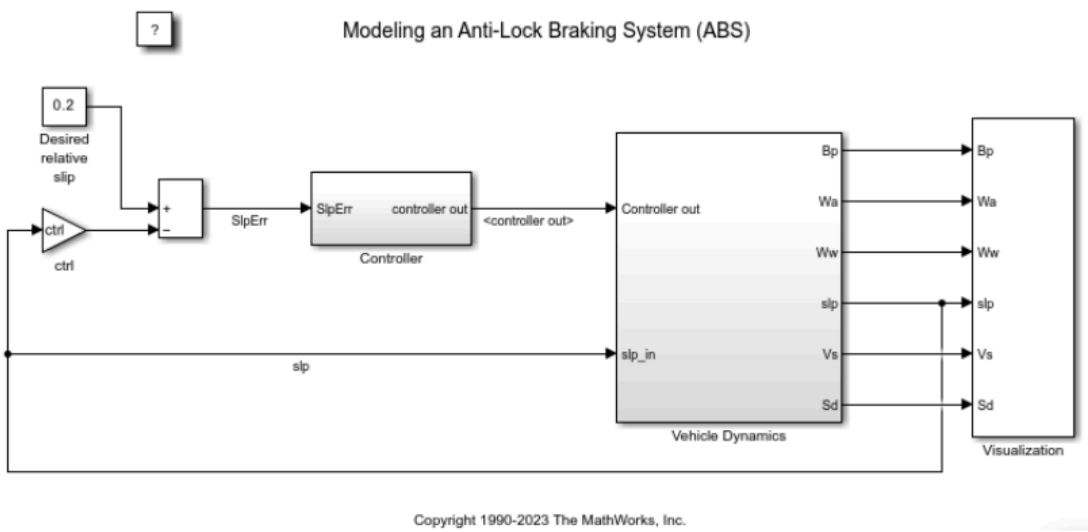
Introduction

A Functional Mockup Unit (FMU) is an abstract presentation of a dynamic system based on a tool-independent standard interface called Functional Mockup Interface (FMI). This interface supports both model exchange (equations only) and cross-platform co-simulation (solver is embedded) using a combination of model description files (.XML), metadata, and compiled Python code. The encapsulated dynamic systems are packaged into a portable format that can be embedded into larger systems/projects. This modular approach bridges the gap between domain-specific modeling environments and integrated simulation platforms, which ensures reusability, system-level integration, and real-time interaction.

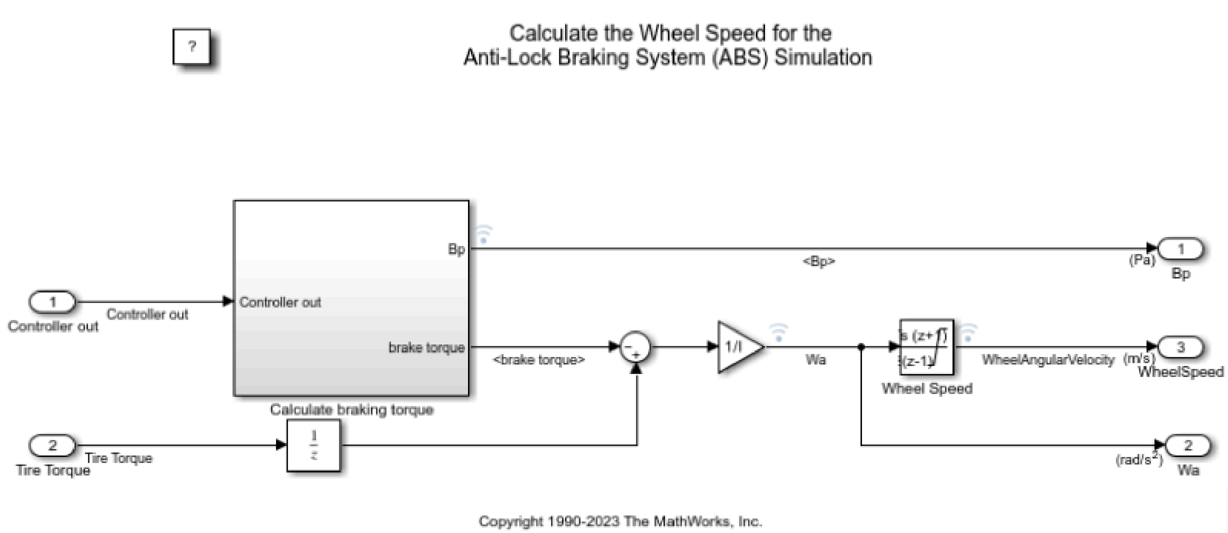
Our primary objective in this part of the project is to develop and simulate a braking system model using MATLAB/Simulink and subsequently exporting it as an FMU to enable co-simulation and integration with our ROS-based IoT framework.

Methodology

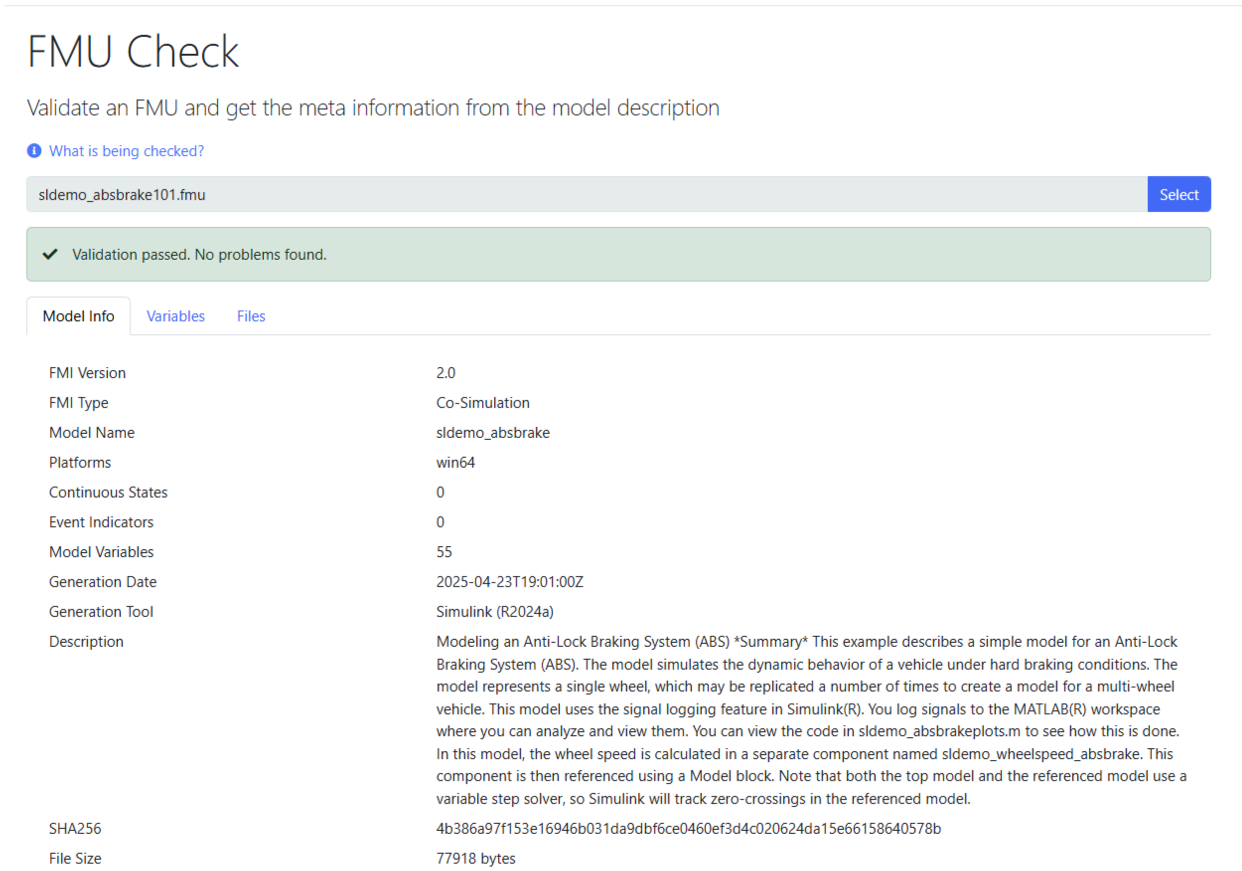
A Simulink model of an Anti-Lock Braking System (ABS) was chosen to replicate the physical system for performance monitoring of the digital twin on the CARLA simulator. The model simulates the dynamics of wheel deceleration under brake pressure according to the calculated delta slip under varying vehicle speeds received from the digital twin.



To control the rate of change of brake pressure, the model subtracts actual slip from the desired slip and feeds this signal into a bang-bang control, which integrates the signal to yield the actual brake pressure and hence, brake torque. The vehicle dynamics model then calculates the net torque on the wheel, which is divided by the wheel rotation inertia to yield the wheel acceleration and hence, the wheel velocity.



The model was tested in simulation mode to verify its behavior under the given operating conditions. Key parameters such as wheel speed, current speed, desired speed, and slip were plotted. The FMU model was exported as a standalone FMU in co-simulation format, version 2.0. FMU-check website ([FMU Check]) was used to validate the exported file while viewing some information about the model as well as the variables of the model (inputs, outputs, parameters, etc.), and the files included.



FMU Check

Validate an FMU and get the meta information from the model description

What is being checked?

sldemo_absbrake101.fmu Select

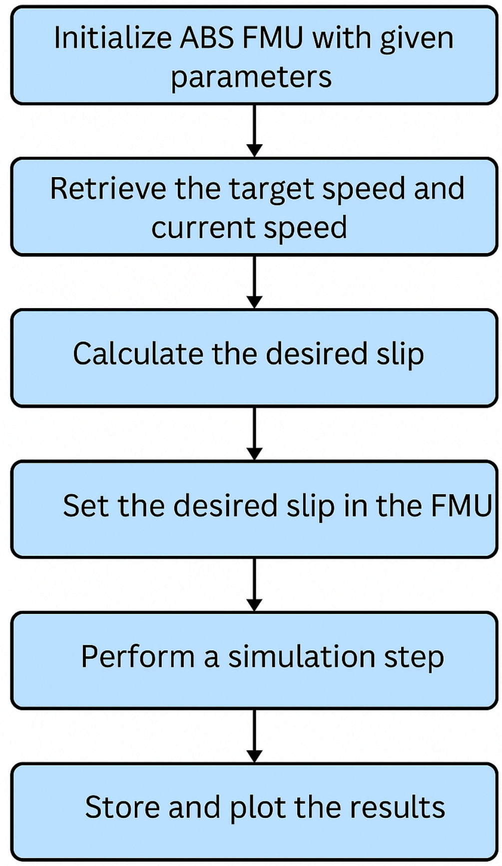
Validation passed. No problems found.

Model Info Variables Files

FMI Version	2.0
FMI Type	Co-Simulation
Model Name	sldemo_absbrake
Platforms	win64
Continuous States	0
Event Indicators	0
Model Variables	55
Generation Date	2025-04-23T19:01:00Z
Generation Tool	Simulink (R2024a)
Description	Modeling an Anti-Lock Braking System (ABS) *Summary* This example describes a simple model for an Anti-Lock Braking System (ABS). The model simulates the dynamic behavior of a vehicle under hard braking conditions. The model represents a single wheel, which may be replicated a number of times to create a model for a multi-wheel vehicle. This model uses the signal logging feature in Simulink(R). You log signals to the MATLAB(R) workspace where you can analyze and view them. You can view the code in sldemo_absbrakeplots.m to see how this is done. In this model, the wheel speed is calculated in a separate component named sldemo_wheelspeed_absbrake. This component is then referenced using a Model block. Note that both the top model and the referenced model use a variable step solver, so Simulink will track zero-crossings in the referenced model.
SHA256	4b386a97f153e16946b031da9dbf6ce0460ef3d4c020624da15e66158640578b
File Size	77918 bytes

Python Code using FmPy

The complete Python implementation of the ABS FMU simulator, including all functions and classes described in this section, is provided in **Appendix A (Source Code Listing)**.



1. **Initialization**

   a. Loading the FMU:

      - FMU is unpacked using extract()
      - Metadata is loaded
      - FMU is displayed using dump()

   b. Variable Handling:

      - Variable references are retrieved from the model
      - Variables are categorized by causality: parameter, inputs, outputs

   c. FMU Instantiation:

      - FMU is instantiated for co-simulation using instantiate()
      - Simulation experiment is set up via setupExperiment()
      - Initialization sequence is completed by calling enterInitializationMode(), applying default parameters, and then exitInitializationMode()

2. **Slip Calculation**

Given that the data received from CARLA are the current_speed and target_speed, we developed a function (speed_to_slip) to map these inputs to the desired_slip, which is the actual input to the model, using the following formula:

$$\text{desired\_slip} = \frac{\text{current\_speed} - \text{target\_speed}}{\max(\text{current\_speed}, 0.001)}$$

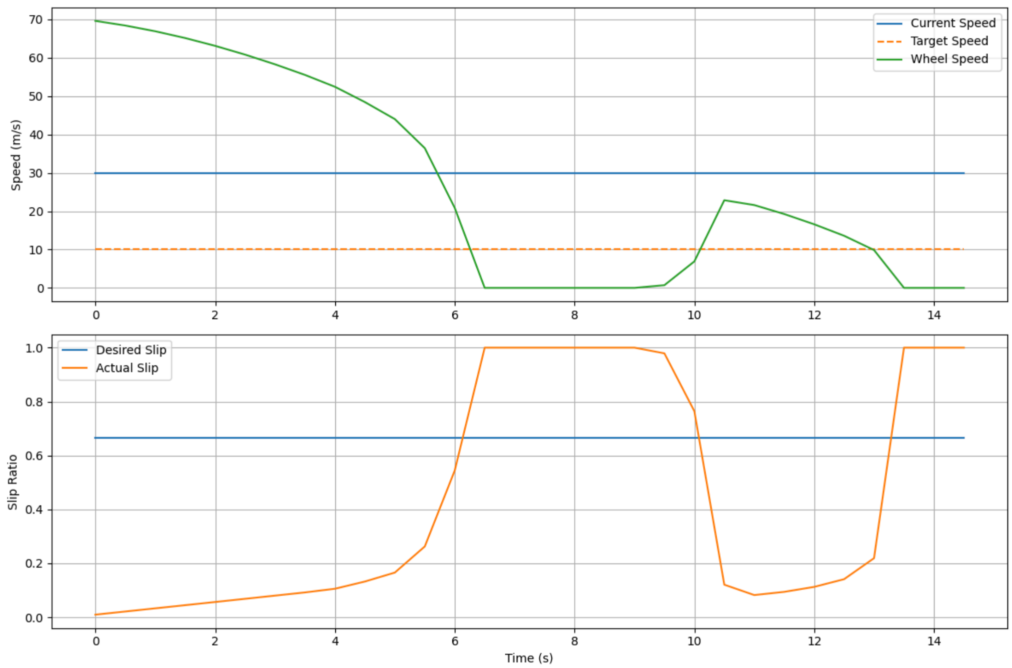
3. **Simulation Step**

The step() function is used to perform a single iteration of the simulation process:

- Calculates desired_slip using speed_to_slip() function
- Applies the calculated slip as input to FMU
- Advances the simulation by one time step using dostep()
- Captures and stores output values, including time, wheel speed, and slip
- Prints current parameter values for each time instance

4. Plotting Results

The results, including the wheel's speed and the desired_slip vs. the actual_slip were plotted over time to help visualize the dynamics of the system.



5. FMU Termination and Cleanup

terminate() ensures closing the simulation safely and helps with deleting temporary files

6. Main Simulation Execution

The simulation is run using the simulate_abs_braking() function, which acts as the main driver.

- Initialize the ABSSimulator instance with the specified FMU file and simulation parameters (current_speed, target_speed, total_time, step_size)
- Calls step() function to iterate until the total simulation time is reached
- plot_results() function is used to visualize the performance
- Cleans up by terminating the FMU instance and removing temporary files

Appendix A

```
from fmpy import read_model_description, extract, dump
from fmpy.fmi2 import FMU2Slave
import numpy as np
import matplotlib.pyplot as plt
import shutil

class ABSSimulator:
    def __init__(self, fmu_path, step_size=0.01):
        """
        Initialize the ABS FMU simulator

        Args:
            fmu_path (str): Path to the ABS FMU file
            step_size (float): Simulation step size in seconds
        """
        self.fmu_filename = fmu_path
        self.step_size = step_size
        self.start_time = 0.0
        self.time = self.start_time
        self.rows = []

        # Load and extract FMU
        dump(self.fmu_filename)
        self.model_description = read_model_description(self.fmu_filename)
        self.unzipdir = extract(self.fmu_filename)

        # Create value reference dictionary
        self.vrs = {var.name: var.valueReference for var in self.model_description.modelVariables}

        # Get variables by causality
        self.parameters = [v for v in self.model_description.modelVariables if v.causality ==
            'parameter']
        self.inputs = [v for v in self.model_description.modelVariables if v.causality == 'input']
        self.outputs = [v for v in self.model_description.modelVariables if v.causality == 'output']

        # Instantiate FMU
        self.fmu = FMU2Slave(
            guid=self.model_description.guid,
            unzipDirectory=self.unzipdir,
            modelIdentifier=self.model_description.coSimulation.modelIdentifier,
            instanceName='abs_instance'
        )

        # Initialize FMU
```

```
    self.fmu.instantiate()
    self.fmu.setupExperiment(startTime=0, stopTime=25.0)  # Extended simulation time
    self.fmu.enterInitializationMode()


    #print(f"\nFMU Initialized | Inputs: {[var.name for var in self.inputs]}")

    # Set default parameters (from model description)
    self.set_parameters({
        'Rr': 1.25,       # Wheel radius
        'ctrl': 1,        # Control parameter
        'g': 32.18,       # Gravity
        'm': 50,          # Mass
        # Mu (friction) parameters
        **{f'mu[1,{i}]': val for i, val in enumerate([
            0, 0.4, 0.8, 0.97, 1, 0.98, 0.96, 0.94, 0.92, 0.9,
            0.88, 0.855, 0.83, 0.81, 0.79, 0.77, 0.75, 0.73, 0.72, 0.71, 0.7
        ], start=1)},
        # Slip parameters
        **{f'slip[1,{i}]': val for i, val in enumerate([0, 0.05, 0.1], start=1)}
    })

    self.desired_slip_vr = None
    for var in self.inputs:
        if 'desired_slip' in var.name.lower():
            self.desired_slip_vr = var.valueReference
            break

    if self.desired_slip_vr is None:
        raise ValueError("'desired_slip' input not found in FMU")

    #self.fmu.exitInitializationMode()
    status = self.fmu.exitInitializationMode()
    if status != 0:
        raise RuntimeError(f"FMU failed to exit initialization mode (status={status})")
    print("FMU Initialized successfully!")

def set_parameters(self, parameters):
    """Set model parameters"""
    for name, value in parameters.items():
        if name in self.vrs:
            self.fmu.setReal([self.vrs[name]], [value])

def speed_to_slip(self, current_speed: float, target_speed: float) -> float:

    # Calculate desired slip (clamped 0-1)
```

```
        denominator = max(abs(current_speed), 1e-3)
        desired_slip = (current_speed - target_speed) / denominator
        return np.clip(desired_slip, 0.0, 1.0)

    def step(self, current_speed: float, target_speed: float):
        """
        Perform one simulation step with automatic slip calculation.

        Args:
            v_vehicle: Current vehicle speed (m/s)
            v_target: Target vehicle speed (m/s)
            desired_slip: Direct slip input (overrides calculation if provided)
        """
        if self.desired_slip_vr is None:
            raise RuntimeError("'desired_slip' input not initialized")

        # Calculate desired slip if target speed provided
        desired_slip = self.speed_to_slip(current_speed, target_speed)

        print(f"\n[Step Inputs]")
        print(f"Current Speed: {current_speed:.2f} m/s")
        print(f"Target Speed: {target_speed:.2f} m/s")
        print(f"Calculated Slip: {desired_slip:.3f}")


        # Apply to FMU
        #if desired_slip is not None and self.desired_slip_vr is not None:
        self.fmu.setReal([self.vrs['desired_slip']], [desired_slip])


        # Execute simulation step
        status = self.fmu.doStep(
            currentCommunicationPoint=self.time,
            communicationStepSize=self.step_size
        )

        #if status != 0:
            #raise RuntimeError(f"Step failed with status {status}")

        # Store results
        outputs = {
            'time': self.time,
            'current_speed': current_speed,
            'target_speed': target_speed,
            'desired_slip': desired_slip,
            **{var.name: self.fmu.getReal([var.valueReference])[0] for var in self.outputs}
```

```python
        }
        self.rows.append(outputs)
        self.time += self.step_size

        # Debug print
        #print(f"FMU Outputs - Ww: {outputs['Ww']:.2f} m/s | Actual Slip: {outputs.get('slp',
            'N/A'):.3f}")
        print(f"FMU Outputs: ")

    def terminate(self):
        """Clean up FMU resources"""
        self.fmu.terminate()
        self.fmu.freeInstance()
        shutil.rmtree(self.unzipdir, ignore_errors=True)

    def plot_results(self):
        """Enhanced plotting with slip comparison"""
        if not self.rows:
            print("No results to plot")
            return

        times = [r['time'] for r in self.rows]
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))


        # Speed plot
        ax1.plot(times, [r['current_speed'] for r in self.rows], label='Current Speed')
        ax1.plot(times, [r['target_speed'] for r in self.rows], '--', label='Target Speed')
        ax1.plot(times, [r['Ww'] for r in self.rows], label='Wheel Speed')
        ax1.set_ylabel('Speed (m/s)')
        ax1.legend()
        ax1.grid(True)

        # Slip plot
        ax2.plot(times, [r['desired_slip'] for r in self.rows], label='Desired Slip')
        if 'slp' in self.rows[0]:
            ax2.plot(times, [r['slp'] for r in self.rows], label='Actual Slip')
        ax2.set_xlabel('Time (s)')
        ax2.set_ylabel('Slip Ratio')
        ax2.legend()
        ax2.grid(True)


        plt.tight_layout()
        plt.show()
```

```
def simulate_abs_braking(fmu_path, current_speed, target_speed, step_size, total_time):
    """Run complete ABS braking simulation with automatic slip control"""
    simulator = ABSSimulator(fmu_path, step_size)
    num_steps = int(total_time/step_size)

    for _ in range(num_steps):
        simulator.step(current_speed=current_speed, target_speed=target_speed)

        last = simulator.rows[-1]
        print(f"t={last['time']:.1f}s | Ww: {last.get('Ww','N/A'):.2f} m/s | "
          f"Actual Slip: {last.get('slp','N/A'):.3f}")

    simulator.plot_results()
    results = simulator.rows
    simulator.terminate()
    return results

if __name__ == "__main__":
    #speed_profile = [(30.0, 25.0), (25.0, 15.0), (15.0, 5.0), (5.0, 0.0)]  # (current, target)

    results = simulate_abs_braking(
        fmu_path='F:/IIOT/FMU/sldemo_absbrake101.fmu',
        current_speed=30.0,
        target_speed=10.0,
        step_size=0.5,
        total_time=15.0
    )
```