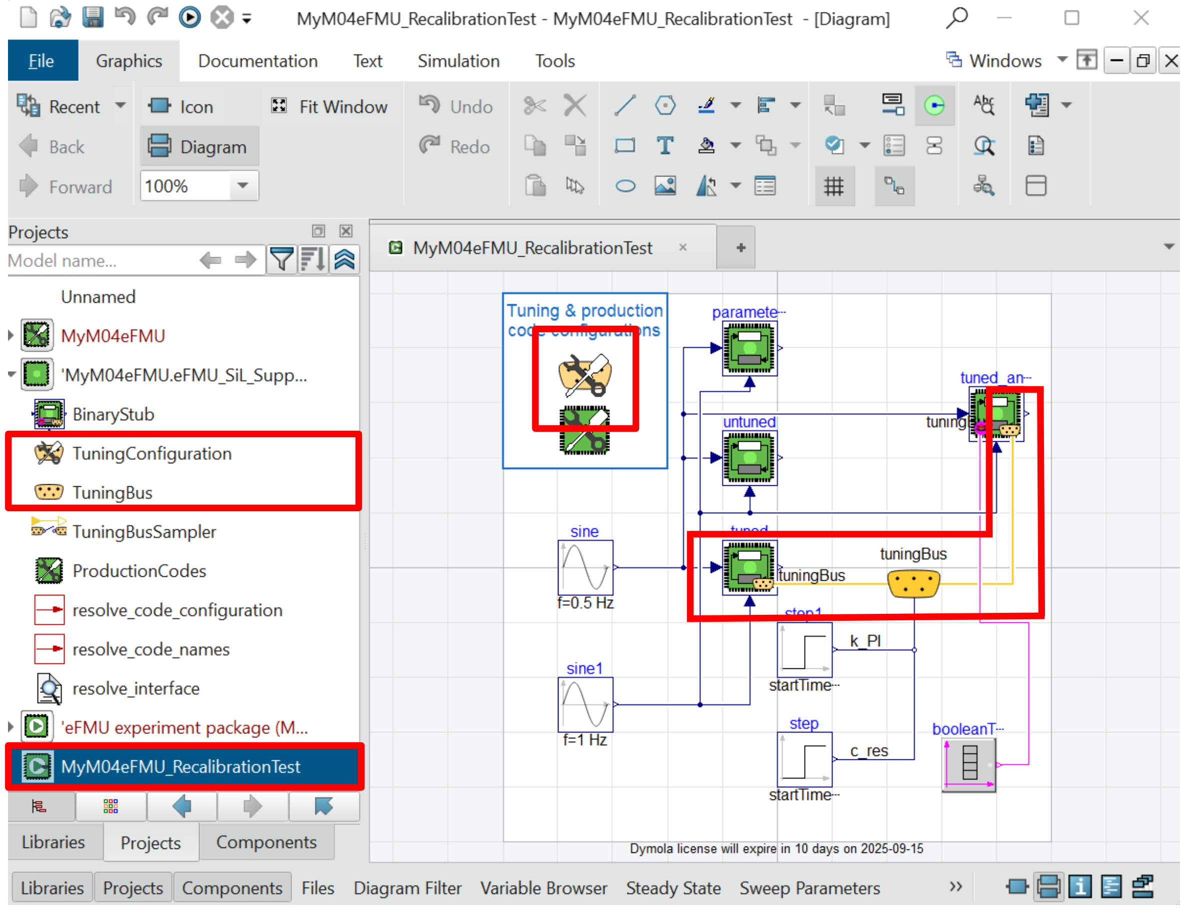
The `c_res` & `k_PI` parameter changes are all switches from the default value to the same new value, just at different time points (as modification before simulation or as recalibration during simulation):

- `c_res`: `4710` → `2710` at t = 0s or 0.25s (`step` runtime value)
- `k_PI`: `-73` → `-10` at t = 0s or 0.6s (`step1` runtime value)

Reinitalization is done at t = 0.7005s (`booleanTable` runtime value).

Functional Mock-up Interface for embedded systems

Investigate recalibration & reinitialization example for M04 controller:



Tuning is enabled by modifying co-simulation subs:

- `__enable_tuning = true`
- selecting/activating the tuned parameters via `__tuning_configuration`

⇒ The tuning bus connector ( ) is enabled.

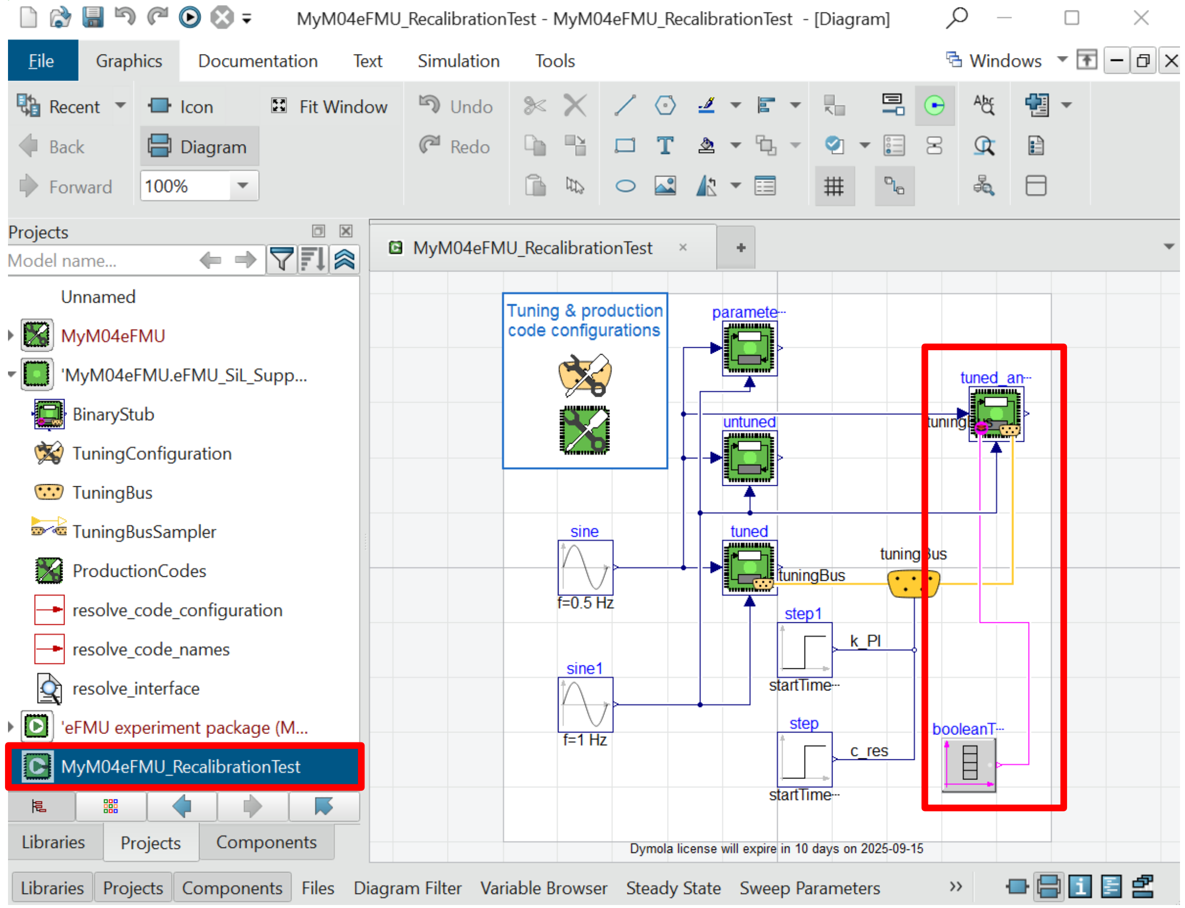
New recalibration parameter values are provided as runtime values connected to the tuning bus. Only tuning-activated parameters have to be provisioned.

Tuning configuration & bus types are provided in the generated eFMU co-simulation stub (drag and drop).

In this model: Tuneable parameters are selected by the global `__tuning_configuration` record parameter in the upper left of the diagram.

Functional Mock-up Interface for embedded systems

Investigate recalibration & reinitialization example for M04 controller:



Reinitialization is enabled by modifying eFMI co-simulation subs:

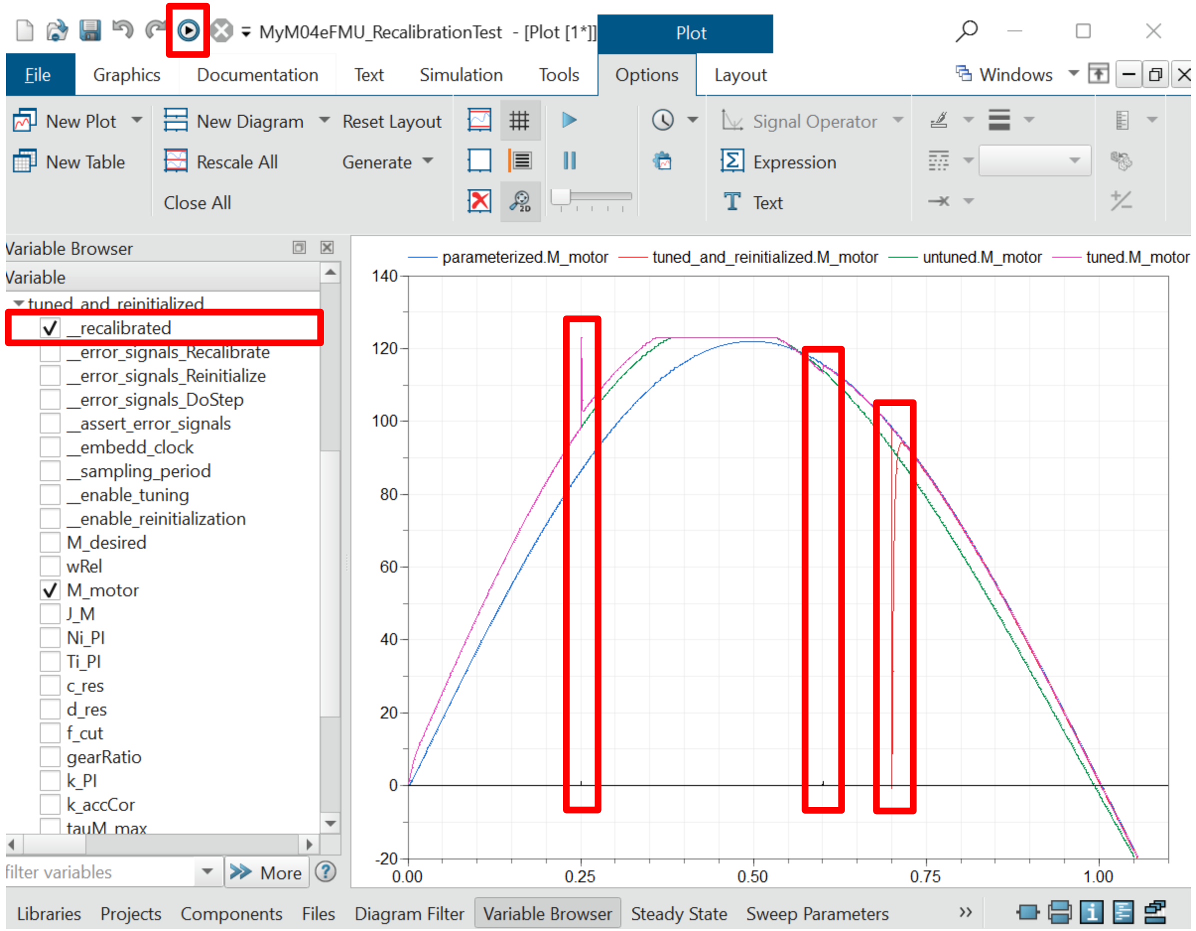
- `__enable_reinitialization = true`

⇒ The “stop push button” ( ) is enabled.

New reinitialization requests are provided as runtime values connected to the “stop push button”. Such are locked until the next sampling; it is sufficient to signal at any point inbetween two samplings that a reinitialization is requested – it is not necessary to ensure `__reinitialize == true` exactly at the sampling.

Functional Mock-up Interface for embedded systems

Investigate recalibration & reinitialization example for M04 controller:



1. Simulate `MyM04eFMU_RecalibrationTest`
2. Plot `M_motor` of all 4 co-simulation stubs
3. Plot `__recalibrated` (true, iff recalibration done)
4. Zoom into the plot at 0.0 ≤ t ≤ 1.05

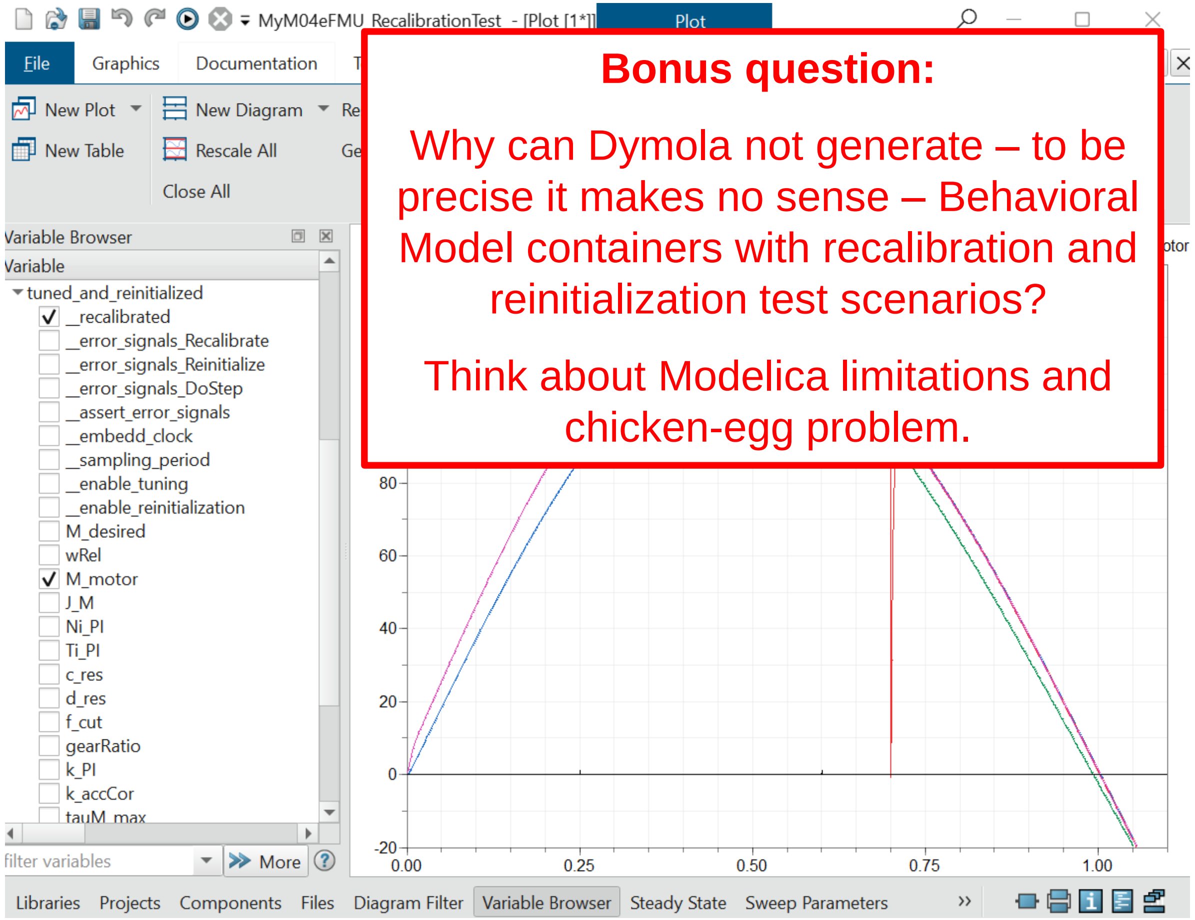
When do `parameterized` and `tuned` plots align? When does `untuned` align? Is the controller fast adapting in case of errors that require a system restart?

Good to remember:

- All controllers use same production code
- `c_res` & `k_PI` parameters change consistently:
  - `c_res` at t = 0s or 0.25s (`step`)
  - `k_PI` at t = 0s or 0.6s (`step1`)
- Reinitalization at t = 0.7005s (`booleanTable`)

efmi Functional Mock-up Interface for embedded systems

Investigate recalibration & reinitialization example for M04 controller:



Bonus question:

Why can Dymola not generate – to be precise it makes no sense – Behavioral Model containers with recalibration and reinitialization test scenarios?

Think about Modelica limitations and chicken-egg problem.

1. Simulate `MyM04eFMU_RecalibrationTest`
2. Plot `M_motor` of all 4 co-simulation stubs
3. Plot `__recalibrated` (true, iff recalibration done)
4. Zoom into the plot at 0.0 ≤ t ≤ 1.05

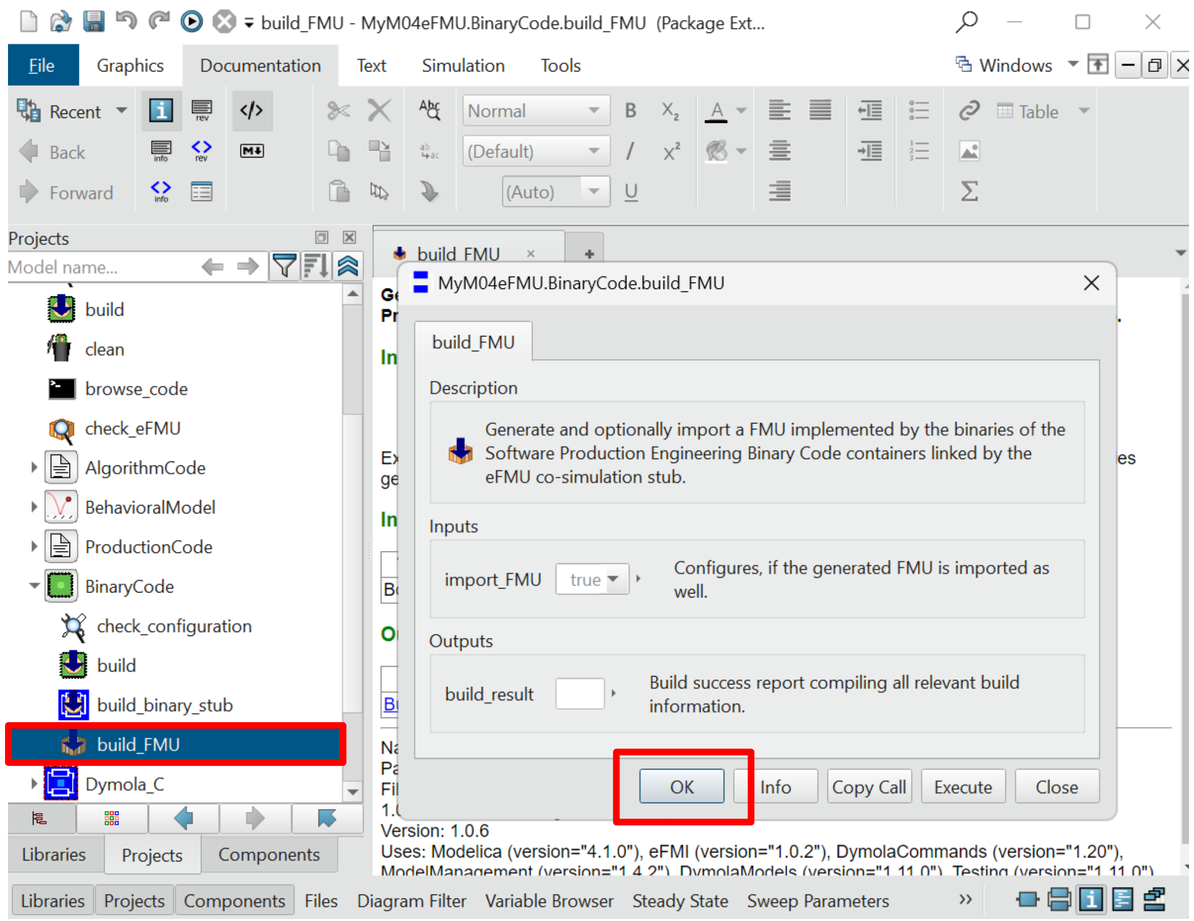
When do `parameterized` and `tuned` plots align? When does `untuned` align? Is the controller fast adapting in case of errors that require a system restart?

Good to remember:

- All controllers use same production code
- `c_res` & `k_PI` parameters change consistently:
  - `c_res` at t = 0s or 0.25s (`step`)
  - `k_PI` at t = 0s or 0.6s (`step1`)
- Reinitalization at t = 0.7005s (`booleanTable`)

Functional Mock-up Interface for embedded systems

Final touch – export eFMU as FMU:



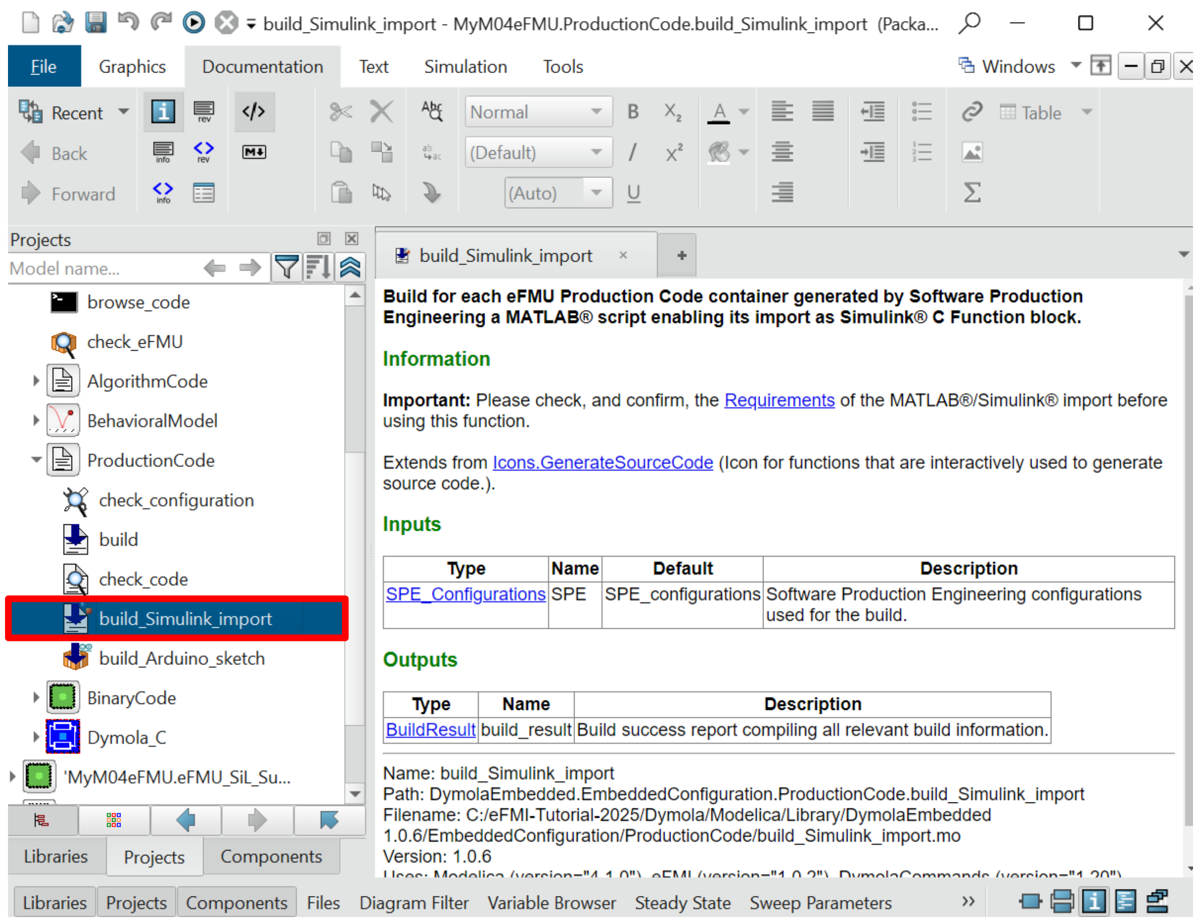
1. Right click `MyM04eFMU.BinaryCode.build_FMU` in *Package Browser / Projects* view
   → *Call Function…*
   → *OK*

The exported FMU has all conditional parameters of the eFMU co-simulation stub fixed to their defaults:

- Floating-point precision: precision of `__defining_code` production code
- Recalibration & reinitialization: disabled, i.e., `__enable_tuning = false`, `__enable_reinitialization = false`
- Error signals: asserted, i.e., `__assert_error_signals = true`
- Internal sampling: embedded & fixed, i.e., `__embedd_clock = true`

Functional Mock-up Interface for embedded systems

Final touch – export eFMU production code as Simulink® C Function block:

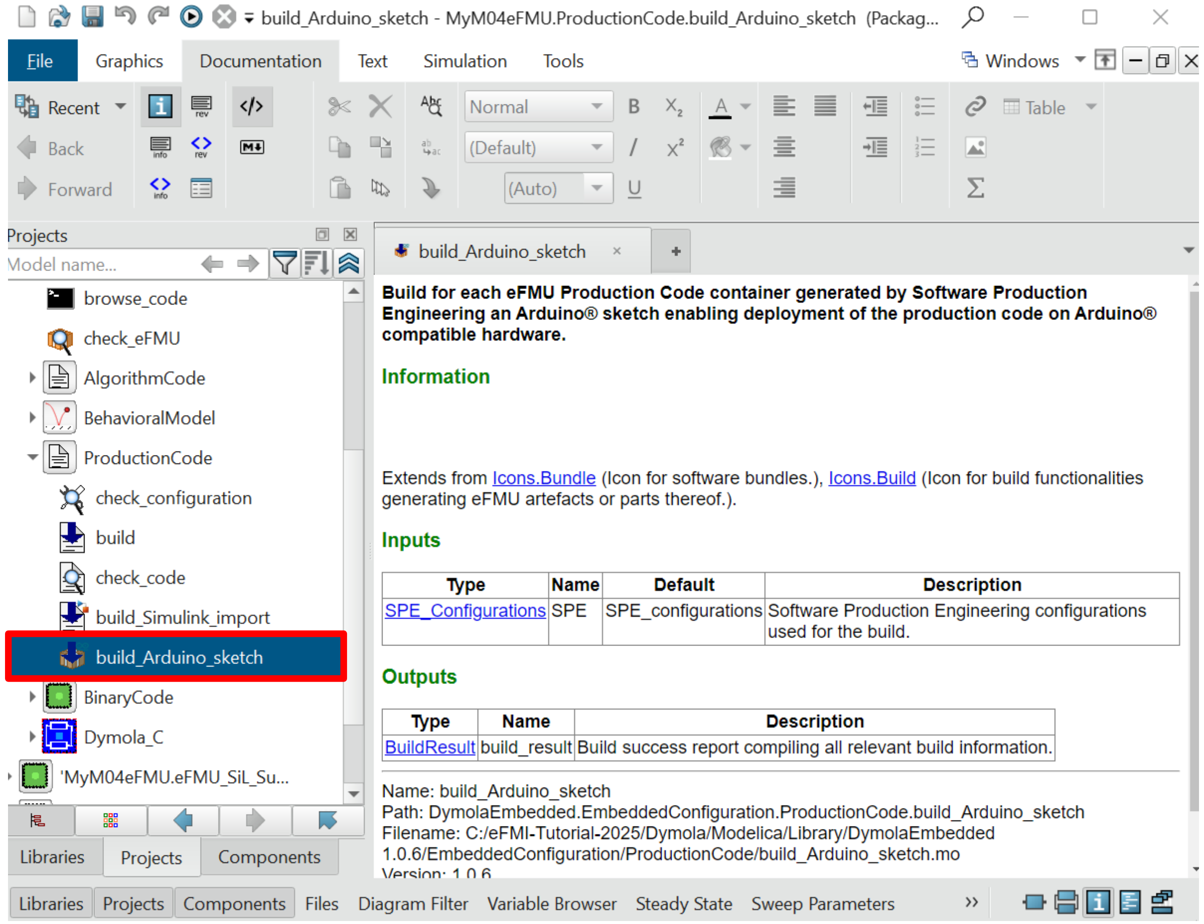


1. Right click `MyM04eFMU.ProductionCode.build_Simulink_import` in *Package Browser / Projects* view
   → *Call Function…*
   → *OK*

Exports a MATLAB® script for each Production Code container that can be used to configure Simulink® C Function blocks such that they are:

- backed (i.e., implemented) by the respective production code
- provide the GALEC block-interface using proper Simulink® types (in-, outputs and parameters)
- setup according to the GALEC block life-cycle

Final touch – export eFMU production code as Arduino® sketch:



1. Right click `MyM04eFMU.ProductionCode.build_Arduino_sketch` in *Package Browser / Projects* view
   → *Call Function…*
   → *OK*

Exports each Production Code container as Arduino® sketch such that the Arduino® IDE can be used for further development and deployment:

- Template comments hint at where in- and outputs have to be connected or parameters can be recalibrated.
- Preprocessor guarded timing code enables fast validation of worst time execution and correct scheduling.

Functional Mock-up Interface for embedded systems

Congratulations, you did it like a PRO!

Functional Mock-up
Interface for
embedded systems

eFMI® tutorial – Agenda

Part 1: eFMI® motivation and overview (40 min)

Part 2: Running use-case introduction (10 min)

Part 3: Hands-on in Dymola and Software Production Engineering (25 min)

Coffee break (30 min)

Part 3: Hands-on in Dymola and Software Production Engineering (30 min)

Part 4: Advanced demonstrators (20 min)

Part 5 (industry case-study): eFMI based thermal management system (TMS) development for fuel cell electric vehicles (FCEV) (20 min)

Part 6: Outlook and conclusion (5 min)

Tutorial leader:
Christoff Bürger

DASSAULT SYSTEMES

Presenter:
Daeoh Kang

iVH Institute of Vehicle Engineering

efmi Functional Mock-up Interface for embedded systems

License for 

https://pixabay.com/illustrations/education-online-school-elearning-5307517/

I create these images with love and like to share them with you. My passion is to provide vintage designs to honor those artists that created something great and timeless. You are most welcome to use it for commercial projects, no need to ask for permission. I only ask that you not resell my images AS IS or claim them as your own creation. As always, a BIG thank you for the coffee donations I received, every dollar is a blessing for my family.

Education Online School royalty-free stock illustration. Free for use & download.

Content License Summary

Welcome to Pixabay! Pixabay is a vibrant community of authors, artists and creators sharing royalty-free images, video, audio and other media. We refer to this collectively as "**Content**". By accessing and using Content, or by contributing Content, you agree to comply with our Content License.

At Pixabay, we like to keep things as simple as possible. For this reason, we have created this short summary of our Content License which is available in full here. Please keep in mind that only the full Content License is legally binding.

What are you allowed to do with Content?

- Subject to the Prohibited Uses (see below), the Content License allows users to:
- Use Content for free
- Use Content without having to attribute the author (although giving credit is always appreciated by our community!)
- Modify or adapt Content into new works

What are you not allowed to do with Content?

We refer to these as Prohibited Uses which include:

- You cannot sell or distribute Content (either in digital or physical form) on a Standalone basis. Standalone means where no creative effort has been applied to the Content and it remains in substantially the same form as it exists on our website.
- If Content contains any recognisable trademarks, logos or brands, you cannot use that Content for commercial purposes in relation to goods and services. In particular, you cannot print that Content on merchandise or other physical products for sale.
- You cannot use Content in any immoral or illegal way, especially Content which features recognisable people.
- You cannot use Content in a misleading or deceptive way.
- Please be aware that certain Content may be subject to additional intellectual property rights (such as copyrights, trademarks, design rights), moral rights, proprietary rights, property rights, privacy rights or similar. It is your responsibility to check whether you require the consent of a third party or a license to use Content.