Academic year
2024 - 2025

Provide a title

Inte Vleminckx

Supervisors
prof. dr. H. Vangheluwe, UAntwerpen
R. Mittal, Doctoral Fellow, UAntwerpen

University of Antwerp
Faculty of Science

Contents

1 Introduction

The development of products in the heating, ventilation, and air conditioning (HVAC) industry presents significant challenges in testing and validation. Building physical prototypes for every design iteration is often costly and time-consuming. A promising alternative is to model the most expensive or complex components in a virtual environment, enabling early testing without full-scale prototypes. This approach allows the evaluation of critical subsystems, particularly the control software that regulates HVAC systems.

In this study, we investigate how to test the control loop of a heating and ventilation system by modeling all physical elements—such as the valve, the actuator controlling the valve, the flow sensor, the pipe network, and the pressure pump that generates the fluid flow. The control loop, which determines the actuator setpoint based on the flow sensor measurements, will interact with the virtual model using co-simulation techniques. To assess the feasibility and performance of this approach, we compare two testing strategies: Software-in-the-Loop (SiL), and Hardware-in-the-Loop (HiL). In SiL testing, the model interacts with a compiled version of the control loop running on a separate system, with all connections established virtually. In HiL testing, the model runs on one system while the control loop is executed on the actual embedded hardware used in the real setup, with physical connections between the two.

Our methodology proceeds in stages. First, we develop a simple flow circuit in Modelica to demonstrate basic co-simulation capabilities. Using this model, we investigate how to integrate it with SiL and HiL environments. Once this foundation is established, we expand the Modelica component library with more detailed and realistic system elements. Finally, we construct an advanced flow circuit model and benchmark SiL results against HiL results to evaluate performance differences and validate the modeling approach.

Inte: add what can be expected in each upcomming section in one sentece.

2 Proposed Approach

2.1 The First Steps

To address the problem at hand, we must first establish a clear understanding of the overall problem statement. As outlined in the introduction, our goal is to test the control loop of a heating and ventilation system by modeling all physical elements and allowing the control loop to interact with this model. To achieve this, we need to define, as simply as possible, what the model requires as input from the control loop and what the control loop requires as input from the model, without yet considering detailed configuration parameters of the individual physical components. With this understanding in place, we can represent the interaction between the control loop and the model using the following diagram:

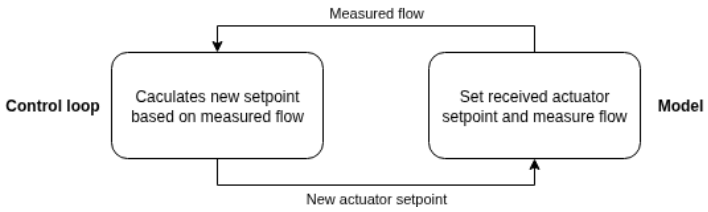


Figure 2.1: Simple represention of interaction between control loop and model

Next, we provide a more detailed explanation of the process illustrated in Figure 2.1. In this setup, the control loop receives a target flow setpoint. Although the source of this setpoint is not relevant to our study, it defines the desired flow rate in the circuit. To achieve this flow rate, the control loop calculates an actuator setpoint, which adjusts the valve opening to provide the required flow. In order to perform this calculation, the control loop must receive feedback from the model in the form of the measured flow in the circuit. By comparing the measured flow to the target flow, the control loop can determine the necessary actuator setpoint and adjust the valve position accordingly.

With this simplified setup, we can construct a Modelica model that accepts one real-valued input and produces one real-valued output—the actuator setpoint and the measured flow, respectively. For testing purposes, we can initially provide fixed output values from the model to the control loop to observe how the actuator setpoint evolves over time. To enable full co-simulation, we must also determine how to establish interaction between the Modelica model and the control loop. The workflow for this process is as follows:

1. Develop a simple Modelica model.
2. Investigate how the compiled control loop can be accessed and controlled using Python.
3. Integrate the Modelica model into the Python code so that the control loop can exchange data with the model in real time.

2.1.1 A Simple Modelica Model

A very simple Modelica model can be represented as shown in Figure 2.2.

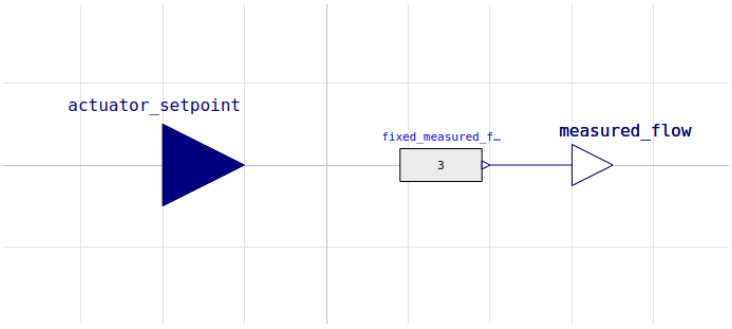


Figure 2.2: A simplified model representatino

This model accepts an actuator setpoint as input and produces a measured flow as output, which is currently fixed at a value of 3 for testing purposes.

2.1.2 Interaction With The Control Loop

The first step in enabling interaction with the control loop is to compile the common platform—containing the control loop—into an executable. This common platform is implemented in C and is compiled using CMake. Due to privacy regulations of the company related to this study, the source code of the common platform cannot be shared.

Once the executable is available, the next step is to establish a method for interacting with the common platform. There are two primary communication interfaces used by the platform:

1. A Flow Sensor Board (FSB), which emulates the behavior of a real flow sensor using a UART connection.
2. A Modbus interface, which allows reading registers from the embedded control board using a TCP connection.

In the case of Software-in-the-Loop (SiL) testing, the common platform is executed locally. A mock Modbus connection is created to replicate the behavior of a physical Modbus link to an embedded board. To keep the discussion focused, we first consider only the SiL setup. A schematic representation of this communication is shown in Figure 2.3.

To enable this interaction, we must first create virtual serial ports that allow communication with the common platform. This can be achieved using the following commands:

```
$ sudo socat -d -d pty,link=/dev/ttyV1,raw,echo=0 pty,link=/dev/ttyV2,raw,echo=0
$ sudo chmod 777 /dev/ttyV1 && sudo chmod 777 /dev/ttyV2 && \
sudo socat -d -d -u OPEN:/dev/ttyV1,raw tcp:localhost:8888,reuseaddr
```

Listing 2.1: Creating virtual ports for communication

This configuration uses socat to create a pair of virtual serial ports and forward data through a TCP socket. First, it establishes two pseudo-terminal devices—/dev/ttyV1 and /dev/ttyV2—which function

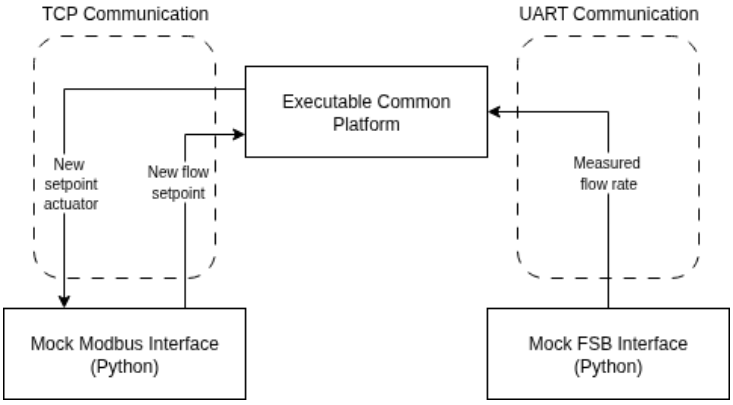


Figure 2.3: Overview communication with common platform

as the two ends of a virtual null-modem cable: any data written to one port immediately appears on the other. Next, their permissions are updated to allow unrestricted access, and a second socat process forwards all data received on /dev/ttyV1 to a TCP connection on port 8888.

The final step is to connect to the virtual ports from Python, enabling direct interaction with the common platform through Python code.

To achieve this, we use two Python modules:

1. The serial module.
2. The ModbusTcpClient module from the pymodbus.client library.

The serial module is used to implement the Flow Sensor Board (FSB) mock. For this purpose, we create a PeriodicPacket class that encapsulates all the data expected by the common platform from a real FSB. This packet is serialized into a byte format compatible with the common platform and transmitted over the serial connection via /dev/ttyV2.

The ModbusTcpClient module is used to implement the Modbus mock. This allows us to read and write registers that correspond to specific data points within the common platform. The Modbus client runs on localhost with port 8080.

Using this setup, we can send new flow measurements to the common platform by updating the PeriodicPacket and transmitting it over the serial connection. Additionally, by accessing the appropriate registers, we can read the actuator setpoint calculated by the common platform and write new flow setpoints directly into the system.

2.1.3 Complete Integration

To complete the integration, we must also interact with the Modelica model developed earlier.

The first step is to export the Modelica model so that it can be integrated into Python. This is achieved by exporting the model as a Functional Mock-up Unit (FMU). Since we intend to perform co-simulation, the FMU export must explicitly be configured for co-simulation mode rather than model-exchange mode.

Several Python libraries support FMU-based simulation, but not all of them allow for co-simulation with externally controlled simulation steps. Step control is essential because we need to dynamically update

the model inputs and retrieve outputs during execution, ensuring synchronization with the common platform. For this reason, we select the fmpy library, which provides full co-simulation capabilities and fine-grained step control.

With this functionality in place, the final task is to design a simulation loop that integrates all components, ensuring seamless data exchange between the Modelica model and the common platform. Before proceeding to implementation, we present a schematic diagram in Figure 2.4 summarizing the complete interaction setup. This diagram highlights the data flow between all components and clarifies which elements produce or consume specific signals.

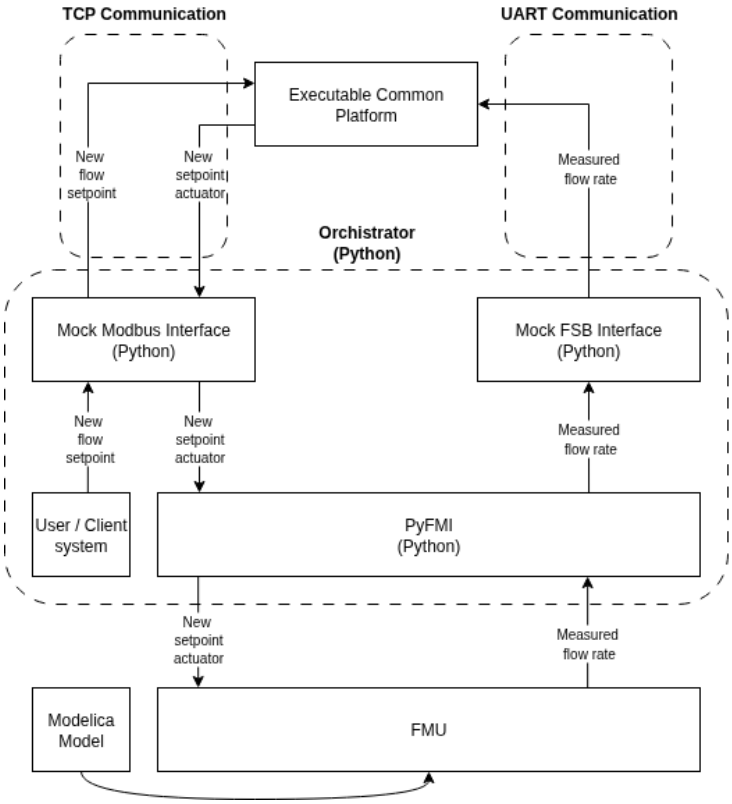


Figure 2.4: Total interaction overview between all components

We observe that this diagram extends Figure 2.3, with the orchestrator managing the communication between the common platform and the FMU. The simulation loop, capturing all interactions, is implemented as follows:

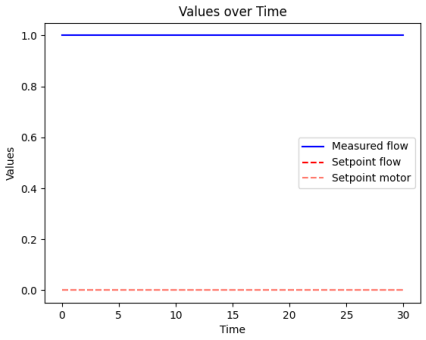
```
def run(self):
    performed_step = True

    # user input value
    s_flow = 0.0
    self._modbus.set_setpoint_flow(s_flow)

    while performed_step:
        time = self._model.current_time

        # Update measured flow
        m_flow = self._model.get_measured_flow()
        self._fsb.update_flow(m_flow)

        # Update model with new setpoint
```

```
        s_motor = self._modbus.get_setpoint_motor()
        self._model.set_setpoint_motor(s_motor)

        self._update_trace(time, m_flow, s_flow, s_motor)

        # Perform a step in the model
        performed_step = self._model.perform_step()
```

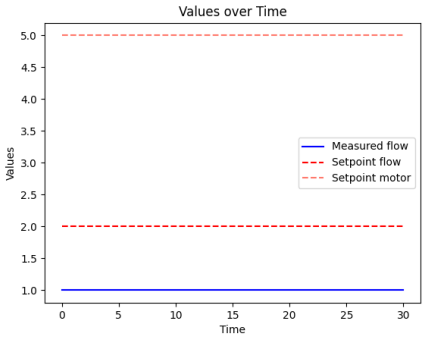
Listing 2.2: Simulation loop

For testing purposes, the simulation loop is deliberately kept simple. First, the measured flow is retrieved from the Modelica model and passed to the FSB, which transmits the corresponding packet to the common platform. Next, the motor setpoint is read from its Modbus register and sent back to the Modelica model. A trace of the key variables is recorded to enable visualization, and finally, the model performs a single simulation step.

To verify correct operation, two test cases are executed: one with the flow setpoint s_flow = 2.0 and another with s_flow = 0.0. The objective is to confirm that the common platform's control loop generates different motor positions depending on whether the flow setpoint is above or below the fixed measured flow value provided by the Modelica model (which is constant at 1.0 in this scenario).



(a) Simple simulation with s_flow = 0.0



(b) Simple simulation with s_flow = 2.0

As shown in Figure 2.5a, the motor setpoint is calculated as 0 when s_flow = 0, whereas in Figure 2.5b, the motor setpoint is 5 when s_flow = 2.0, both evaluated against a fixed measured flow of 1.0. These results confirm that the control loop correctly receives the measured flow and computes the motor setpoint accordingly.

This demonstrates that the minimal setup functions as intended: all components successfully interact with one another. With this validation in place, the next step is to extend the Modelica components to simulate a complete flow circuit and enhance the Python code to support the expanded system.

3 Modelica Library Overview

4 Results

5 Conclusion

Bibliography