M.Tech. Dissertation

On

Programming Embedded Systems with Open-Source Modeling Tools

Submitted in partial fulfillment of the requirements
of the degree of
Master of Technology
in Systems and Control Engineering

by

Sudhakar Kumar
Roll No. 183236001

Under the guidance of
Prof. P. S. V. Nataraj
Prof. Kannan M. Moudgalya

Department of Systems and Control Engineering
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
Powai, Mumbai, Maharashtra
India, 400 076
June 2021

Approval Sheet

This dissertation entitled "Programming Embedded Systems with Open-Source Modeling Tools" by Mr. Sudhakar Kumar (183236001) is approved for the degree of Master of Technology in Systems and Control Engineering from IIT Bombay.

Examiners

_____

_____

Supervisors

_____

_____

Chairman

_____

Date:

Place: IIT Bombay

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Sudhakar Kumar
Roll No. 183236001
M.Tech, SysCon
IIT Bombay

Date:

Place: IIT Bombay

Acknowledgement

I would like to express my sincere gratitude to Prof. P. S. V. Nataraj (Department of Systems and Control Engineering, IIT Bombay) for supporting this project. I wish to express my heartfelt gratitude to Prof. Kannan Moudgalya (Department of Chemical Engineering, IIT Bombay) for his systematic and valuable advice. His insightful directives have always steered me in the right direction and encouraged me to perform to the best of my abilities. I would like to thank him for driving *FOSSEE* and *Spoken Tutorial* projects which have been crucial for achieving the objectives of this project. I would also like to thank Prof. Sachin C Patwardhan (Department of Chemical Engineering, IIT Bombay) for several useful suggestions.

I want to thank Dr. Sunil Shah (ModeliCon InfoTech LLP) for providing me with invaluable inputs from time to time. His short telephonic conversations have been instrumental in shaping this project. I would like to mention the help I received from Mr. Ritesh Sharma (ModeliCon InfoTech LLP) in designing the Python OPC UA clients for this project. The many discussions that I had with him on the built-in OPC UA server in OpenModelica and Python OPC UA package helped me progress in this project. He has been kind enough to bear with me throughout this project.

I would like to extend my sincere thanks to Mr. Nikhil Sharma (Department of Chemical Engineering, IIT Bombay) for demystifying the modeling concepts of OpenModelica. Additionally, I would remain thankful to him for elaborating the batch distillation column. Had it not been for his support, I would never have realized the potential of control systems in chemical plants. I also take this opportunity to acknowledge the support offered by Mr. Priyam Nayak (FOSSEE, IIT Bombay) in visualizing the object-oriented programming in OpenModelica. I would like to acknowledge Mr. Rahul Patnikar (Department of Computer Science and Engineering, IIT Bombay) for helping me out in comprehending the discrete event simulation in OpenModelica. Last but not least, I would like to thank Mr. Sridhatta Jayaram Aithal (FOSSEE, IIT Bombay) for helping me with shell scripting.

My thanks and appreciation also go to my friends who have willingly helped me out with their abilities.

Sudhakar Kumar
Roll No. 183236001
M.Tech, SysCon
IIT Bombay

Date:

Place: IIT Bombay

Abstract

In this project, we are utilizing open-source modeling tools to program embedded systems. For this, we have applied the concept of Hardware-in-Loop (HIL) simulation, in which the model (digital twin) of a plant is being simulated in OpenModelica. Accordingly, we monitor or control this plant with an embedded system running on external hardware or a single board computer. A client-server model has been accomplished between these two systems. In this model, the plant running in OpenModelica has been treated as a server with which the embedded system running on external hardware will connect as a client. For this communication, we are using a protocol named OPC UA, which is a machine-to-machine protocol mainly used for industrial automation. OpenModelica has a built-in OPC UA server. Thus, we have developed a Python OPC UA client for the embedded system running on external hardware. As a proof-of-concept, we have built a client-server communication model between OpenModelica (server) and Raspberry Pi (client). We have also implemented controllers like ON-OFF, P, PI, PID controllers on the client-side to control various plants running at the server end. From the results, we may conclude that HIL can be a good option for digitizing a plant at a low cost and almost no time investment. Furthermore, making HIL a part of the system adds value to the finished product, which is not the case if we use HIL only as a hardware testing tool. This project puts forward a good alternative to proprietary simulators in this direction. Along with designing Python OPC UA clients, we have also discussed the implementation of OPC UA in OpenModelica. It would provide the readers with an insight into the working of OPC UA in OpenModelica. We have also worked on automating the simulation in OpenModelicaby developing a shell script and an user interface. We have explored the availability of OPC UA servers in other simulation tools like MATLAB, Scilab, DWSIM, etc. As none of these tools have built-in OPC UA servers, OpenModelica becomes the only choice for those who wish to simulate a model or a plant over an OPC UA server. We have also discussed how a cyber-physical system (CPS) can be realized using the object-oriented programming in OpenModelica.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the introduction of the fourth industrial revolution (Industry 4.0), it has become an unavoidable necessity for a manufacturing plant to have an identical digital twin to realize the concept of Industrial Internet of Things (IoT) [1]. Here, the digital twin is not just a graphical representation of the plant but is in sync with the hardware and adds value to the plant's actual functioning in real-time. We know that a cyber-physical system (CPS) is a programmed computer system to monitor (control) a mechanism involving sensors and actuators [2]. Due to its efficacy and the presence of IoT [3] [4], almost all manufacturing units want to deploy a CPS for monitoring (controlling) a wide range of processes (mechanisms) running in their units. For this, they need to simulate the process first in sync with an external controller (posing as CPS) to check whether the process gives the desired results under feedback input from an external controller. After the engineers (of the manufacturing units) receive satisfactory performance from this set-up, they can plan for deploying a CPS to monitor their processes. Therefore, we need to devise techniques and tools by which we can simulate or program embedded systems in sync with external hardware. Adding to this, we need to focus on open-source tools, which would reduce our dependency on proprietary solutions and help us create cost-effective solutions for the entire society.

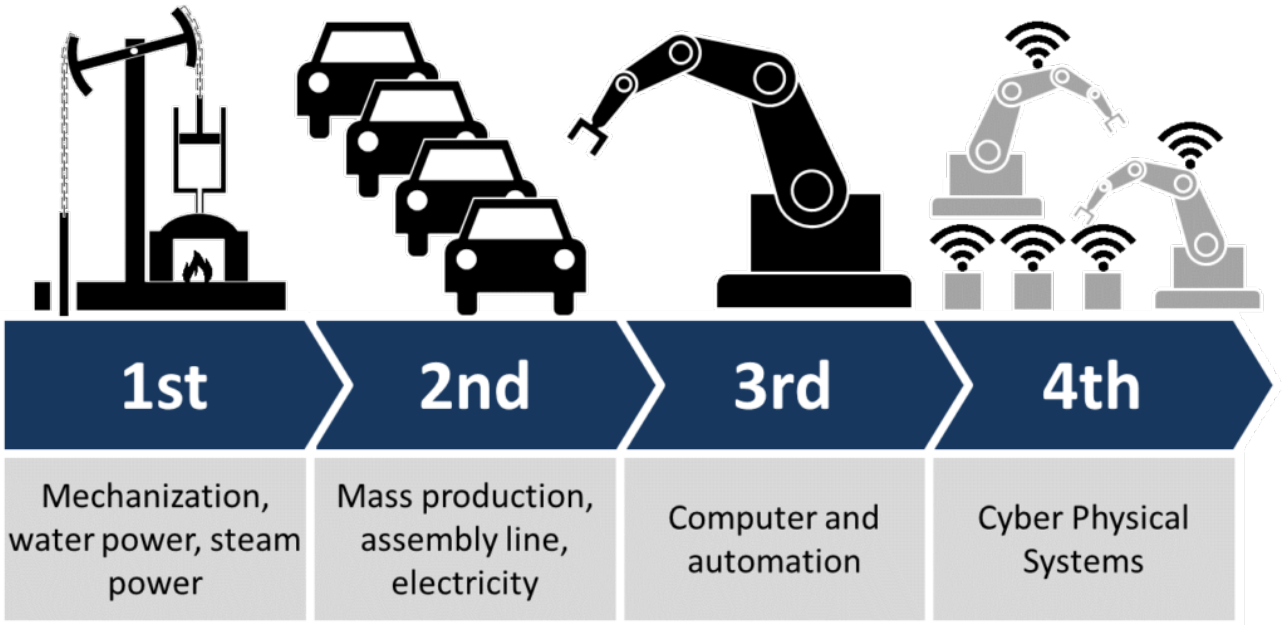

Figure 1.1: Focus of the four industrial revolutions
Source: Industry 4.0 – The Fourth Industrial Revolution.
https://www.renaix.com/industry-4-0-the-fourth-industrial-revolution/

This project will discuss how to simulate (control) a digital twin of a process (plant) using an

external controller. Let us consider a plant. As the first step, we will derive its mathematical model theoretically. Next, we will design a controller to help the plant gain the desired set-point or desired performance. Now, we need two separate systems. One, we will require a modeling tool that will run the plant in real-time. Two, we will require a microprocessor or microcontroller to host the controller and establish a duplex mode of communication with the plant. As we can see, the first system (modeling tool) would run a digital twin of the actual plant being studied. The second system (controller) would be a physical system that would monitor the plant's states and provide some feedback to the plant so that the plant can provide the desired results. In other words, the latter system would control the behavior of the former system. For the second system, we will have a microprocessor or microcontroller on which the controller will run. Let us call the second system the hardware. As we can see, the hardware is controlling a plant being simulated in a modeling tool. Though the hardware is not dealing with the actual plant (in physical form), it would run in a loop as if it is connected with the actual plant. This type of technique is known as Hardware-in-Loop (HIL) simulation [5]. HIL simulation is a type of real-time simulation. We use the HIL simulation to test our controller design. HIL simulation shows how our controller responds, in real-time, to realistic virtual stimuli. We can also use HIL to determine if our physical system (plant) model is valid. In HIL simulation, we use a real-time computer as a virtual representation of our plant model and a real version of our controller. In our HIL set-up, OpenModelica, an open-source implementation of the Modelica language [6] will have a plant running on it. Along with this, a controller running on external hardware will control the plant running in OpenModelicavia OPC UAcommunication protocol.

From the literature survey, it is pretty evident that OpenModelica has widely been used as a modeling tool for programming embedded systems. In the paper titled *Chemical process simulation using OpenModelica* [6], the authors have discussed how an OpenModelica simulation environment is suitable for both steady-state and dynamic simulation of chemical processes. In this research article, the authors have demonstrated OpenModelica's capability by simulating the semi-batch steam distillation of a binary mixture. They have also shown that the results of OpenModelicaare in agreement with that of the model being simulated in other proprietary tools.

In another article titled *Modelica as a design tool for hardware-in-the-loop simulation* [7], the authors have discussed the benefits of Modelica as a simulation tool as well as a design tool for HIL simulation. According to this article, the best attribute of Modelica is that it is domain-independent. Therefore it can be used for simulation as well as multi-domain modeling.

In another article titled *Feasibility analysis for networked control systems by simulation in Modelica* [8], the authors have emphasized decentralized structures in automation and control. Also, they have discussed their impacts. For instance, decentralization adds new behavior in the form of non-deterministic delays to the resulting networked control systems. Accordingly, they conclude that simulation is a viable approach to analyze a closed-loop control system including network communication.

In another article titled *Dynamic simulation of chemical engineering systems using OpenModelica and CAPE-OPEN* [9], a Modelica library allowing an interface between Modelica and CAPE-OPEN is developed. Its functionality is demonstrated using a ten plate distillation column model simulated in OpenModelica on a Linux machine, with thermodynamic and property data from Honeywell Unisim on a Windows machine. The data interfacing was done over a TCP/IP network using CORBA. It is found that real-time operation is possible. Still, that network overhead makes up a significant fraction of the running time, posing problems for

faster-than-real-time off-line simulation and optimization.

Motivated by the above research articles, we wish to harness the capacity of OpenModelica to program an embedded system. The rest of the dissertation is organized as follows. Chapter 2 will demonstrate the features of OpenModelica along with its simulation environment. In chapter 3, we will discuss the critical differences between a microprocessor and a microcontroller. In this chapter, we will also discuss Raspberry Pi (R Pi), which is a single-board computer. Chapter 4 will discuss the fundamentals of OPC UA protocol, which is the medium of communication between OpenModelica and an external controller. In chapter 5, we will show how to interface R Pi and OpenModelica. This chapter will program R Pi as a client, which will connect with OpenModelica. Chapter 6 will present how to implement controllers on the client-side to control a plant running in OpenModelica. In this chapter, we have covered quite a few plants being controlled by various controllers. Chapter 7 will present a shell script deployed to automate the simulation of a model in OpenModelica. Chapter 8 will discuss the implementation of OPC UA server in OpenModelica. This chapter will also explore the availability of OPC UA server in other simulation tools. At last, we will conclude this dissertation, followed by the future directions of this project. The code files used in this project are presented in the appendices at the end.

Chapter 2

About OpenModelica

In this project, we will use OpenModelica as a modeling tool. OpenModelica is an open-source implementation of the Modelica language [6]. Modelica language is designed primarily for modeling the dynamic behavior of engineering systems. The Modelica Association ensures that the Modelica language is maintained and constantly improved. The Modelica language has been used in industry since 2000. Some of the Modelica language's major implementations are Dymola, MapleSim, OpenModelica Scicos, Xcos, etc. It substantiates the fact that the Modelica approach is popular in the industry.

Before discussing the features of OpenModelica we should look at the Modelica language in general and how it works. At the onset, this language is strongly typed. It means that the compiler is more likely to generate an error if, for example, the passed type to an argument does not match the expected type. Furthermore, Modelica is equation-based, which means that the code generator can convert equations related to the model to a more suitable form for solving large-scale non-linear equations. Let us see which programming paradigm Modelica follows. We know that there are two paradigms, namely *Declarative* programming and *Imperative* programming. Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow. Thus, programming a declarative language focuses on explaining what the program should accomplish rather than how. On the other hand, Imperative programming is a programming paradigm that uses statements that change a program's state. Modelica is a declarative programming language, which means that the programmer is not supposed to explicitly state how the program operates, unlike an imperative language such as C/C++.

OpenModelica is an open-source implementation of the Modelica language by a consortium of more than 50 members [6]. OpenModelica can be used by large numbers of students and small- and medium-scale chemical companies as it is open-source software. It has a good set of solvers for different mathematical systems: algebraic, differential, and differential-algebraic. Also, OpenModelica comes with a handy debugger. It can provide model-level debugging, indicating at which model equation the problem has occurred, and it can give the incidence matrix of the system being solved. It can point out whether the number of variables is correctly, over, or under specified at compile time. OpenModelica Compiler *aka* OMC translates Modelica to C code, with a symbol table having definitions of variables, functions, and classes. These definitions can be predefined, user-defined, or taken from libraries. OpenModelica comes with 75 libraries in diverse fields, such as hydraulics, power-system simulation, motorcycle dynamics, servomechanisms, and thermal power, with about 1000 models. Apart from this, OpenModelica has a built-in OPC UA server, which is crucial for establishing Machine-to-Machine (M2M) communication. With these features, OpenModelica seems to be a potential candidate for sim-

ulating a wide range of processes or plants. Thus, OpenModelica has been used as a modeling tool in this project.

2.1 Features of OpenModelica

Modelica language is strongly typed. Strongly typed is a concept used to refer to a programming language that enforces strict restrictions on the intermixing of values with different data types. When such restrictions are violated, an error (exception) occurs. OpenModelica being an open-source implementation of the Modelica language [6], will also be strongly typed.

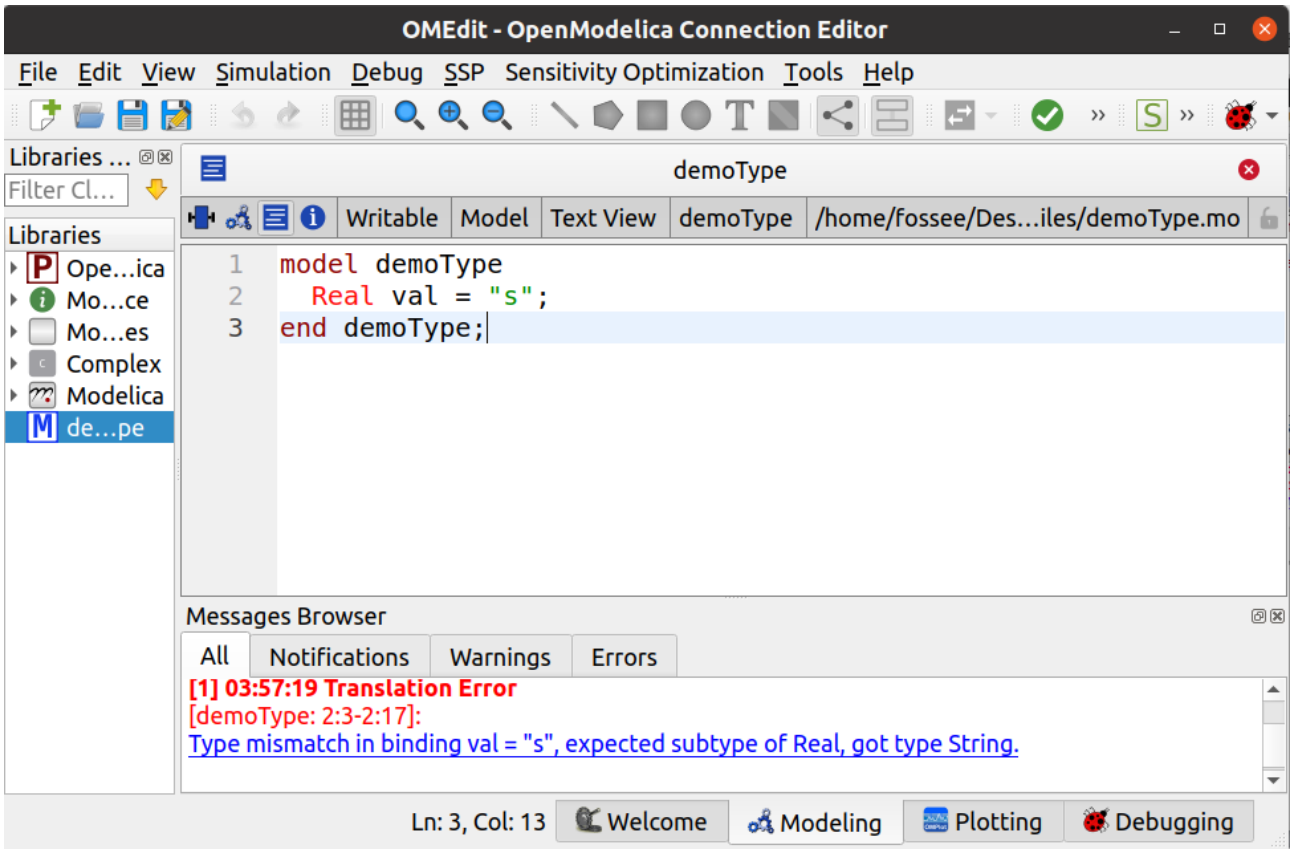

Figure 2.1: Model in OMEdit to show that Modelica is strongly typed

To demonstrate this, we will create a simple model in OpenModelica Connection Editor (OMEdit). We will discuss OMEdit in the upcoming sections. For now, OMEdit is an advanced open-source, user-friendly graphical user interface that provides the users with easy-to-use model creation, connection editing, simulation of models, and plotting of results [10]. Let us look at the model, as shown in Fig. 2.1. In this model, we have declared a variable named `val` of `Real` type. Real is just one of the four built-in types (`Real`, `Integer`, `Boolean`, and `String`) in Modelica. As the name suggests, `Real` is used to represent real-valued variables (which will generally be translated into floating-point representations by a Modelica compiler). In our model, we have declared `Real val = "s";`. It means that we are trying to assign a `String` to a `Real`, which is not acceptable in Modelica language. Accordingly, checking this model in OMEdit throws an error, as shown in the bottom left of Fig. 2.1.

Modelica is equation-based. It means that the code generator can convert equations related to the model to a more suitable form for solving large-scale non-linear equations. While simulating

a system in Modelica, we define the equations. To demonstrate this equation-based concept, we would consider an RLC circuit, as shown in Fig. 2.2.

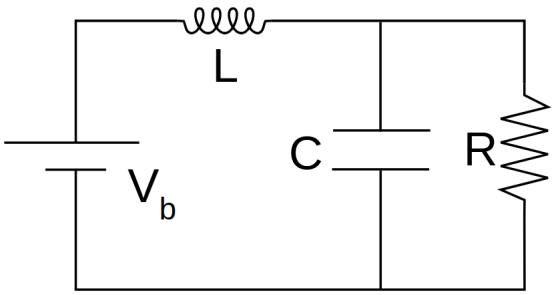

Figure 2.2: Low-pass RLC circuit (filter)

In this circuit, we have an inductor (L), capacitor (C), and resistor (R). Along with these components, we have a voltage source, i.e., battery (V_b), which will drive the circuit. Let us assume the following parameters:

- V - Voltage across the capacitor (C) as time progresses.
- i_L - Current flowing through the inductor (L) as time progresses.
- i_C - Current flowing through the capacitor (C) as time progresses.
- i_R - Current flowing through the resistor (R) as time progresses.

Let us solve the RLC circuit for V, i_L, i_C, and i_R. To solve for each of the currents i.e., i_L, i_C, and i_R, we can use the equations pertinent to L, C, and R respectively:

$$\begin{aligned} V &= i_R \times R \\ L\frac{di_L}{dt} &= V_b - V \\ i_C &= C\frac{dV}{dt} \end{aligned} \tag{2.1}$$

Since we have only 3 equations, but 4 variables (V, i_L, i_C, and i_R), we need one additional equation. For this, we can use Kirchoff's current law to have one more equation as given below:

$$i_L = i_C + i_R \tag{2.2}$$

From equations 2.1 and 2.2, we have four equations and four variables (V, i_L, i_C, and i_R) to solve. Thus, we will create a model (as shown in Fig. 2.3) in OMEdit with all these equations and ask OpenModelica to solve these. From Fig. 2.3, we can observe that there are two sections in this model. First, we declare all the physical types (like Voltage, Current, Resistance, Capacitance, Inductance, etc.) that we will need. Next, we start the equation part, where we write all the mathematical equations, as mentioned in equations 2.1 and 2.2. With these two sections, OpenModelica will solve the equations and help us estimate the behavior of variables over time. We have only mentioned the equations, and we haven't detailed how to solve them. It is the duty of solvers present in OpenModelica to solve these equations efficiently. It is the beauty of the Modelica language. Now, we check the model for any errors. Once the model is successfully checked, we simulate the model (given in Fig. 2.3) for 10 seconds in OMEdit and plot the values of V_b and V. Remember, V_b is the battery voltage, and V is the voltage across the capacitor. Accordingly, we get a plot as given in Fig. 2.4. For the sake of convenience,

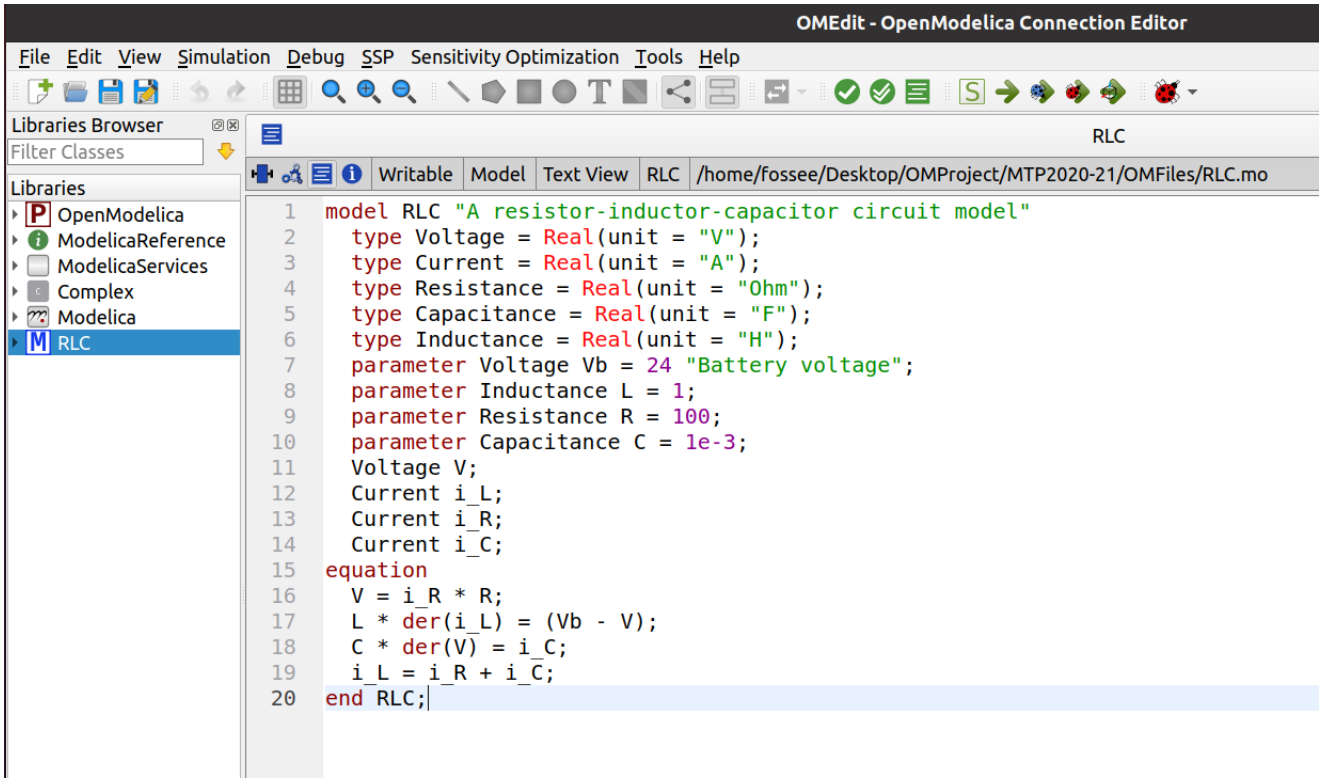

Figure 2.3: Model of RLC circuit in OMEdit

we have plotted only two variables. However, we can plot any of the parameters and variables declared in the model. For that, one can click on the **Plotting** perspective in OMEdit. Next, we need to visit the section **Variable Browser** and select the variables (or parameters) we want to plot. A sectional view of **Variable Browser** is presented in Fig. 2.5.

We have discussed the two important features of Modelica language. First, Modelica is strongly typed, and second, Modelica is equation-based. Another important feature is that Modelica is a declarative programming language. This means that the programmer is not supposed to explicitly state how the program operates, unlike an imperative language such as C/C++. Let us consider the model shown in Fig. 2.3. In this model, we have only mentioned the mathematical equations held in the case of an RLC circuit shown in figure Fig. 2.2. We have not explicitly stated the roadmap which Modelica should follow to solve the equations. So, we are focusing more on *What than on How*. It is the duty of Modelica to find good solvers for the equations and efficiently solve them.

2.2 OpenModelica Connection Editor (OMEdit)

The interactive OpenModelica environment consists of quite a few components like **OMEdit**, **OMNotebook**, **OMShell**, **OMShell-terminal**. When writing this dissertation, we have used the latest official release, 1.17.0 of OpenModelica. A typical installation of OpenModelica results in the automatic installation of components described above, as shown in Fig. 2.6. In this section, we will discuss OMEdit.

OMEdit is an advanced open-source, user-friendly graphical user interface that provides the users with easy-to-use model creation, connection editing, simulation of models, and plotting of results [10]. In short, OMEdit is a graphical user interface for the graphical model, and it

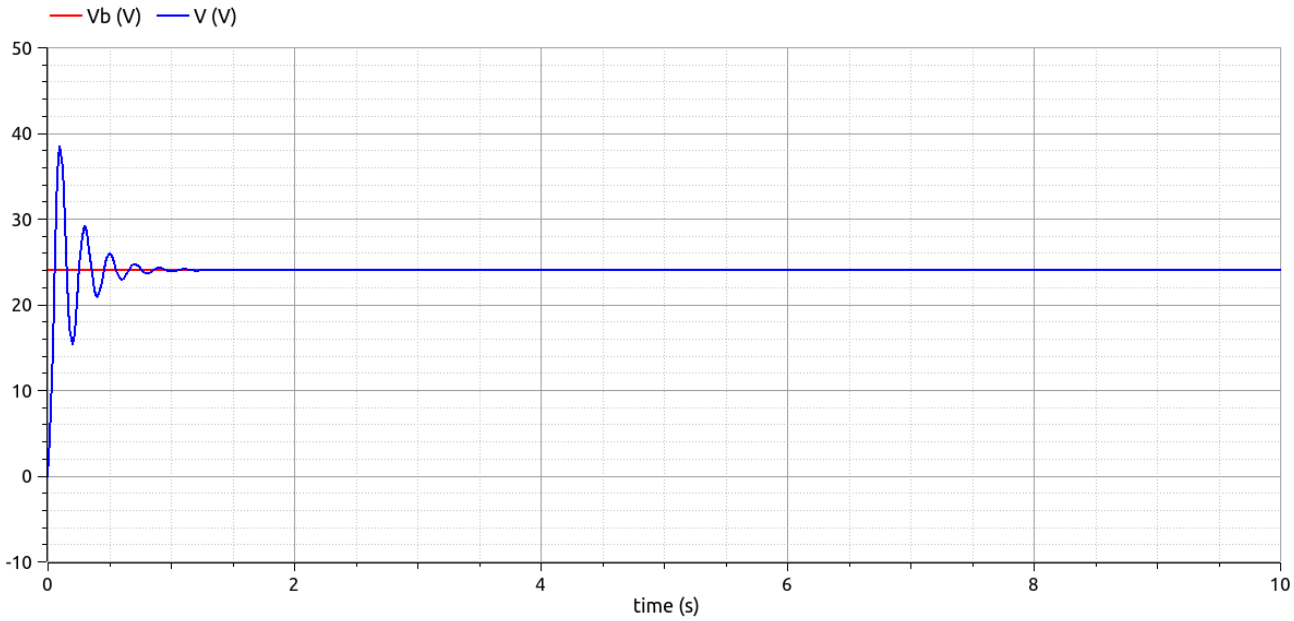

Figure 2.4: Response of capacitor voltage and battery voltage in RLC circuit

has several features to create and modify the models. In Fig. 2.1, we can observe the modeling happening in OMEdit. In the bottom of this figure, we can see that there are four different perspectives named, **Welcome**, **Modeling**, **Plotting**, and **Debugging**. In the bottom right of Fig. 2.7, we can see all of these perspectives. By default, OMEdit gets launched in the Welcome Perspective. We now briefly describe each of these Perspectives, as given below:

1. Welcome Perspective: It shows the list of recent files and the list of the latest news from https://www.openmodelica.org.

2. Modeling Perpective: It provides the interface where users can create and design their models.

3. Plotting Perspective: It shows the simulation results of the models. Plotting Perspective will automatically become active when the simulation of the model is finished successfully.

4. Debugging Perspective: The application automatically switches to Debugging Perspective when the user simulates the class with an algorithmic debugger.

One can see that the current perspective in Fig. 2.1 is **Modeling**. After modeling a system, we first check it and then simulate it. When we click on the Simulation Setup tab in OMEdit, we get access to the **Simulation Setup** window for that particular model. This setup window is as shown in Fig. 2.8. Once we simulate the model, we can go to the **Plotting** perspective to view the required plots. Fig. 2.4 is one of such plots that has been generated after simulating the model given in Fig. 2.3. In this perspective, we also get access to the **Variable Browser**, as shown in Fig. 2.5. There, we can select the variables (or parameters) which we want to plot.

2.3 Modelica Library

Modelica Library is an open-source package. OMEdit automatically loads it for every session. It can be seen in Libraries Browser. Modelica Library has classes from a wide range of domains like mechanical, electrical, thermal, etc. Classes of this library can be referenced and used for designing models as per the requirement. For instance, let us simulate a feedback control system

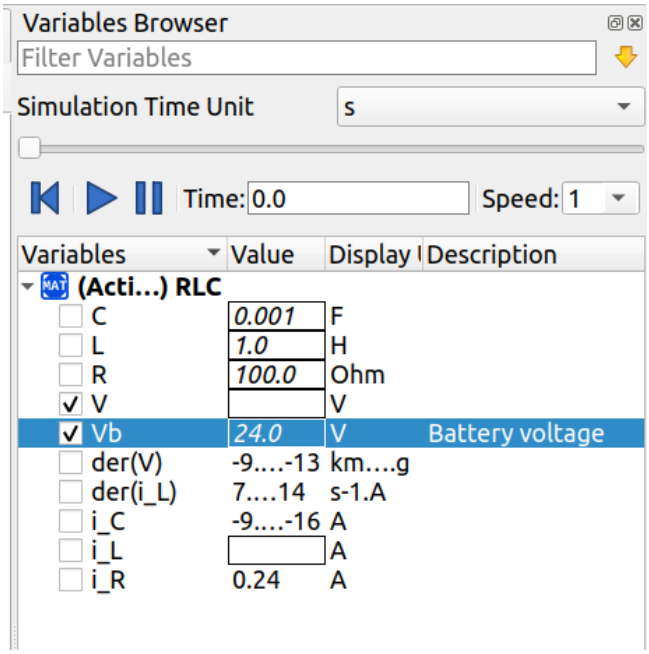

Figure 2.5: Variables Browser in the Plotting perspective of OMEdit

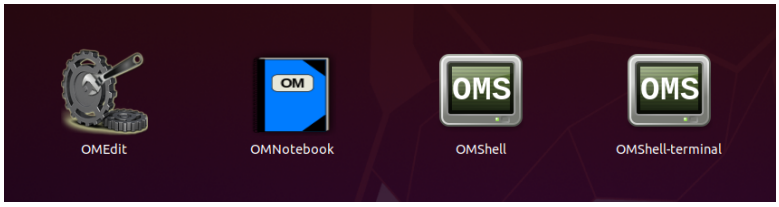

Figure 2.6: Various components of interactive OpenModelica environment

in OMEdit. For this, we will use the components (like Step, PID, SecondOrder, and Feedback) available in **Blocks** package of the Modelica library. For this feedback control system, we will use a second-order system (Sys) as a plant and a PID controller (PID). For the second-order system, we use a plant with the following parameters:

$$\begin{aligned} Sys &= \frac{k \times \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \\ &= \frac{2 \times 1^2}{s^2 + 2 \times 0.7 \times 1} \end{aligned} \tag{2.3}$$

where, k is the gain, ω_n is the angular frequency, and ζ is the damping ratio. Thus, with $k = 2$, $\omega_n = 1$, and $\zeta = 0.7$, we get the transfer function of the second order system as

$$\begin{aligned} Sys &= \frac{2 \times 1^2}{s^2 + 2 \times 0.7 \times 1} \\ &= \frac{2}{s^2 + 1.4s + 1} \end{aligned} \tag{2.4}$$

Next, we use a PID controller with the following parameters:

$$PID = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right) \tag{2.5}$$

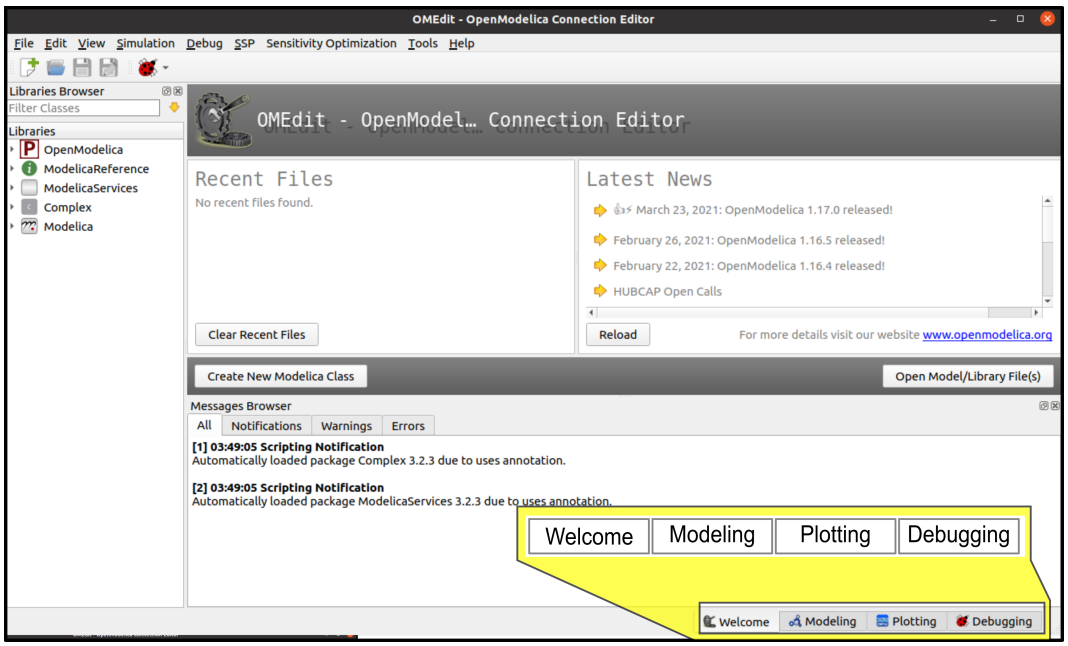

Figure 2.7: User interface of OMEdit

where, T_i is the integral time constant and T_d is the derivative time constant. With $K = 1$, $T_i = 10$, and $T_d = 0$, we get the transfer function of the PID controller as given below:

$$PID = \frac{10s + 1}{10s} \tag{2.6}$$

With this, the designed model is as shown in Fig. 2.9. After simulating this model for 50 seconds, we get the response as given in the Fig. 2.10.

2.4 Command-line Interface of OpenModelica

As discussed in Sec. 2.2, we know that the interactive OpenModelica environment consists of quite a few components which include both UI-based applications and commamnd-line interface based applications. In this section, we discuss the command-line interfaces of OpenModelica as given below:

- **OMShell** - It is an interactive session handler that parses and interprets commands and Modelica expressions for evaluation, simulation, plotting, etc. The session handler also contains simple history facilities and completion of file names and certain identifiers in commands.
- **OMShell-terminal**: It is also an interactive session handler. The main difference between OMShell and OMShell-terminal is that the former is a Qt application, whereas the latter is a shell application.
- **OMPlot**: It is a plotting tool for OM-generated result files.

In the upcoming sections, we will discuss how to use **OMShell-terminal** for simulating a model via Terminal. We will also see how to visualize the OM-generated result files in OMEdit.

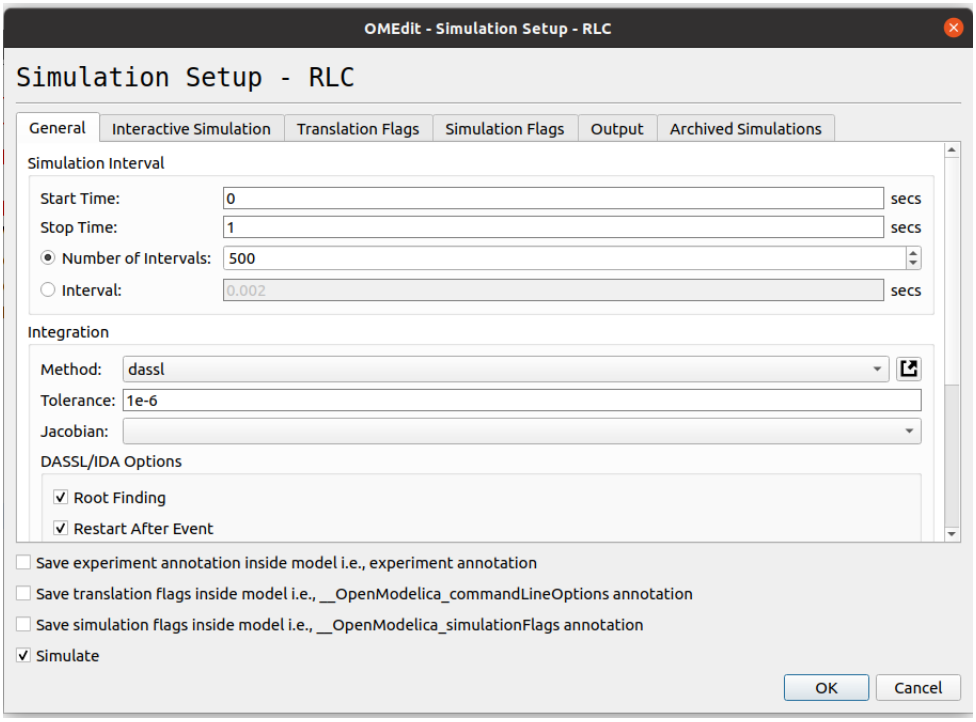

Figure 2.8: View of Simulation Setup in OMEdit

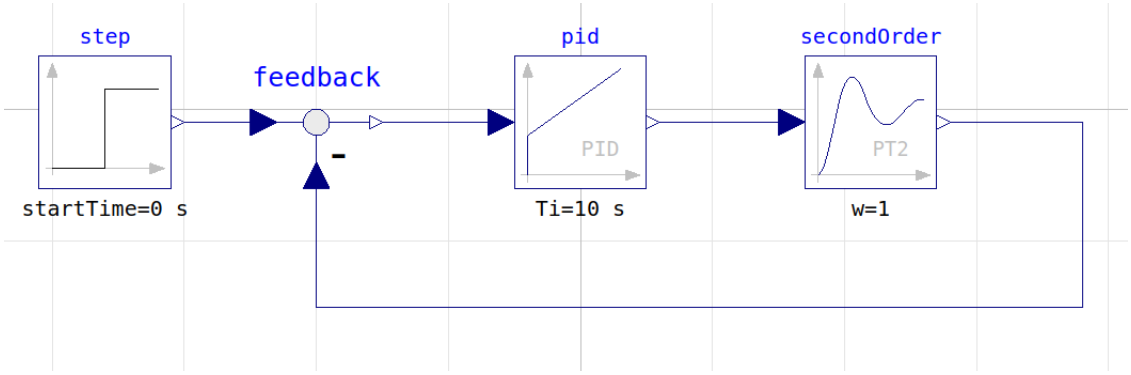

Figure 2.9: Modeling a second order feedback control system in OMEdit

2.5 OPC UA in OpenModelica

OPC UAstands for **O**pen **P**latform **C**ommunications **U**nited **A**rchitecture. OPC UA is a protocol for industrial communication and has been standardized in the IEC62541. OPC UA defines a set of services to interact with a server-side object-oriented information model. In this project, we will use OpenModelica as a server. The client/server interaction is mapped to a binary encoding and TCP-based transmission. Currently, OPC UA is marketed as the one standard for non-real-time industrial communication, but the server's real-time implementation is in the research phase right now. In OPC UA, all communication is based on service calls, each consisting of a request and a response message. The information model in each OPC UA server is a web of interconnected nodes. Objects in models are accessed through their node IDs.

Current design of OpenModelica supports OPC UA as one of the interfaces, and it is very much capable of handling a large amount of data effectively. OPC UA has similar properties to OpenModelica, such as its open-source variants are available, and it is platform-independent, so one application made for one kind of system will run on all other systems independent of its hardware and system software. Remember, this project requires two separate systems. One is

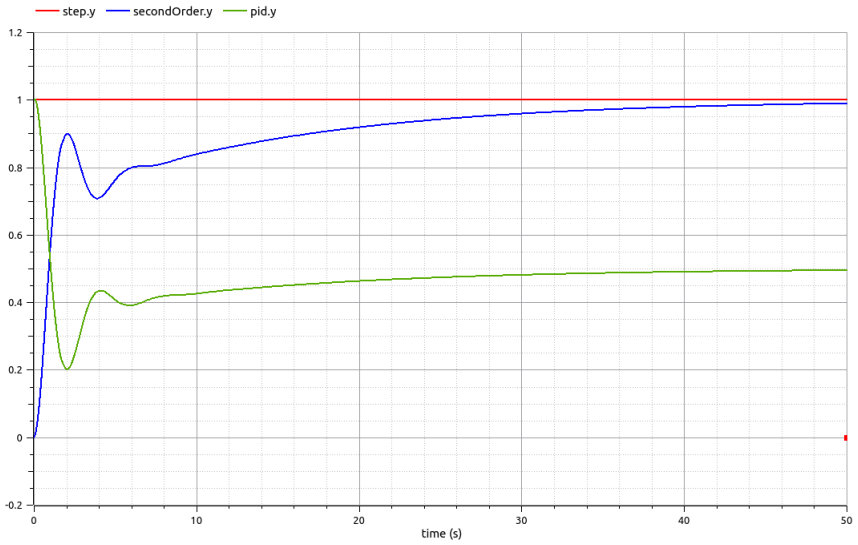

Figure 2.10: Response of a second order feedback control system

OpenModelica in which we would simulate a digital twin of the actual plant being studied. The second system would be an R Pi that would monitor the plant's states and provide feedback to the plant to provide the desired results. So, we need to have a communication protocol so that the two systems could talk to each other. For this, we will use the OPC UA protocol. As OpenModelica has a built-in OPC UA server, we will build an OPC UA client to communicate with the process running in OpenModelica. This OPC UA client would run on the R Pi side. For this project, we are using the Python OPC UA package to develop the client-side. In the upcoming sections, we will discuss the address space of OPC UA and the ways to connect a client with a server.

Chapter 3

About Raspberry Pi

In this era of electronics, we often come across the word *processors*. For example, we keep on discussing which processor your laptop does have. One of the latest processors is the Intel Core i7, which is quite prevalent these days. Today, most laptops are powered by an Intel Central Processing Unit (CPU), though several models use AMD processors. That's why a processor is indispensable to a laptop (or a CPU). So all the laptops have a processor or microprocessor, which works as the brain of the computer. We can relate this brain with a human's brain. So, we know that a human's brain is the command center for the human nervous system. It receives signals from the body's sensory organs and outputs information to the muscles. However, a brain is competent only when most of the body parts coordinate with the brain. Thus, a brain does not hold any significance without the parts of the body. Similarly, microprocessors need to connect with a lot of peripherals to perform their tasks. Overall, it contains a CPU, but many other parts are required in order to work with the CPU to complete a process. These all other parts like RAM, ROM, etc., are connected externally.

In addition to microprocessors, we hear about microcontrollers. In most academic projects, we tend to use a microcontroller, not a microprocessor. So, we need to know what differentiates a microcontroller from a microprocessor. In the case of a microprocessor, we know that it consists of only a CPU. However, a microcontroller has a CPU, in addition to a fixed amount of RAM, ROM, and other peripherals, all embedded on a single chip. That's why, while designing an embedded system, we prefer to use microcontrollers. Arduino Uno is a microcontroller board based on the ATmega328P. Even though microcontrollers come in handy while designing embedded systems, they have some limitations. For instance, a microcontroller cannot interface with a higher power device directly. Moreover, it can only perform a limited number of executions simultaneously. However, due to the advent of IoT, CPS, and Industry 4.0; we are looking for systems capable of running many tasks and that too in real-time. Microprocessors definitely can handle this load, but they will require interfacing with other necessary peripherals like RAM, ROM, etc. That's why the embedded enthusiasts end up using a single board computer.

When it comes to single-board computers, R Pi is the undisputed heavyweight champion of the world, with more than 30 million units sold. As this project is exploiting HIL's concept, we need to have a robust system that must be able to control a plant running in OpenModelica. That's why we have decided to use R Pi. It would be a physical system that would monitor the plant's states and provide some feedback to the plant so that the plant can provide the desired results. In this project, we will use the R Pi 3 Model B+. In the upcoming sections, we will discuss why this particular model is being deployed. We have seen that R Pi is a single-board computer. However, as it turns out, there is no single computer called the R Pi! R Pi refers to

a series of computers produced by the Raspberry Pi Foundation. Depending on the project's requirements at hand, we need to figure out which Pi is suitable for us? There are more than ten models of the R Pi currently available [11]. So, we need to pick one of these available models for the project.

3.1 Selecting the right Pi

Some of the important aspects which we usually keep in mind while selecting a Pi are as given below:

- **Speed**: It is the processing power of the system. The speed of R Pi models varies from 700MHz (R Pi Model A+ and R Pi Model B+) to 1800MHz (R Pi 400). As far as the number of cores is concerned, all the models(except Pi B, Pi A+, Pi Zero, Pi Zero W, and Pi Zero WH) have four cores. For our project, we will select a Pi with 1.4GHz speed.
- **Memory**: It means how much RAM and ROM we require for the project. Memory is a crucial thing, especially when we have to run large programs. Starting from 512MB, R Pi offers models with RAM up to 8GB (R Pi 4 Model B). For our project, we will select a Pi with 1GB RAM.
- **Size** and **weight**: It is the physical size and weight of the computing system. For most users, this requirement may not apply because all Raspberry computers are already tiny and light (up to 45-50g). In our project, we are not having any constraints on size and weight.
- **Input-Output** (I/O): It means how much I/O support is available. The recent Pi models have 40 general-purpose input/output (GPIO) pins. In our project, we don't intend to connect any sensors or actuators with the Pi. Thus, I/O is not a concern for the project at hand.
- **Cost**: It refers to the financial cost of the system. Pi computers' prices vary from $5 (R Pi Zero) to $70 (R Pi 400).

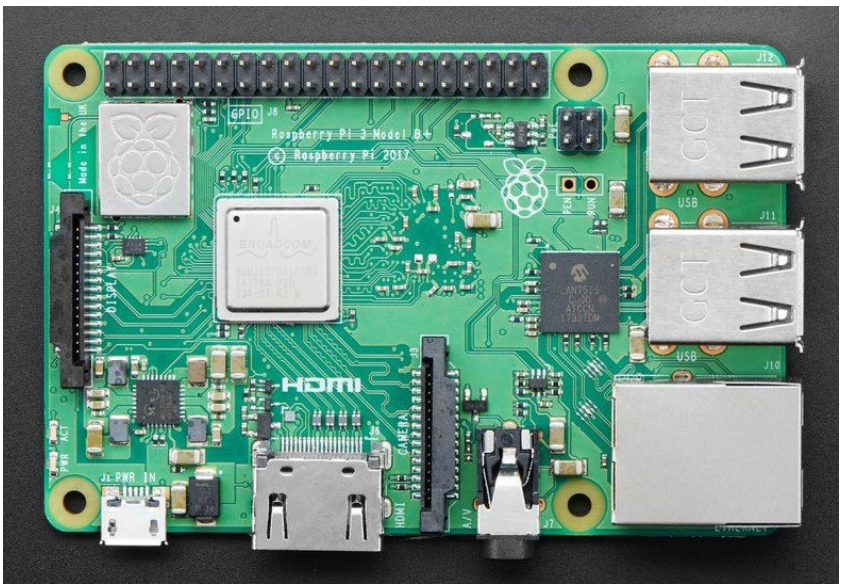

Figure 3.1: Raspberry Pi 3 Model B+

3.2 Pi 3 versus Pi 4

Based on the project's requirements, we have selected to work with R Pi 3 Model B+, as shown in Fig. 3.1. The technical specifications of this model are as given below:

- **Speed**: 1.4GHz
- **Memory**: 1.4GB RAM
- **Size**: 85.6mm × 56.5mm × 17mm
- **Weight**: 46g
- **Number of GPIO pins**: 40
- **Cost**: $35

We can see that the cost of R Pi 3 Model B+ - 1GB RAM is $35, which evaluates to INR 2600 (as of June 18, 2021). If we explore the other Pi models, we find that the cost of Raspberry Pi 4 Model B - 2 GB RAM is around INR 2700. Given that Pi 3 Model B+ and Pi 4 Model B costs the same, one can argue that we should consider buying the latter model of Pi. However, we need to understand the differences in the technical specifications of these two models. Table 3.1 summarizes a comparison of these two models' technical specifications.

Product Details	**Pi 3 Model B+**	**Pi 4 Model B**
Speed	1.4GHz	1.5GHz
Memory	1GB RAM	2GB RAM
Number of cores	4	4
USB	4x USB2.0	2x USB3.0 + 2x USB2.0
Number of GPIO pins	40	40
Ethernet	1000Base-T	1000Base-T
Wireless	802.11ac/n	802.11ac/n
Bluetooth	4.2	5.0
HDMI port	Yes	2x micro HDMI
Energy consumption	5V / 2.4A DC	5.1V / 3.0A DC

Table 3.1: Technical Differences between Pi 3 Model B+ and Pi 4 Model B

From Table 3.1, we can observe that all the specifications, except the last two (HDMI port and Energy consumption), are almost similar. When it comes to the HDMI port, Pi 3 Model B+ has a normal HDMI port that is compatible with the HDMI port of most modern TVs and computer monitors. In other words, Raspberry Pi 3 has a single full-size HDMI port, as shown in the Fig. 3.2. So we can connect them to a screen using a standard HDMI to HDMI cable. On the other hand, R Pi 4 has two micro HDMI ports. It allows us to connect two separate monitors. However, since it has micro HDMI ports, we will need either a micro HDMI to HDMI cable, or a standard HDMI to HDMI cable plus a micro HDMI to HDMI adapter, to connect R Pi 4 to a screen. The connector or the adapter is as shown in Fig. 3.3.

Today, most computers don't have a micro HDMI port. Thus, while using R Pi 4, we will require HDMI to the micro-HDMI adapter. To avoid the need for extra paraphernalia, we prefer to use Raspberry Pi 3 Model B+.

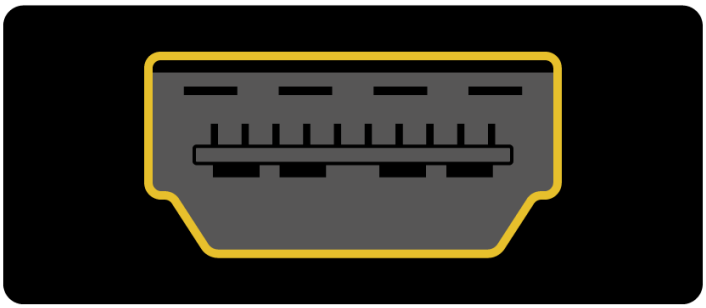

Figure 3.2: HDMI port available in Raspberry Pi 3 Model B+
Source: Raspberry Pi Foundation
https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up/1

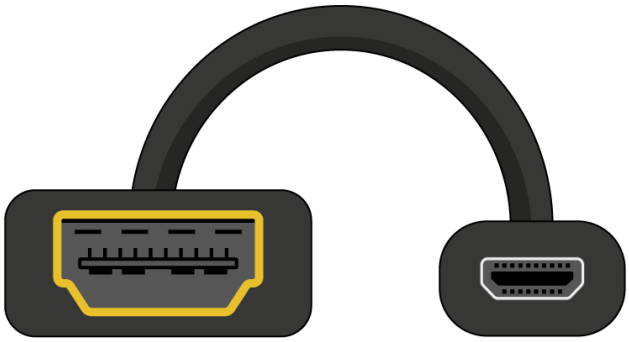

Figure 3.3: Adapter for connecting micro HDMI to HDMI cable
Source: Raspberry Pi Foundation
https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up/1

From Table 3.1, we can observe that Raspberry Pi 3 requires a power supply that provides at least 2.5A. However, Raspberry Pi 4 requires a power supply that provides at least 3.0A. Thus, we can infer that Pi 4 requires high power as compared to Pi 3. Moreover, Pi 3 uses a normal micro USB port for power. This port is the same as that we have on our Android phones. So, we can use the chargers of our Android phones to power Pi 3. On the other hand, R Pi 4 uses a USB-C port for power. Owing to the two reasons stated above, we have decided to use Pi 3 Model B+ for the project at hand.

3.3 Getting Started with Pi 3

Here, we will discuss how to start using a Pi 3. First, we will list down the things which we need to get started:

- One R Pi computer (Pi 3)
- A power supply (5V / 2.4A)
- A micro-SD card: R Pi needs an SD card to store all its files and the R Pi OS operating system.
- A keyboard and a mouse

- A TV or computer screen along with its own power supply

After procuring all these components in place, we will set up our SD card. It means that we will store the R Pi operating system (OS) on the card. The procedure to store the OS on the card is available on the Raspberry Pi foundation website. However, these days, we can buy an SD card which already has NOOBS on it. NOOBS – New Out Of the Box Software – is an alternative straightforward way to install an operating system. In our case, we have purchased a pre-installed NOOBS SD card. Now, we have gathered all the hardware components and the required OS for setting up the Pi. Now, we need to connect your Raspberry Pi. The instructions to do so have been provided on https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up.

Chapter 4

What is OPC UA?

For this project, we will use OPC UA (Open Platform Communications United Architecture) protocol for establishing communication between R Pi and OpenModelica. We know that OpenModelica provides an inbuilt OPC UA interface. OPC UA is an industrial Machine to Machine (M2M) communication protocol developed by the OPC Foundation. It is the successor to Open Platform Communications (OPC) – DA (Data Access) and legacy. Although developed by the same organization, OPC UA differs significantly from its predecessor.

4.1 OPC UA versus OPC-DA

OPC specifies the communication between the control devices by a different manufacturer in real-time. Initially, the requirement to form a typical bridge for various Windows-based software applications and process hardware gave rise to OPC. The protocols that are included in the classical OPC model are DA (Data Access), AE (Alarm and Events), HAD (Historical Data Access), XML DA (XML Data Access), AND DX (Data Exchange). On the other hand, there is OPC UA (Unified Architecture).

OPC-DA (Data Access) is the most basic protocol of OPC. The representation of the data for DA is as follows: first the Value, i.e., the data itself along with Name comes, and to that other information comes along with it such as Timestamp that provides the exact time when the value was read, and at last, the information regarding the validity of data which is called Quality comes.

For automation in industries, a protocol called OPC Unified Architecture (OPC UA) is developed by the OPC Foundation that was mainly focused on the communication between machines, i.e., machine to machine communication. The features that UA supports are:

- It is a cross-platform protocol, and that's why it does not depend on any particular operating system or programming language.
- The architecture of OPC UA is service-oriented.
- The security provided by OPC UA is robust.

The basic difference between OPC-DA and OPC UA is the version. The older version of OPC is DA, where it supports the older version of data modeling, which is not as great as the data modeling provided by OPC UA. Also, the transfer of the information between the server and client in OPC-DA is only VQT that stands for Value Quantity and Time. But on the other hand, OPC UA, which is the new version of OPC, provides data and information modeling and

many properties that can be shared between the client and server about a variable.

As OPC-DA comes under OPC classic model; it supports Distributed Component Object Model (DCOM) communication for connecting the client and server, where DCOM is dependent on the operating system and supports only Window OS. And on the other hand, OPC UA does not rely on DCOM communication for connecting clients and servers. Thus it is a platform or OS independent, and it supports platforms such as Linux (Java), Apple, or Windows.

OPC-DA allows access to only the current data and is incapable of generating alarms and historical events. In contrast, OPC UA supports features like historical events, multiple hierarchies, and provides methods and programs (that are called commands). One of the limitations of OPC-DA is its inadequate security. Security is the major issue in today's world because some sophisticated viruses and malware more frequently attack systems. This security issue is solved in a higher version of OPC UA. An OPC-DA specification defines how real-time data can be exchanged between a data source and a data sink (for example, a PLC and an HMI) without either of them having to know each other's native protocol. An OPC UA specification states how real-time data is communicated between machine to machine for industrial automation.

4.2 Key components of OPC UA

When communicating over an OPC UA protocol, we need to have both client and server connected on the same network. Fig. 4.1 shows a block diagram of client-server communication in OPC UA protocol. As we can see, OPC UA is a Client/Server based communication. It means

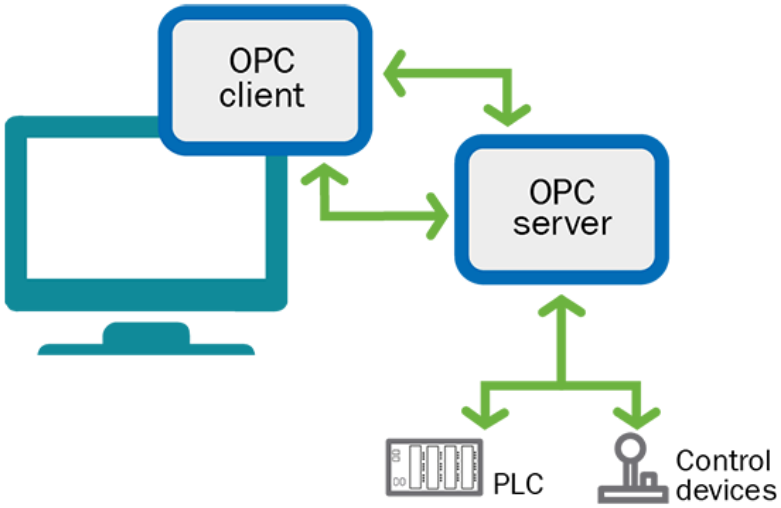

Figure 4.1: Communication in an OPC UA protocol
Source: NOVOTEK
https:
//www.novotek.com/uk/solutions/kepware-communication-platform/opc-and-opc-ua-explained/

that we have one or more servers that wait for several Clients to make requests.

4.2.1 OPC UA server

All industrial Servers provide the physical interface to the real world. Servers measure physical properties, indicate status, initiate physical actions, and do all sorts of physical measurements and activations in the real world under a remote Client device's direction. Servers are where the physical world meets the digital world [12].

The specific capabilities of an OPC UA Server are described by the Profile it supports. A Profile indicates to other devices (electronically) and people (human-readable form) what specific features of the OPC UA specification are supported. Engineers can determine from the Profile if this device is suitable for an application. A Client device can interrogate the Server and decide whether it is compatible with the Client and its application and initiate the connection process with the device. An OPC UA Server announces its availability to interested Client devices. It provides a list of its capabilities and functionality to interested Clients. It gives notifications of different kinds of events. It executes small pieces of logic called methods, and it provides address space information in bulk to Clients (Query service), it includes browsing services so that a Client can walk through its address space. It can allow Clients to modify the node structure of its address space.

An OPC UA Server models data, information, processes, and systems as Object and presents those Objects to Clients in ways that are useful to vastly different types of Client applications. Better yet, the UA Server provides sophisticated services that the Client can use, including:

- Discovery Services – services that Clients can use to know what Objects are available, how they are linked to other Objects, what kind of data and what type is available, what meta-data is available to be used to organize, classify and describe those Objects and Values.
- Subscription services – services that the Clients can use to identify what kind of data is available for notifications. Services that Clients can use to decide how little, how much, and when they wish to be notified about changes, not only to data values but to the meta-data and structure of Objects.
- Query Services – services that deliver bulk data to a Client like historical data for a data value.
- Node Services – services that Clients can use to create, delete, and modify the structure of the data maintained by the Server.
- Method Services – services that the Clients can use to make function calls associated with Objects.

Unlike the standard industrial protocols, an OPC UA Server is a data engine that gathers information and presents it in ways useful to various types of OPC UA Client devices. Devices that could be located on the factory floor like an HMI, a proprietary control program like a recipe manager or a database, dashboard, or sophisticated analytics program might be located on an Enterprise Server.

4.2.2 OPC UA client

There is a controlling device in most industrial networking technologies: a device that connects to and controls one or more end devices. In OPC UA, a device of this type is known as an OPC UA Client. Like controlling devices in these other technologies, an OPC UA Client device

sends message packets to Server devices and receives responses from its Server devices. But beyond this basic functionality, an OPC UA Client device is fundamentally more sophisticated than controllers in other technologies.

Following are some of the important concepts that are important while analyzing OPC UA Clients:

- Client devices request services from OPC UA Server devices. Server devices send response messages and notifications to the OPC UA Client device.
- The Subscription Service Set, which drives notifications, and the Read Service of the Attribute Service Set are the primary services OPC UA Clients use to interact with the address space on an OPC UA Server.
- Clients find OPC UA Server devices in multiple ways. Clients can find Servers using traditional configuration, by using a Local Discovery Server, by using a Local Discovery Server with a Multicast Extension, or by using a Global Discovery Server.
- Once a Client finds a Server, it obtains the list of available endpoints and selects an endpoint that supports the security profile and transport that matches its application requirements.
- Clients begin accessing an OPC UA Server by creating a channel, a long term, a connection between it and an OPC UA Server. Channels are the authenticated connections between two devices.
- Once the channel is established, Clients create sessions, long term, logical connections between OPC UA applications. A session is the authorized connection between the Client's application and the Server's address space.
- Clients can subscribe to data value changes, alarm conditions, and any results from programs executed by Servers. Servers publish notifications back to the Client when those items are triggered.
- Clients invoke methods, which are small program segments. Programs can return results to the Client in the Method call or a Notification if the Client subscribes to it.

This project involves the programming of embedded systems using open-source tools. For this, we will have a digital twin of a plant/process running in OpenModelica on a computer. This system will be treated as an OPC UA server. Next, we will implement an OPC UA client on external hardware (Pi 3), which will connect with the server. Accordingly, this client will monitor and (or) control the plant running in OpenModelica and connected to the OPC UA server. Though we will design our own OPC UA client in Python programming language, we will also use one general-purpose test client named UaExpert.

4.3 General Purpose OPC UA test clients

This section will discuss UaExpert [13], which is designed as a general-purpose test client supporting OPC UA features like DataAccess, Alarms & Conditions, Historical Access, and calling of UA Methods. The UaExpert is a cross-platform OPC UA test client programmed in C++. It uses the sophisticated GUI library QT from Nokia (formerly Trolltech), forming the basic framework that Plugins offers.

In the upcoming sections, we will discuss using UaExpert as a client to a server.

Chapter 5

Interfacing Raspberry Pi and OpenModelica

In this chapter, we will discuss how to establish client-server communication using the OPC UA protocol. As we know that OpenModelica has a built-in OPC UA server, we need to design an OPC UA client. For this, we will use the Python implementation of OPC UA. So, we need to install the OPC UA package [14] on our machine. For this project, we will use Pi 3 Model B+, Linux Ubuntu 18.04 as an operating system, Python3, and OpenModelica 1.17.0. For the sake of convenience, we have tested OPC UA on Linux OS first. Then, we switch to replicating the same over Pi 3. Thus, we will discuss how to install the OPC UA package on a Linux OS. The same method can be extended for Pi 3 as well.

5.1 Installing Python OPC UA on Linux OS

To install OPC UA, we can use `pip`, which is the package installer for Python. On Linux OS, `pip` is used for Python version < 3. On the other hand, `pip3` is used when we want to install packages for Python version ≥ 3. We may have both Python 2 and Python 3 installed on our machine. Accordingly, we need to use `pip` or `pip3`. However, installing a package via `pip3` does not guarantee that the package is installed in the library of Python 3 or the latest version of Python on your machine. That's why we need to check which path `pip` or `pip3` are pointing to on our machine. To check this, we can use the `show` function, as demonstrated in Fig. 5.1.

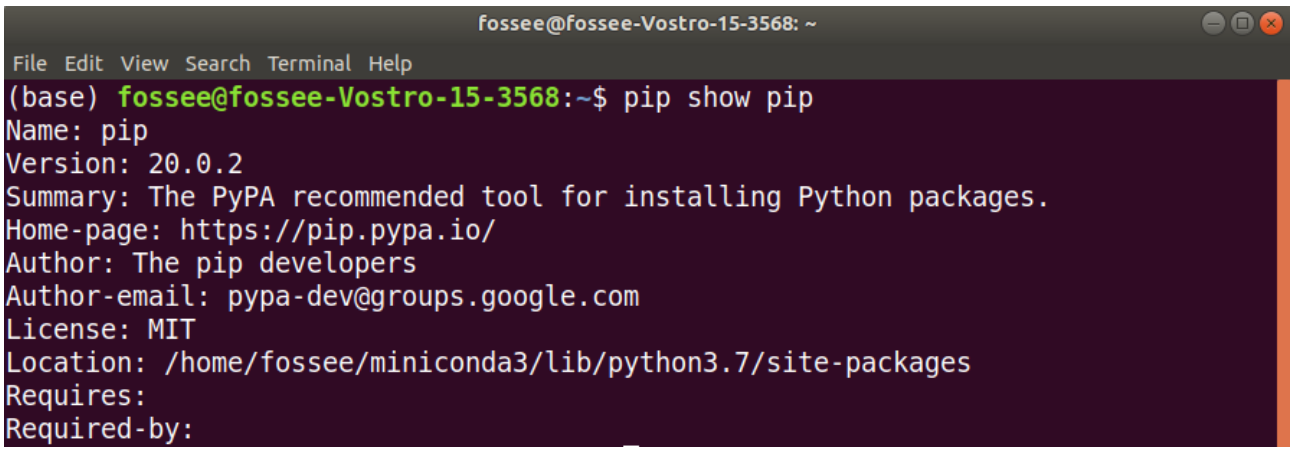

Figure 5.1: Shell command to check the path pip is pointing to

From Fig. 5.1, we can observe that `pip` is pointing towards `python3.7`. Next, we will check which path `pip3` is pointing to, as shown in Fig. 5.2. From Fig. 5.2, we can observe that `pip3`

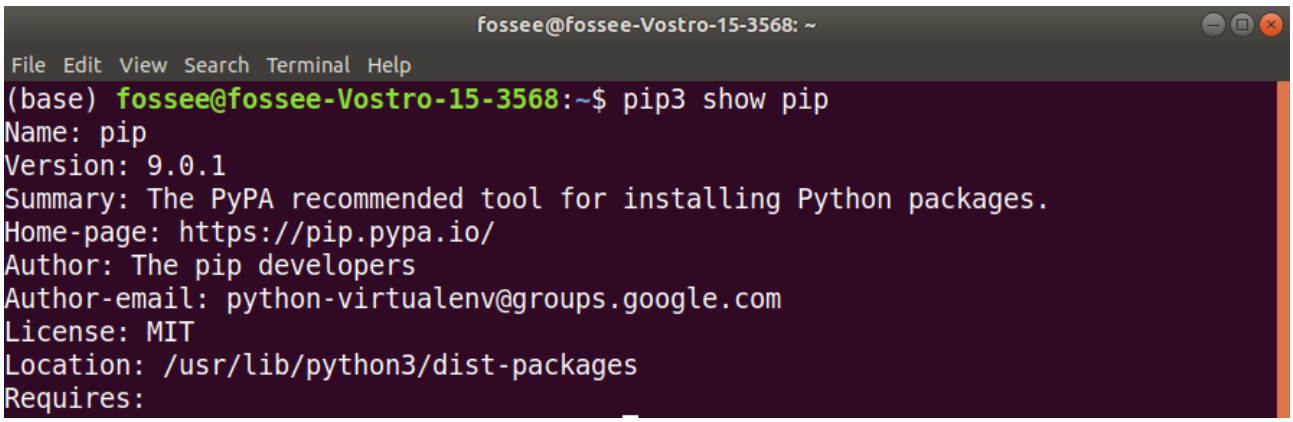

Figure 5.2: Shell command to check the path pip3 is pointing to

is pointing towards `python3`. One can infer that `pip` (not `pip3`) is pointing towards the latest version of Python. Thus, we will use `pip` to install the package OPC UA, as shown in Fig. 5.3. Please note that a stable Internet connection is required for installing this package.

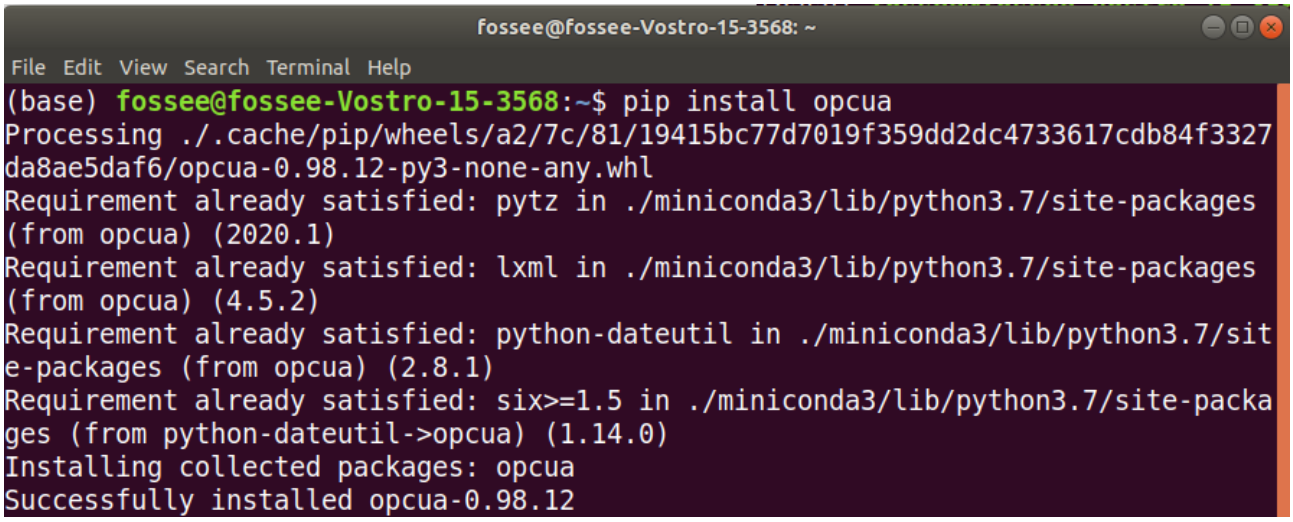

Figure 5.3: Shell command to install OPC UA

5.2 How to establish client-server communication

After installing the Python OPC UA package, we can proceed with designing an OPC UA client to connect to the built-in OPC UA server of OpenModelica. Before doing so, we will execute a sample client-server communication in OPC UA. We implement the server to connect to an IP address and store three random variables (say, Temperature, Pressure, and time). Fig. 5.4 shows the simulation of the server. Accordingly, we will implement the client in such a way that it connects with the server on the same IP address and reads the three variables stored at the server. First, we will use a general-purpose UI based client application named **UaExpert** to connect to the server. To do so, we will first start **UaExpert**, and then, we will connect to the server as a client from UaExpert. After adding the server with its IP address in UaExpert, we will connect to the server. Fig. 5.5 shows the connection of the UaExpert client with the server.

In Fig. 5.5, the **Data Access View** shows the values being read at the client end. It can be used to monitor changes of the value attribute of variables. There are different columns in the

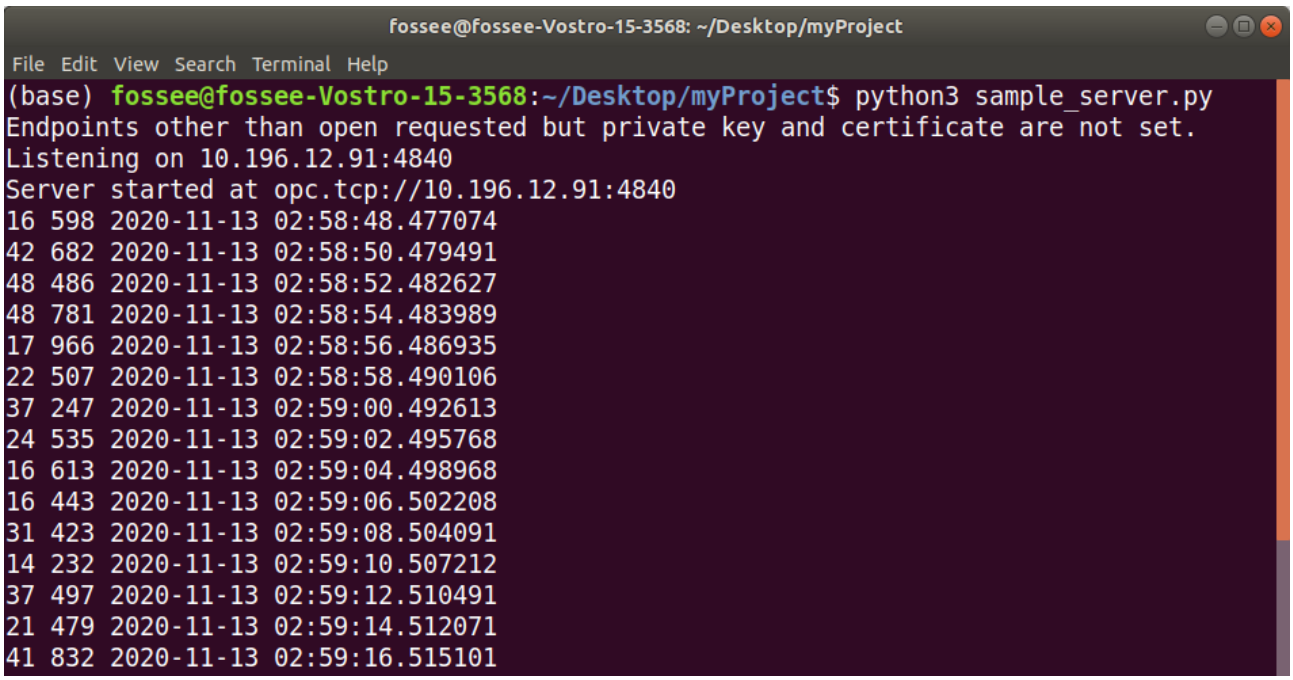

Figure 5.4: Simulation of an OPC UA server

Data Access View, namely, **Server**, **Node Id**, **Display Name**, **Value**, **Datatype**, etc. We now explain the signifinace of these columns in brief:

- **Server** - The Server shows the name of the server to which the client is connected to. We add this name while adding a new Server in UaExpert.
- **Node Id** - It shows the node Ids of the variables that the client has read from the server. These node Ids will be crucial while implementing an OPC UA client in Python.
- **Display Name** - It shows the names of all the three variables being read by the client. These variables are the same, which we have defined in the Server.
- **Value** - It shows the values of the variables being read by the client.
- **Datatype** - As the name indicates, it shows the data type of the variables. Like in Fig. 5.5, the datatype of Pressure and Temperature is `int64`, whereas the datatype of Time is `DateTime`.

Apart from these columns, there are two more columns (not shown in Fig. 5.5): **Server Timestamp** and **Statuscode** in the aforesaid **Data Access View** in UaExpert.

Next, we will implement an OPC UA client in Python and execute it. Accordingly, we can check whether the client is able to read the values stored by the server. Fig. 5.4 shows the server's simulation, whereas Fig. 5.6 shows the connection of a sample client with the server. The Python code for both server and client are available in App. A.1.

5.3 Connecting Pi 3 with OpenModelica

After establishing a sample client-server communication, we will now connect Pi 3 with OpenModelica. For this, we will consider a process or plant and simulate it in OpenModelica. As we want OpenModelica to connect to a server while simulating the plant, we will have to enable

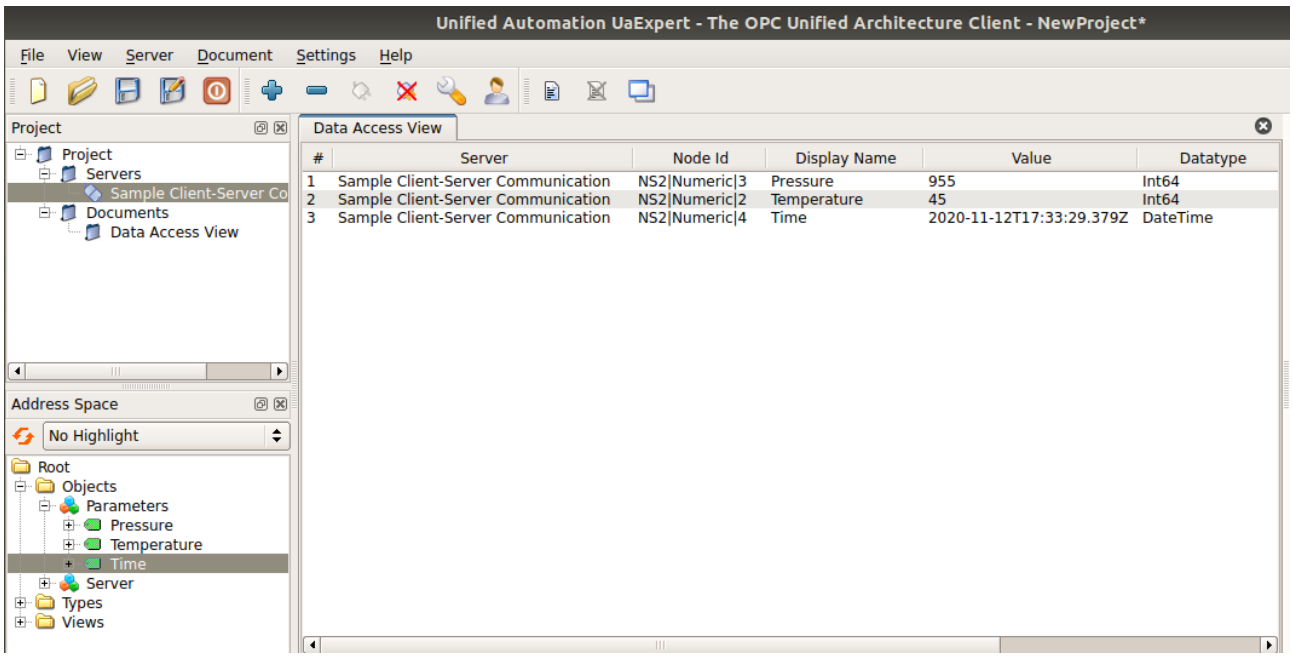

Figure 5.5: Simulation of an OPC UA client in UaExpert

certain simulation flags *aka* simflags.

Let us consider a two-tank liquid level system in which two tanks are arranged so that the first tank's outlet flow is the inlet flow to the second tank. In Fig. 5.7, the outlet flow from tank 1 discharges directly into the atmosphere before spilling into tank 2, and the flow-through R_1 depends only on h_1. The variation in h_2 in tank 2 does not affect the transient response occurring in tank 1. This type of system is referred to as a non-interacting system. First, we will use OpenModelica to model this system in OMEdit. The mathematical model (in OpenModelica) is shown in App. A.2. After this, we will simulate it in OMEdit for 20 seconds. As discussed in Sec. 2.2, we will use the UI-based application OMEdit for simulating this model. While setting up the simulation, we will activate certain simulation flags in OpenModelica so that its built-in OPC UA server is enabled and connected to an IP address. To do so, we will need two simulation Flags in OpenModelica (C-runtime).

- `-embeddedServer=value` or `-embeddedServer` value: Enables an embedded server.
  - none - default, run without embedded server
  - `opc-ua` - run with embedded OPC UA server (TCP port 4841)
- `-rt=value` or `-rt value`: Value specifies the scaling factor for real-time synchronization (0 disables). A value > 1 means the simulation takes a longer time to simulate.

That's why we will enable these two flags in the Simulation Setup while simulating the two-tank model in OMEdit, as shown in Fig. 5.8.

Now, we will implement an OPC UA client, which will connect with this server. We will program the client so that it will connect to the server and fetch all the values available at the server. We know that OpenModelica uses the open62541 library, an open-source OPC UA implementation, when using the simulation flag `-embeddedServer=opc-ua`. The simulation can be controlled by setting variables through the OPC UA interface:

- **OpenModelica.run** Runs asynchronously (possibly synchronizing to real-time after each time step)

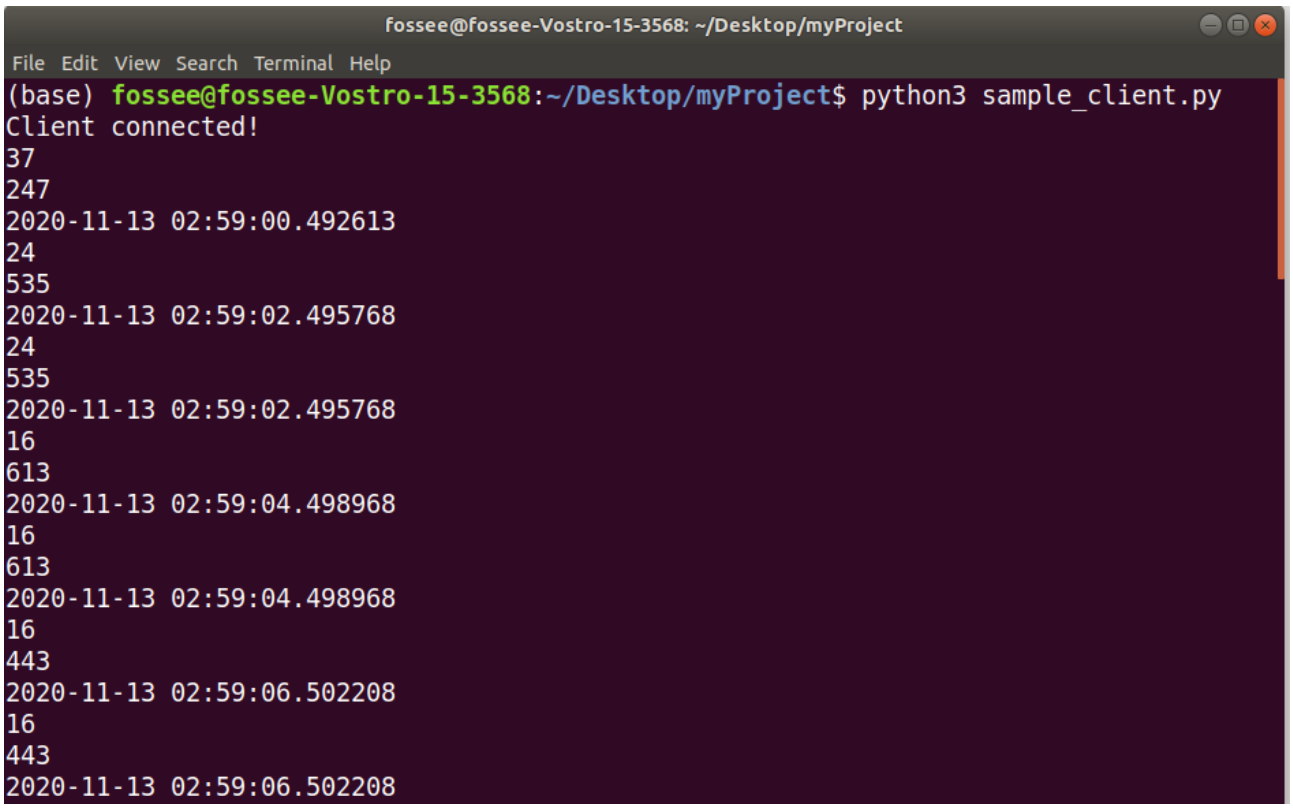

Figure 5.6: Simulation of an OPC UA client in Python

- **OpenModelica.realTimeScalingFactor** 0.0 disables, 1.0 synchronizes in real-time
- **OpenModelica.enableStopTime** When disabled, simulation continues without stop-time

In our OPC UA client, we will want the client to fetch the server's values continuously. So, we will set `enableStopTime` to False. Apart from this, we will set `run` to True, as we would like to monitor the values in real-time. It may be noted that the default value of `enableStopTime` is True, i.e., the simulation will run for a fixed amount of time. While simulating a model in OpenModelica we specify the `stopTime`, which will stop the simulation as the `stopTime` is reached. For practical simulations, we will expect the simulation to stop after a given amount of time. For those simulations, we will not set `enableStopTime` to False. The Python OPC UA client is available in Appendix A.2. After executing the client code, we will be able to get all the values from the server in real-time, as shown in Fig. 5.9. Fig. 5.10 shows the behaviour of the flow rates across the two tanks. Here, Fi denotes the input flow rate to a particular tank and F denotes the output flow rate from the tank.

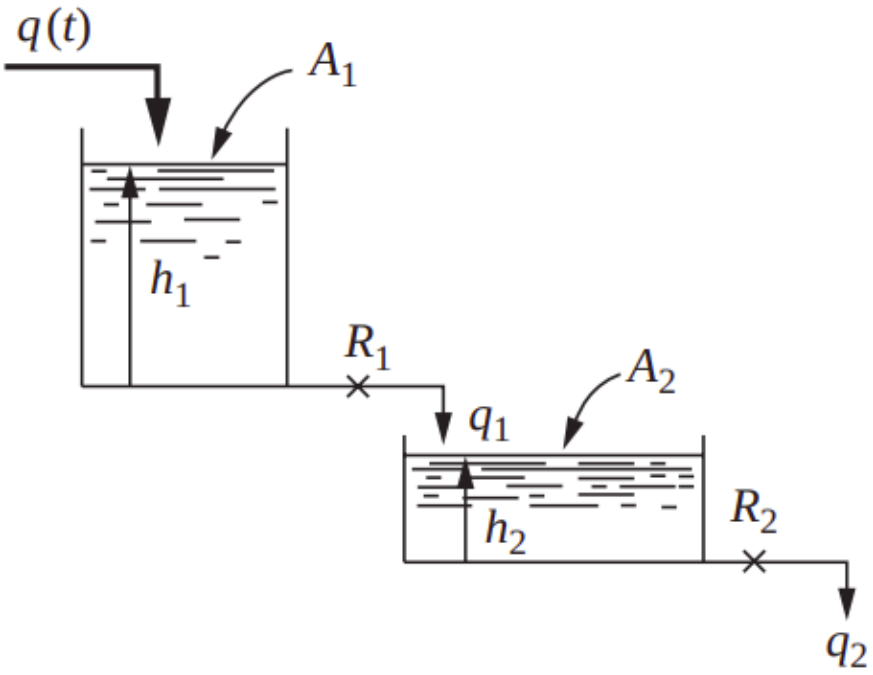

Figure 5.7: Two-tank liquid-level system

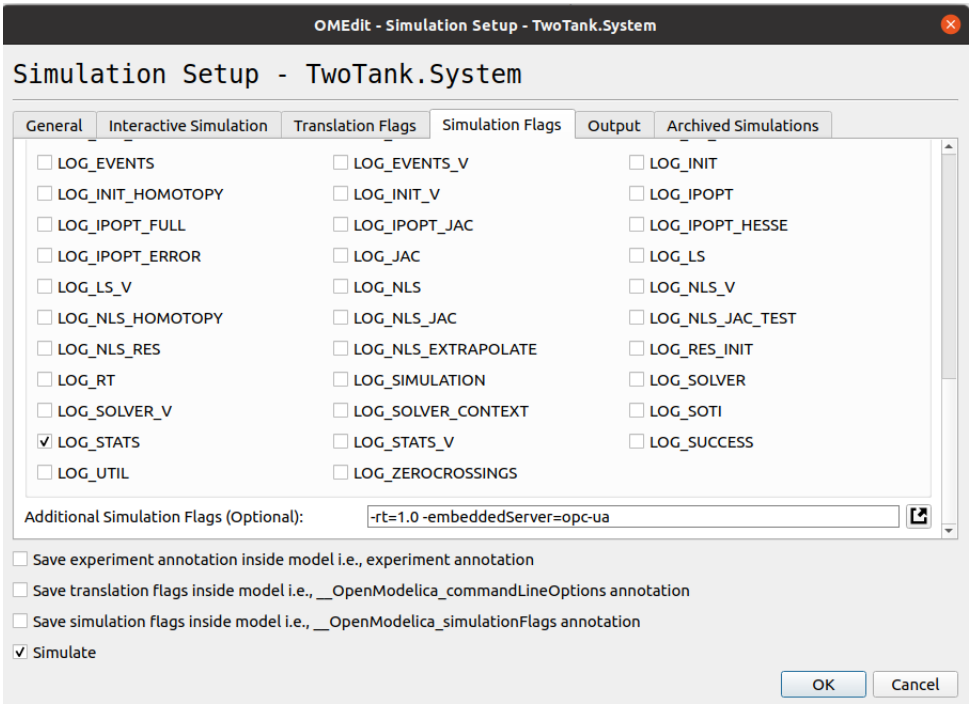

Figure 5.8: Simulation Flags in OMEdit for enabling the OPC UA server

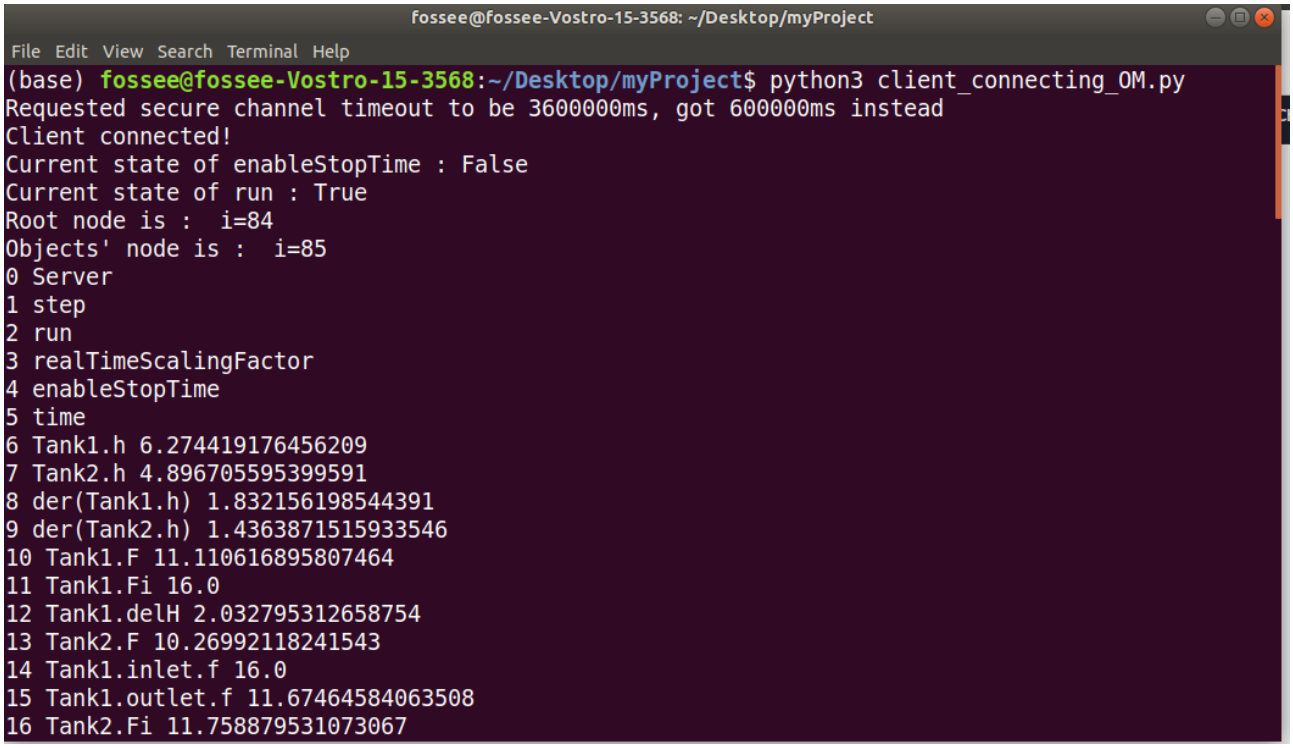

Figure 5.9: Simulation of OPC UA client to read the values of the two-tank system from OpenModelica server

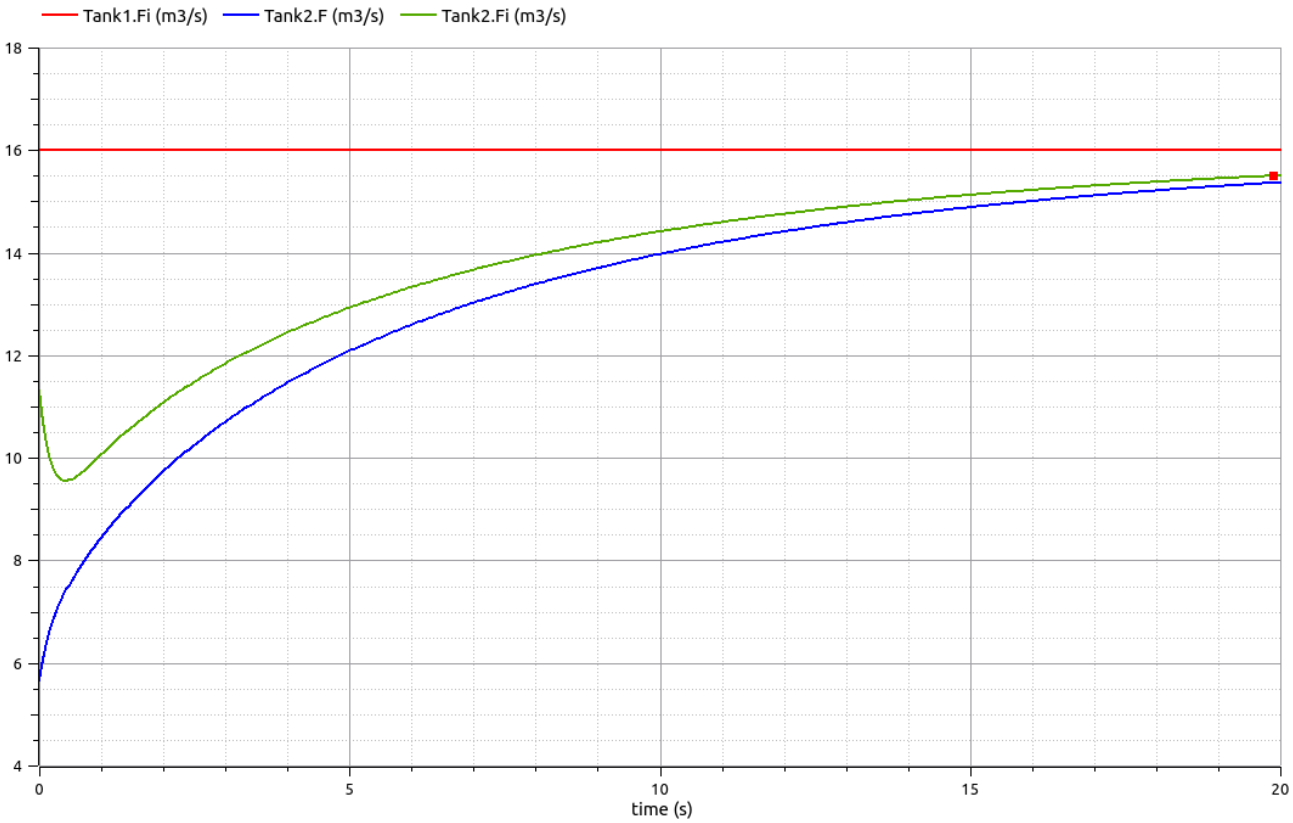

Figure 5.10: Simulation of the two-tank system in OMEdit

Chapter 6

Implementing controllers on client

In the previous chapter, we have discussed how to design an OPC UA client so that it connects to an OPC UA server in real-time. Until now, our client can read the values only. However, as discussed in the previous chapters, we want our client to monitor the plant's states and provide feedback to the plant. That's why we need to implement a controller on the client-side (on the Pi 3) so that we can read and write the values on the OPC UA server.

PID [1] controllers are used in more than 95% of closed-loop industrial processes. That means that PID Controllers are omnipresent. Hence, we will implement one PID Controller on our client using a sample model in OpenModelica. Later, we will discuss how to control other plants. Before that, we need to know how the various components K_P, K_I, and K_D affect the transient response. Table 6.1 summarizes the effect of increasing each of the controller parameters. Please note that NT refers to No Definite Trend.

Response	**Rise Time**	**Overshoot**	**Settling Time**	**Steady-state Error**
K_P	$\downarrow$	$\uparrow$	NT	$\downarrow$
K_I	$\downarrow$	$\uparrow$	$\uparrow$	Eliminate
K_D	NT	$\downarrow$	$\downarrow$	NT

Table 6.1: Effect of PID parameters on a system

6.1 Controlling a sample system

From Table 6.1, we can observe that we need to have a combination of K_P, K_I, and K_D for better control. To understand how a PID controller scores over a simple P controller, we will first implement a P controller on the client side. For the initial testing, we will simulate a plant $y = x$ in OpenModelica. This plant would be controlled by a P-controller running on the client-side. So, the block diagram of the proposed feedback control system would look like Fig. 6.1.

Let us consider a set-point (SP or r) of 10 for the aforesaid model. The client will continuously read the process variable, i.e., PV (y), and calculate the error (e). Accordingly, it will generate a manipulated variable, i.e., MV (u), that will serve as an input to our plant. Thus, MV is given by

$$MV = K_P(SP - PV) \tag{6.1}$$

[1] P: proportional, I: Integral, D: Derivative

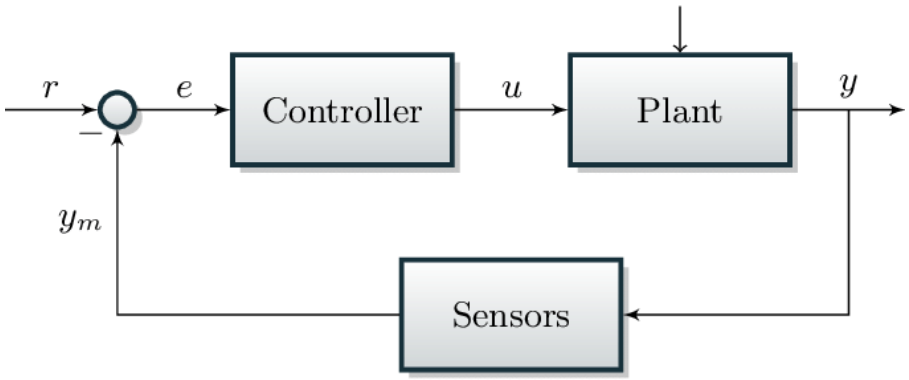

Figure 6.1: Feedback control system

The sample plant in OpenModelica and Python OPC UA client are available in App. B.1. In the sample plant, one can observe that we have only two variables i.e., `x` and `y` and these two are governed by an equation $y = x$. As a part of our control strategy, we want y to achieve the SP of 10. So, we can say that y is the PV and x is the MV which will be fed to the plant. In other words, MV (x) will help the PV (y) to achieve the SP. Thus, our client should perform the following steps:

- Read the value of PV (y).
- Calculate the error i.e., how far PV is from the SP.
- Modify the value of MV (x).

That's why we will only modify the value of x on the OpenModelica server. For this, OpenModelica needs to know that the values of x has to be provided from the simulation environment. Following lines in OpenModelica model help us achieve this:

```
input Real x;
Real y;
```

In the Python OPC UA client, we need to specify the IDs of the variables being defined in OpenModelica. That would let the client read and write certain values from the OpenModelica server. To know the IDs, one may execute the client code, titled OPC UA client connecting to the two-tank model, given in App. A.2. Along with this, from the Python OPC UA client stated in App. B.1, one can observe that we are setting **run** to True after connecting with the server. Following lines of code help us achieve this:

```
run = client.get_node(ua.NodeId(10001, 0))
run.set_value(True)
```

It is required because we want to monitor the plant in real-time. It means that we want to observe the change in values as time progresses. However, we are not changing the state of `enableStopTime`, as we don't want to run the simulation indefinitely. One may recall that the default value of `enableStopTime` is True, i.e., the simulation will run for a fixed amount of time. Also, we need to execute the command for the three steps mentioned above. Following lines in the Python OPC UA client help us achieve these:

```
set_point = 10

while True:
    # Evaluate the PV and MV
```

```
error = set_point - modelicaId[readID].get_value()
valWrite = 0.7 * error
print("Error = {}, valWrite = {}".format(error, valWrite))

if (error > 0):
    modelicaId[writeID].set_value(valWrite)
```

Now, we will first simulate the plant in OpenModelica with the two simulation flags (`-embeddedServer=value` and `-rt=value`). For this, we will use OMShell-terminal, as discussed in Sec. 2.4. Fig. 6.2 shows the commands to be executed in OMShell-terminal for running the simulation.

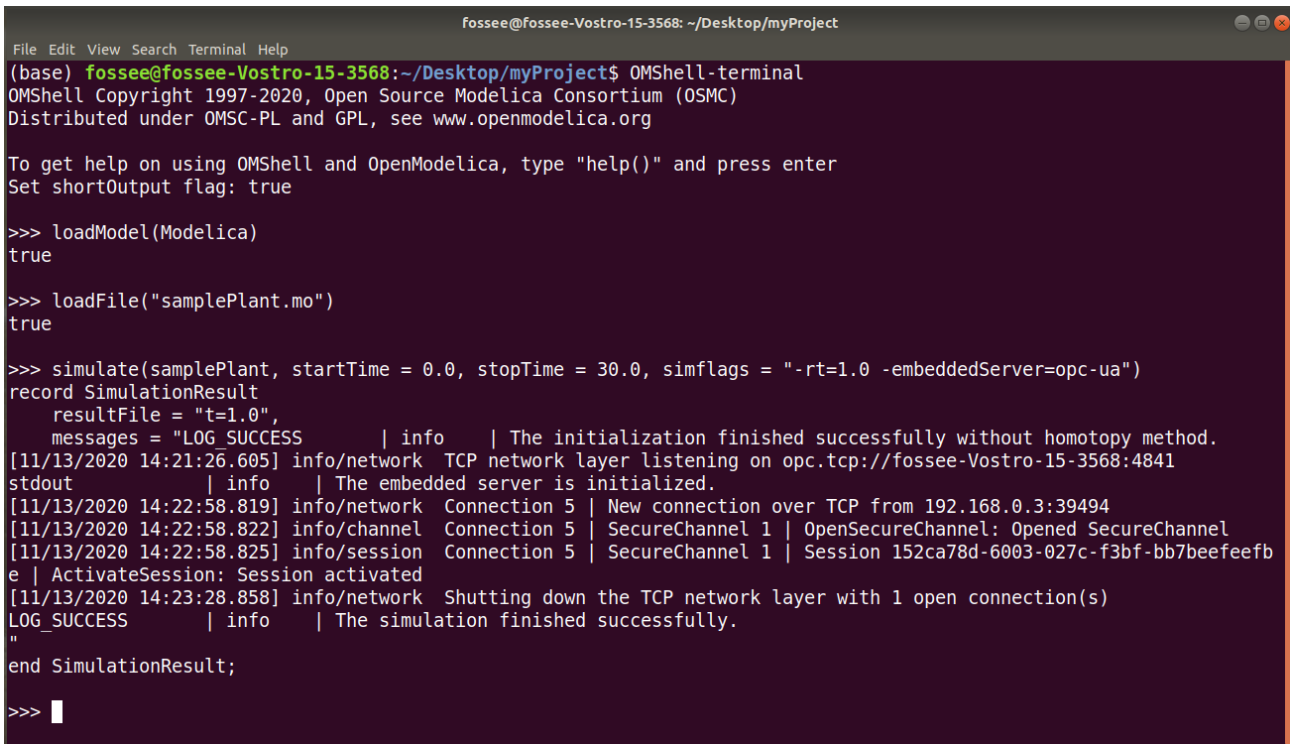

Figure 6.2: Shell commands to simulate a plant using OMShell-terminal

Once the simulation in OpenModelica begins, we will execute our OPC UA client to implement a P controller on our plant. As shown in Fig. 6.2, we are simulating the sample plant for 30 seconds. It may be noted that the last command (simulate) would save a result file (`.mat`) in the current working directory. As we are executing the OPC UA client to implement a P controller on this plant, we will load that result file in OMEdit to visualize the controller's performance. On loading this result file, we get the process variable's response, as shown in Fig. 6.3. Remember, we have the set-point as 10. However, the response got saturated at approximately 4. So, we get a very high steady-state error. To eliminate this error, we need to apply integrative action in our controller. Thus, quite often, P controllers are not sufficient for the task at hand.

Now, we will implement a PID controller. Let us have a look at the mathematical formulation of a PID controller. Remember, we have already seen such formulation in equation 2.5. As we have to write Python code, we will see how to deduce the integral and derivative concepts of a programming language. A simple form of PID control is an example of a controller with an internal state. In velocity form, it is given by

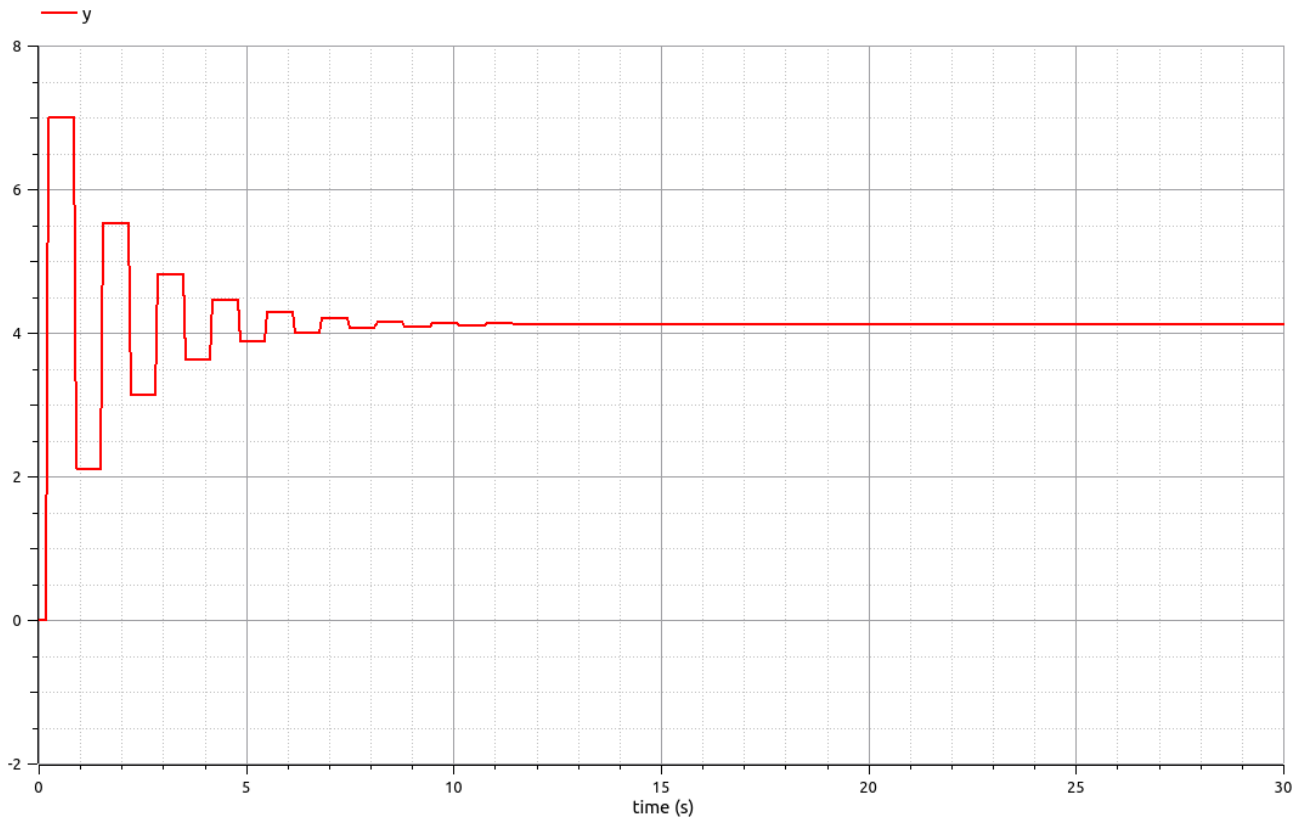

Figure 6.3: Response of process variable controlled by a P controller

$$MV_k = \overline{MV} + K_P e_k + K_I \sum_{k'=0}^{k} e'_k(t_k - t_{k-1}) + K_D \frac{e_k - e_{k-1}}{t_k - t_{k-1}} \tag{6.2}$$

where e_k is the difference between set-point and measured process variable with $e_k = SP_k - PV_k$. K_P, K_I, and K_D are control constants, and t_k is the sampling time. The Python OPC UA client with PID controller is available in App. B.1. Now, we simulate the plant using OMShell-terminal, as shown in Fig. 6.2. Again, the result file will be saved in the current working directory. We will load the result file in OMEdit to visualize the response of the process variable. On loading this result file, we get the process variable's response, as shown in Fig. 6.4. As shown in Fig. 6.4, the system has almost achieved the set-point of 10. Thus, PID controller scores over P controller. One can tune PID controller to smooth the response. However, the purpose of the current project is to establish a communication between OpenModelica and an external controller. In the upcoming sections, we will learn how to control actual systems (or models) by using an OPC UA client.

6.2 Controlling a single tank system

We will first consider a tank system example containing a level sensor and a controller controlling a valve via an actuator. As shown in Fig. 6.5, the liquid enters the tank through a pipe from a source and leaves the tank via another pipe at a rate controlled by a valve. As a control problem, we need to design a controller such that the liquid level in the tank must be maintained at a fixed level as closely as possible. One may note that the model of this system is borrowed from [15]. However, the model given in [15] contains both the governing equations and the controller. As a part of our project, we will suggest ways to control this model via a

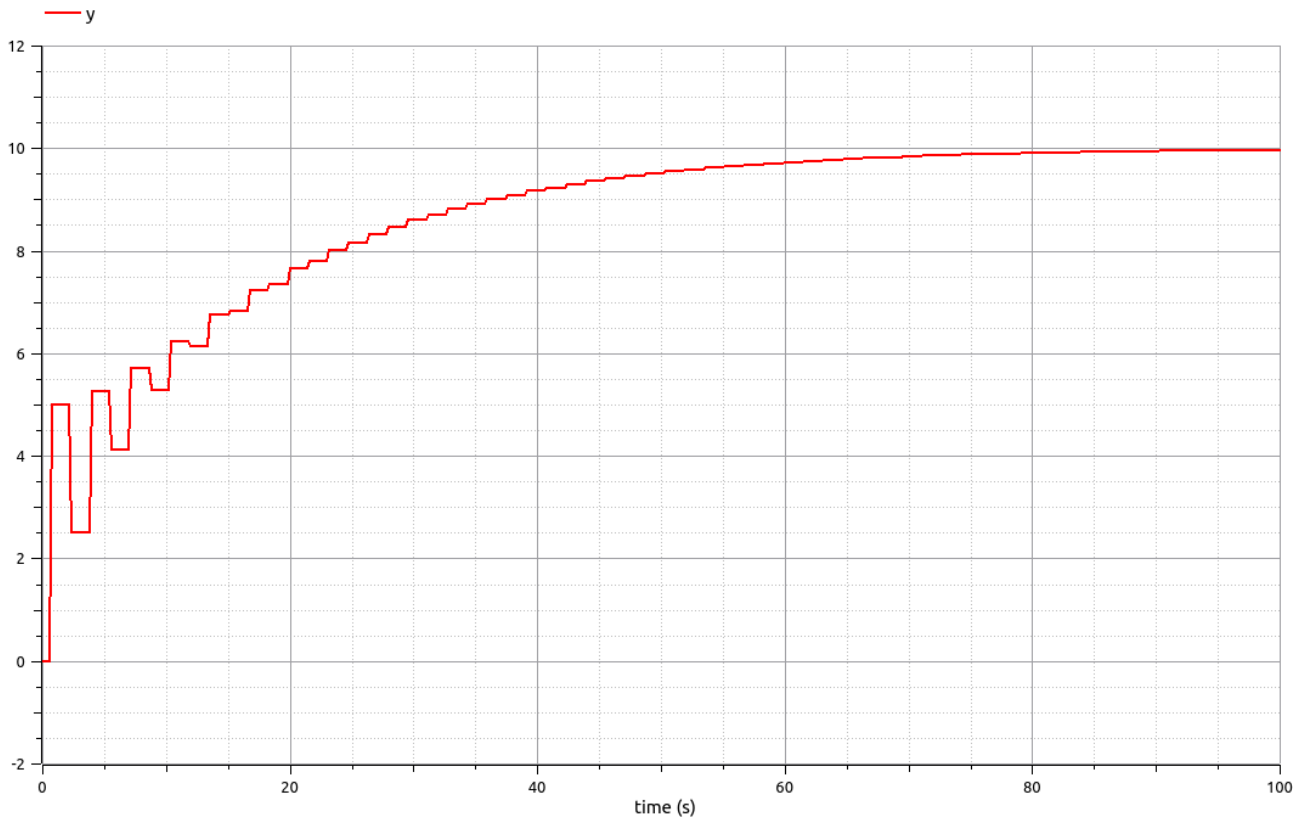

Figure 6.4: Response of process variable controlled by PID controller

Python OPC UA client. That's why we will modify the model accordingly.

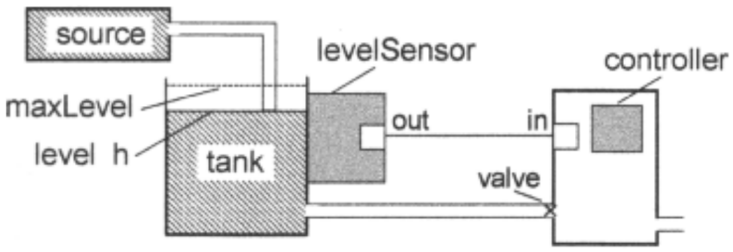

Figure 6.5: A tank system with a single tank, a source for liquid, and a controller

For the model shown in Fig. 6.5, let us say that the height of the liquid in the tank is h and the area of the tank is A. On the other hand, the input flowrate is Q_i and output flowrate is Q_o. Then, the following equation governs the height of liquid in the tank.

$$\frac{dh}{dt} = \frac{Q_i - Q_o}{A} \tag{6.3}$$

Along with this, we will require to specify Q_i while modeling this system in OpenModelica. A complete OpenModelica package of single tank system, as given in [15], is presented in App. B.2. From this package, one may observe that the equations governing the system and PI controller have been implemented together. We can simulate this model for approximately 250 seconds and observe the response of various parameters like h, etc. However, the focus of this project is to control this single tank system via a Python OPC UA client. So, we will first tweak this package so that the corresponding model can be controlled externally.

In a single tank system, we need to design a controller that would maintain a fixed liquid level (h) in the tank. Thus, the PV in this case is h, MV is the controller output ($outCtr$) to be fed back to the system and SP is the desired height. The modified OpenModelica model which can be controlled via a Python OPC UA client is given in App. B.2. In this modified model, we can observe that the following line of the OpenModelica model

```
Real outCtr "Control signal without limiter";
```

has been changed to

```
input Real outCtr "Control signal without limiter";
```

We can begin the simulation with our modified OpenModelica model and the Python OPC UA client. To do so, we first simulate the OpenModelica model by using OMShell-terminal. One may simply issue the commands presented in Fig. 6.2. However, the last command `simulate` in this figure needs to be modified. It is because Fig. 6.2 shows how to simulate a model in OpenModelica. It does not discuss how to simulate a model embedded inside a package in OpenModelica. And, in the case of a single tank system, we have defined our model as one of the components inside a package. From the model (to be controlled via Python OPC UA client) presented in App. B.2, one may observe that we have defined a package (`SingleTankExt`) followed by a model (`Tank`). So, to simulate this model, we will launch OMShell-terminal first and execute the commands as given below:

1. `loadModel(Modelica)` - Successful execution of this command would return `true`.
2. `loadFile("singleTankExt.mo")` - Successful execution of this command would return `true`. One may note that we are loading the entire package here.
3. `simulate(singleTankExt.Tank, startTime = 0, stopTime = 250, simflags = "-rt=1.0 -embeddedServer=opc-ua")` - Here, we are passing the package name (`SingleTankExt`) followed by the model (`Tank`) which we want to simulate. Once this command is issued, the model is simulated for the specified time (`stopTime`). Successful execution of this command will return a success message (`The simulation finished successfully.`) and the result file will be saved in the current working directory. We can load the result file in OMEdit to visualize the response of the process variable.

The Python OPC UA client to control the single tank system is presented in App. B.2. In this client, we have implemented a PI controller to maintain a fixed height of liquid in the tank. As of now, the values of the controller coefficients like K_P and K_I are borrowed from the single tank system provided in [15]. In this OPC UA client, the following lines specify the IDs which the client needs to access:

```
# Find the IDs for MV, PV
writeID = 10 # Controller output
readID = 6 # h
```

Accordingly, the client will continuously fetch the values of PV from `readID` and modify the value of MV at `writeID`. Following lines of the client are used to perform this read and write operations:

```
# Evaluate the PV and MV
PV = modelicaId[readID].get_value()
MV = controller.send([t, PV, SP])

modelicaId[writeID].set_value(MV)
```

Now, we simulate the model in OpenModelica and execute the Python OPC UA client as presented in App. B.2. Accordingly, we plot the response of height of liquid in both cases: one, the controller is implemented in the OpenModelica model itself and another, the controller is implemented in the Python OPC UA client. Fig. 6.6 shows the response of heights in both of the cases. Even though the two responses are pretty similar, one may observe a slight difference.

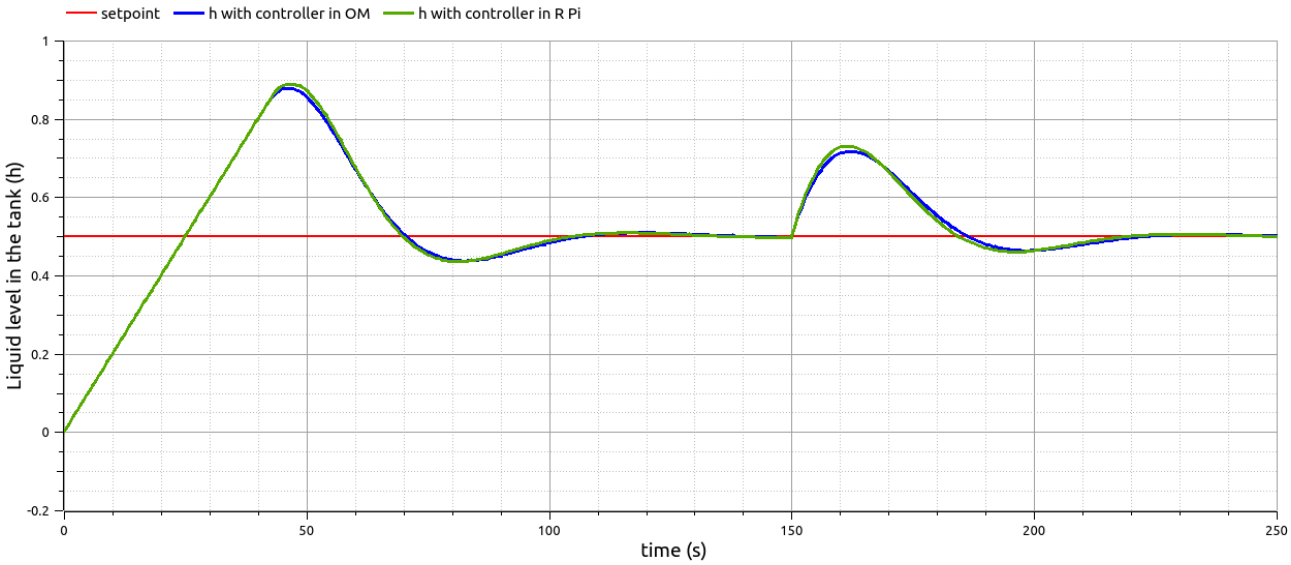

Figure 6.6: Modeling a single tank system with PI controller via OpenModelica and Raspberry Pi (R Pi)

This difference can be attributed to the variation in the solving methods used by OpenModelica and Raspberry Pi. We can also apply PID controller on the Python OPC UA client. However, we will not discuss the PID implementation on a single tank system. Instead, we will proceed towards controlling a connected tanks system.

6.3 Controlling a connected tanks system

We will now discuss a non-interacting two tank system, as shown in Fig. 5.7. In this system, the liquid flows from the first tank to the second tank. As a control problem, we will design two different PID controllers such that a fixed height of the liquid is maintained in each of the two tanks. So, we will have two different values of SP i.e., h_1 and h_2. While designing the model in OpenModelica we will connect the outlet of the first tank to the inlet of the second tank. A flat OpenModelica model for this connected tanks system is given in App. B.3. Please note that this is a package, not a model. From this package, one may observe that the equations governing the system and PID controller have been implemented together. We can simulate this model for approximately 350 seconds and observe the response of various parameters, etc. However, the focus of this project is to control this two tank system via a Python OPC UA client. So, we will first tweak this package so that the corresponding model can be controlled externally.

Like a single tank system, we need to design a controller that would maintain a fixed liquid level (h) in the tank. Thus, the PV in this case is h, MV is the controller output ($outCtr$) to be fed back to the system and SP is the desired height. The only difference is that we have two tanks here and the output valve of first tank needs to be connected to the input valve of the second tank. Following line in the OpenModelica code is used for this connection:

```
tank2_Qi = tank1_Qo;
```

In the modified OpenModelica model, we can observe that the following line of the flat OpenModelica model

```
Real tank1_outCtr, tank2_outCtr "Control signal without limiter";
```

has been changed to

```
input Real tank1_outCtr, tank2_outCtr "Control signal without limiter";
```

So, with our modifed OpenModelica model and the Python OPC UA client, we can begin the simulation. To do so, we first simulate the OpenModelica model by using OMShell-terminal. Once again, we need to remember that this is a model embedded inside a package. That's why it calls for a change in the `simulate` command, as discussed above.

The Python OPC UA client to control the connected tanks system is presented in App. B.3. As per the requirement of the system, we need two different PID controllers to control each of the two tanks. We can think of two approaches for this:

1. Define two different PID controllers and call them one by one.
2. Define a Python class having methods to implement PID and then create two different objects of this class.

This project has followed the class approach to implement the Python OPC UA client for the connected tanks system. In this client, we have implemented a PID controller to maintain a fixed height of liquid in each of the two tanks. While designing the derivative part (D) in the controller, we first implement it using the concept of derivative on error, as shown below:

$$D = K_D \times \frac{d}{dt} error \tag{6.4}$$

As of now, the values of the controller coefficients like K_P, K_I, and K_D are borrowed from the connected tanks system provided in [15]. Now, we simulate the model in OpenModelica and execute the Python OPC UA client as presented in App. B.3. Accordingly, we plot the response of height of liquid in both cases: one, the controller is implemented in the OpenModelica model itself and another, the controller is implemented in the Python OPC UA client. Fig. 6.7 shows the response of heights in both of the cases. Even though the two responses are similar, one may observe an oscillation in the responses when the Python OPC UA client controls the model. This oscillation can be attributed to the inclusion of the derivative term in the controller. As we know that the derivative term in the PID controller (as given in equation 6.2) considers how fast, or the rate at which, error (or PV as we discuss next) is changing at the current moment. That's why the derivative on error (as given in equation 6.4) can result in excessive volatility - spikes in the controller output's behavior often referred to as derivative kick [16]. In contrast, a derivative on measurement applies sensitivity to changes in the setpoint that is more appropriate for practical applications. The following equation shows the implementation

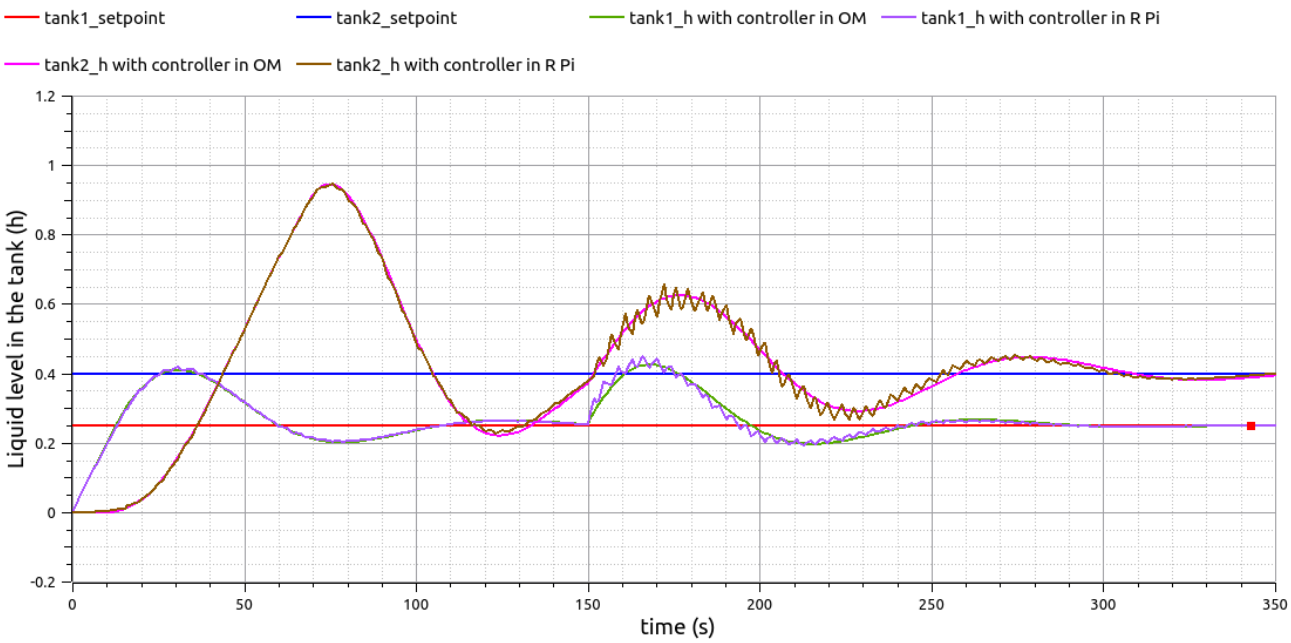

Figure 6.7: Modeling a connected tank system with PID controller (with derivative on error) via OpenModelica and Raspberry Pi (R Pi)

of the derivative part using the concept of derivative on measurement:

$$\begin{aligned} D &= K_D \times \frac{d}{dt} error \\ &= K_D \times \left[\frac{d}{dt}(SP - PV)\right] \\ &= K_D \times \left[\frac{d}{dt}SP - \frac{d}{dt}PV\right] \\ &= K_D \times \left[0 - \frac{d}{dt}PV\right] \\ &= -K_D \times \frac{d}{dt}PV \end{aligned} \tag{6.5}$$

The equation 6.5 assumes that the setpoint is constant. Now, we design the derivative part using the concept of derivative on measurement, as given in equation 6.5. Accordingly, we simulate the model in OpenModelica and execute the Python OPC UA client as presented in App. B.3. Next, we plot the responses. Fig. 6.8 shows the response of heights in each of the two tanks. From Fig. 6.8, we can observe that the effect of oscillations is reduced. So, we can conclude that derivative on measurement provides fewer oscillations as compared to derivative on error [16].

6.4 Modeling hybrid systems

Physical systems evolve continuously over time, whereas specific artificial systems evolve by discrete steps between states [15]. That's why we often come across hybrid systems which exhibit both continuous and discrete behavior. By continuous behavior, we mean that a differential equation describes the flow of the system. On the other hand, discrete behavior implies that a state machine describes the system. Therefore, hybrid systems arise in embedded control when digital controllers, computers, and subsystems modeled as finite-state machines are coupled with controllers and plants modeled by partial or ordinary differential equations or difference

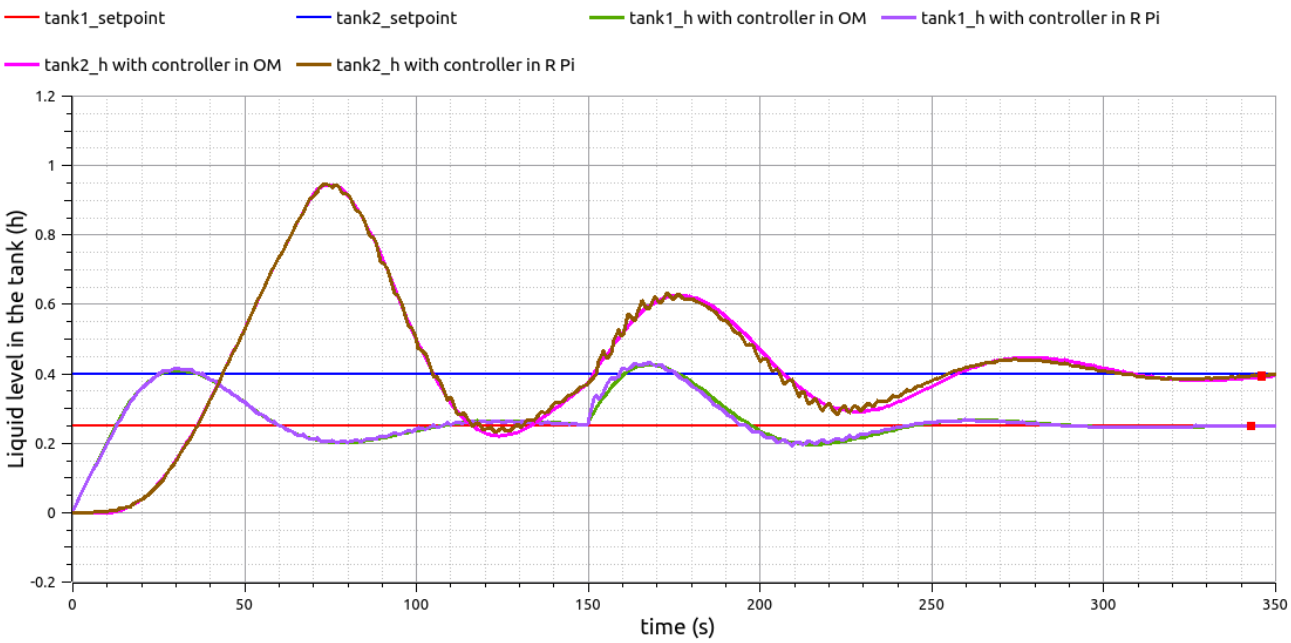

Figure 6.8: Modeling a connected tank system with PID controller (with derivative on measurement) via OpenModelica and Raspberry Pi (R Pi)

equations [17]. We have discussed implementing continuous controllers on various systems in the previous sections, like a single tank, connected tanks, etc. Therefore we dedicate this section to the implementation of discrete controllers on the same systems.

6.4.1 Applying discrete controllers on single tank system

We will first consider the single tank system. The OpenModelica model (package) for this system is stated in App. B.2. In this model, we have used a continuous PI controller, as shown by the following lines of the model:

```
der(x) = error / T;
outCtr = K * (error + x);
```

From the above-mentioned lines, we can infer the following equations govern the behavior of PI controller.

$$\begin{aligned} \frac{dx}{dt} &= \frac{error}{T} \\ outCtr &= K \times (error + x) \end{aligned} \tag{6.6}$$

Now, we need to modify these equations to implement a discrete controller. For this, we will discretize the equation 6.6 to obtain a difference equation. Remember, difference means the change in value over time instants. Assuming a sampling period of $dt \approx T_s$, the equation 6.6 can be modified as given below:

$$\begin{aligned} dx &= \frac{error}{T} \times dt \\ x_k - x_{k-1} &= \frac{error}{T} \times dt \\ x_k &= x_{k-1} + \frac{error}{T} \times T_s \end{aligned} \tag{6.7}$$

So, the modified equations 6.7 can be used in the model to implement a discrete PI controller. Following lines of code are used to implement these equations for an unclocked controller:

```
when sample(0, Ts) then
  x = pre(x) + (error * Ts / T);
  outCtr = K * (error + x);
end when;
```

In a real-life scenario, the sensor signals are sampled, and the actuator signals are held constant during one sample period. Thus, one can modify the above-mentioned implementation accordingly. However, we will work with the unclocked PI discrete controller (as shown above) as of now. The complete model is presented in App. B.4. This model is borrowed from [15]. However, the model given in [15] contains both the governing equations and the controller. As a part of our project, we will suggest ways to control this model via a Python OPC UA client. Remember, we have already discussed how to implement continuous controllers via a Python OPC UA client. Here, we will discuss how to design discrete controllers via a Python OPC UA client. For this, we need to modify the model with unclocked PI discrete controller.

Just as the continuous controllers model, we need to inform OpenModelica that it should expect the controller signal from the simulation environment. For this, we will remove the equations governing PI controller from the model and add a variable as shown below:

```
input Real outCtr "Control signal without limiter";
```

The modified OpenModelica model is given in App. B.4. Now, we will see how to design the Python OPC UA client to mimic the behavior of a discrete PI controller. From the Python OPC UA client, given in App. B.2, one may observe that we have an infinite loop that modifies the value on the server at an interval of one second. As a result, the values of the integrator in the controller are being modified at one second (`time.sleep(1)`). As presented in App. B.4, the unclocked PI discrete controller in the model has a sampling time (Ts) of 0.1 second. One may think that we can change the existing interval in the Python OPC UA client from 1 second to 0.1 second (`time.sleep(0.1)`) and get going. However, this won't work. In a discrete system, not only do we want values at a particular T_s, but also we want the value to remain unchanged throughout each non-zero region of time. For this, we need to implement a `clock` in our client, which one cannot implement by just changing the value of the existing interval.

To implement the `clock`, we will make use of the `clock` function provided by `tclab` module [18]. `clock(period)` is an iterator that generates a sequence of equally spaced time steps from zero to period separated by one second intervals. There is an optional argument `step` which can also be passed in `clock()` function. `step` specifies a time step different from one second. For the Python OPC UA client, we will both of these arguments, where `period` means the time for which we want to execute the simulation and `step` means the T_s. Instead of an infinite loop, as shown in App. B.2, we will now use a `for` loop, as shown below:

```
tfinal = 350
Ts = 0.1

for t in clock(tfinal, Ts):
```

The rest of the implementation is the same as that given in App. B.2. The complete Python OPC UA client to implement a discrete PI controller on the single tank system is given in App. B.4. Now, we simulate the model in OpenModelica and execute the Python OPC UA client as presented in App. B.4. Accordingly, we plot the response of height of liquid in both cases: one, the discrete controller is implemented in the OpenModelica model itself, and another, the discrete controller is implemented in the Python OPC UA client. Fig. 6.9 shows the response of heights in both of the cases. Even though the two responses are quite similar, one may observe

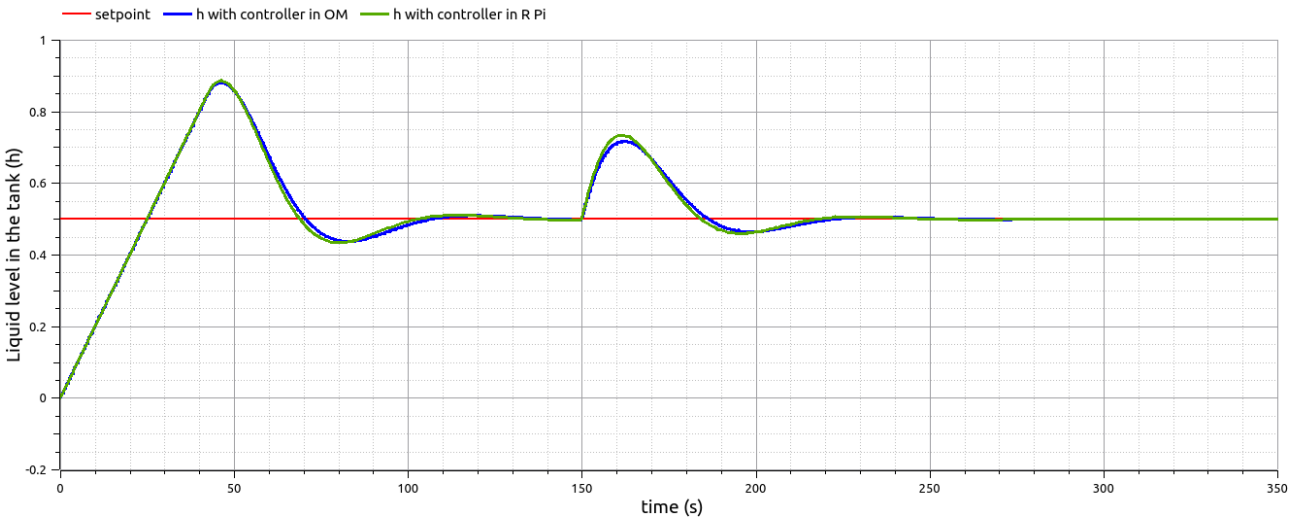

Figure 6.9: Modeling a single tank system with a discrete PI controller via OpenModelica and Raspberry Pi (R Pi)

a slight difference. This difference can be attributed to the variation in the solving methods used by OpenModelica and that by Raspberry Pi. We can also apply PID controller on the Python OPC UA client. However, we will not discuss the PID implementation on single tank system. Instead, we will proceed towards controlling a connected tanks system with discrete PID controllers.

6.4.2 Applying discrete controllers on connected tanks system

We will consider a non-interacting two tanks system, as discussed in Sec. 6.3. Remember, non-interacting means that the variation in h_2 in tank 2 does not affect the transient response occurring in tank 1 [19]. The OpenModelica model (package) for this system is stated in App. B.3. In this model, we have used a continuous PID controller, as shown by the following lines of the model:

```
der(tank1_x) = tank1_error / T;
tank1_y = T * der(tank1_error);
tank1_outCtr = K * (tank1_error + tank1_x + tank1_y);
```

We have already seen how to transform a continuous PI controller into a discrete PI controller. Thus, we only need to know how to modify the derivative part to have a discrete PID controller. From the lines mentioned above, one can observe that

```
tank1_y = T * der(tank1_error);
```

is responsible for governing the derivative part. In this line, $tank1\_y$ is the state variable, T is the time constant, and $tank1\_error$ is the deviation from the reference level (setpoint). Also, we know that difference means the change in value over time instants. Thus, we can rewrite the derivative part, as given below:

$$tank1\_y = T \times (tank1\_y_k - tank1\_y_{k-1}) \tag{6.8}$$

So, the modified equations 6.7 and 6.8 can be used in the model to implement a discrete PID controller. Following lines of code are used to implement these equations for an unclocked PID discrete controller:

```
tank1_x = pre(tank1_x) + (tank1_error * Ts / T);
```

```
tank1_y = T * (tank1_error - pre(tank1_error));
tank1_outCtr = K *(tank1_error + tank1_x + tank1_y);
```

The complete model is presented in App. B.4. This model is adapted from [15]. However, the model given in [15] contains both the governing equations and the controller. As a part of our project, we will suggest ways to control this model via a Python OPC UA client. So, we need to modify the model with an unclocked PID discrete controller. To do so, we will remove the equations governing the PID controller from the model and add a variable named `input Real`. We will not present the modified OpenModelica model here, as it can be easily adapted from the model of single tank system.

Now, we will design the Python OPC UA client to mimic the behavior of a discrete PID controller. Like in the case of single tank system, we will use `clock` function provided by `tclab` module [18]. The rest of the implementation is the same as that given in App. B.3. As we can easily design the client as we did for the discrete PI controller, we will not present the complete Python OPC UA client for discrete PID controller. However, we would like to add that we have followed the derivative on measurement approach for implementing the PID controllers [16]. Next, we will simulate the model in OpenModelica and execute the Python OPC UA client as presented. Accordingly, we plot the response of height of liquid in both cases: one, the discrete PID controller is implemented in the OpenModelica model itself, and another, the discrete PID controller is implemented in the Python OPC UA client. Fig. 6.10 shows the response of heights in both of the cases.

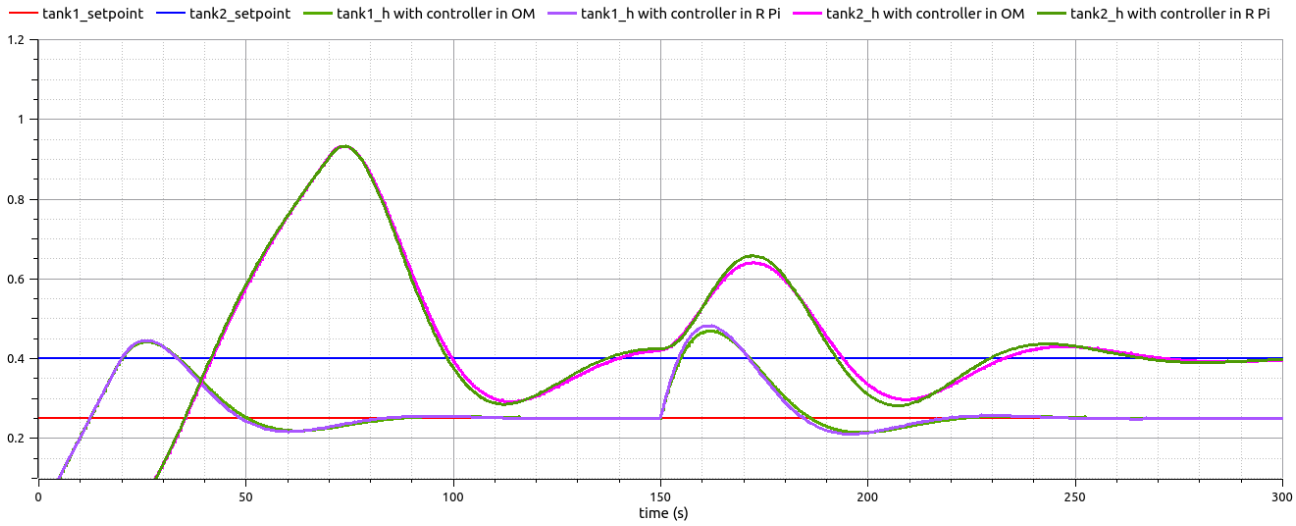

Figure 6.10: Modeling a connected tank system with discrete PID controllers (with derivative on measurement) via OpenModelica and Raspberry Pi (R Pi)

From Fig. 6.8 and Fig. 6.10, we can observe that the Python OPC UA client with discrete PID controllers have shown better performance as compared to the client with continuous PID controllers. This may be attributed to the fact that discrete controllers make sure that the values remain unchanged throughout each non-zero region of time.

6.5 Controlling a batch distillation column

In the previous sections, we have discussed how to control the liquid levels in single tank and connected tanks. In this section, we will explore how to model a batch distillation column [20] in OpenModelica. Next, we will design a Python OPC UA client to control this batch

distillation column.

Distillation is a process in which we separate the components of a mixture based on their boiling points. Let us consider a binary mixture in a distillation column. Binary refers to the mixture having two components, say A and B. The boiling point of A is 50° Celsius, and that of B is 80° Celsius. When steam is subjected to this column, the component having a lower boiling point, i.e., A would evaporate first. Subsequently, another component can be obtained as a residue from the bottom of the distillation column. Even though the process sounds straightforward, its implementation in industries is rigorous and generally costs a lot. We will not dive deep into the types of distillation in this project. Instead, we will consider a batch distillation column. As discussed in [20], the batch distillation process is characterized by a large number of design and operating parameters to be optimized: the number of trays, the size of the initial charge to the still pot, and the reflux ratio as a function of time (during the product withdrawal periods and the slop cut periods). In binary separations, there are two products and one slop cut. In ternary separations, there are three products and two slop cuts. Batch time is established by the time it takes to produce the two distillate products and the heavy product left in the still pot at specified purity levels. A multicomponent batch distillation column is shown in Fig. 6.11. This figure is adapted from [20].

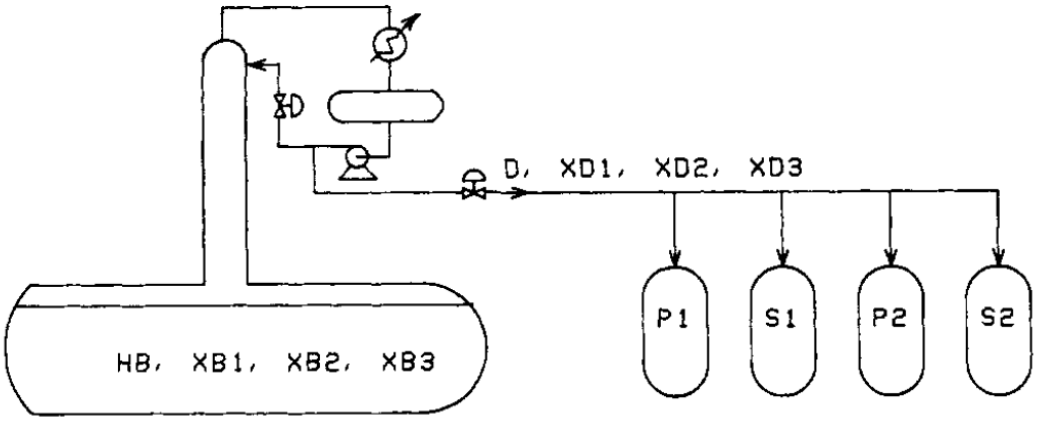

Figure 6.11: Configuration and nomenclature in multicomponent batch distillation

For this project, we will model a batch distillation column in OpenModelica. This model is motivated by design developed in [21]. In this model, we will have a mixture having three components. Thus, we will have three products and two slop cuts. Table 6.2 shows the values of various parameters considered in this model. The OpenModelica model for this distillation column is stated in App. B.4.5. As shown in this model, there are three components to be separated. Once the vapor boilup is begun, a sequence of the following steps occur in the column:

- The column is operated at total reflux until the concentration of the lightest component (or the most volatile) in the distillate, $XD[1]$, reaches the specified purity level. Then distillate product withdrawal is begun at flow rate D.
- The distillate stream is initially the light-component product ($product1$) and is withdrawn into a product tank until the average composition of the material in this tank drops to the specified purity level $XD1SP$. Subsequently, the distillate stream is diverted to another tank, and the first slop cut ($slop1$) is produced.

- When the concentration of the intermediate component in the distillate, $XD[2]$, reaches its specified purity level $XD2SP$, the distillate is diverted to a third tank in which second product ($product2$) is collected.

Parameter	Description	Value
C	Number of components	3
N	Number of trays	40
$XB0[C]$	Initial mole fraction	{0.3, 0.3, 0.4}
$alpha[C]$	Relative volatilities	9, 3, 1
$HB0$	Amount of material charged to the column	300 mol
HD	Reflux drum holdup	10 mol
HN	Tray liquid holdup	1 mol
V	Vapor boilup rate	100 moles/hour
$XD1SP$, $XD2SP$, and $XB3SP$	Specified purity levels	each 0.95
RR	Reflux ratio	1.22
D	Distillate flow rate	40 moles/hour
$XD[C]$	Composition of distillate	
HB	Instantaneous holdup in the still pot	
$product[C-1]$	Holdup tanks for products	
$slop[C-1]$	Holdup tanks for slops	
$valveProduct[C-1]$	ON/OFF valves for the product tanks	
$valveSlop[C-1]$	ON/OFF valves for the slop tanks	

Table 6.2: Parameters considered in the batch distillation column

As shown in the model (App. B.4.5), we have defined two valves: $valveProduct[C-1]$ and $valveSlop[C-1]$. The former valve is used to open or close the product tanks, whereas the latter is used to open or close the slop tanks. So, these valves control the withdrawal process, as shown in Fig. 6.12. Apart from the first four conditions shown in Fig. 6.12, we need to collect the remaining quantity in still pot. For this, we will terminate the simulation once the concentration of $XB[3]$ exceeds the specified purity level ($XB3SP$). This termination is achieved by the `when` statement, as shown in Fig. 6.12.

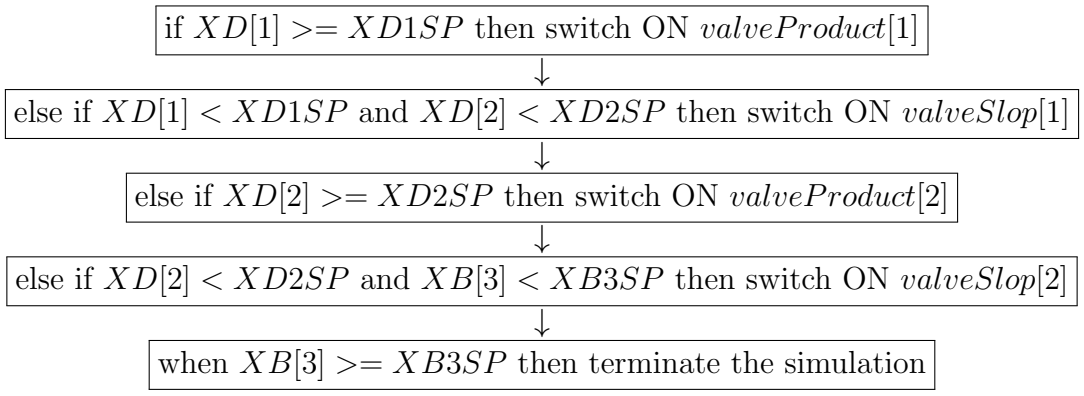

Figure 6.12: Controlling the withdrawal process in the distillation column (App. B.4.5)

Now we simulate the model for 16000 seconds and analyze the results. One may notice that the simulation is terminated at 15269.81 seconds, as shown in Fig. 6.13. This happens due to the inclusion of `when` statement in the model, as shown in Fig. 6.12. Fig. 6.14 shows the mole fraction of $XD[1]$ and $XD[2]$. As shown in Fig. 6.14, the mole fraction of both

$XD[1]$ and $XD[2]$ begins from 0.3. This is because we have stated this as our base condition ($XB0[C] = \{0.3, 0.3, 0.4\}$). Starting from 0.3, the concentration of $XD[1]$ increases, whereas the concentration of $XD[2]$ decreases. One may ask why $XD[1]$ is increasing first? Again, this may be attributed to our base condition of relative volatilities, where $XD[1]$ is set to be the most volatile.

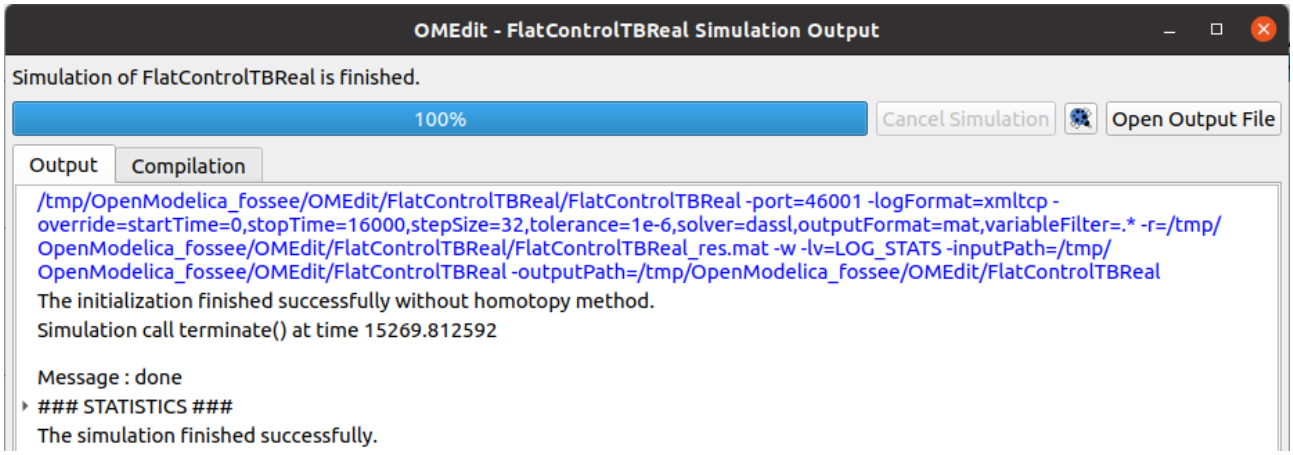

Figure 6.13: Output window after simulating batch distillation column in OpenModelica

Once the concentration of $XD[1]$ starts increasing, it goes from 0.3 to 1. When the concentration exceeds the specified purity level ($XD1SP = 0.95$), we need to collect the distillate steam by opening the valve for $product[1]$. From Fig. 6.14, one may observe that the mole fraction of $XD[1]$ exceeds 0.95 at approximately 1600 seconds. So, at 1600 seconds, we should expect that $valveProduct[1]$ is switched ON and $product[1]$ is being withdrawn. Fig. 6.14 shows that the concentration of $XD[1]$ starts decreasing at approximately 8000 seconds and at the same time, the concentration of $XD[2]$ starts increasing.

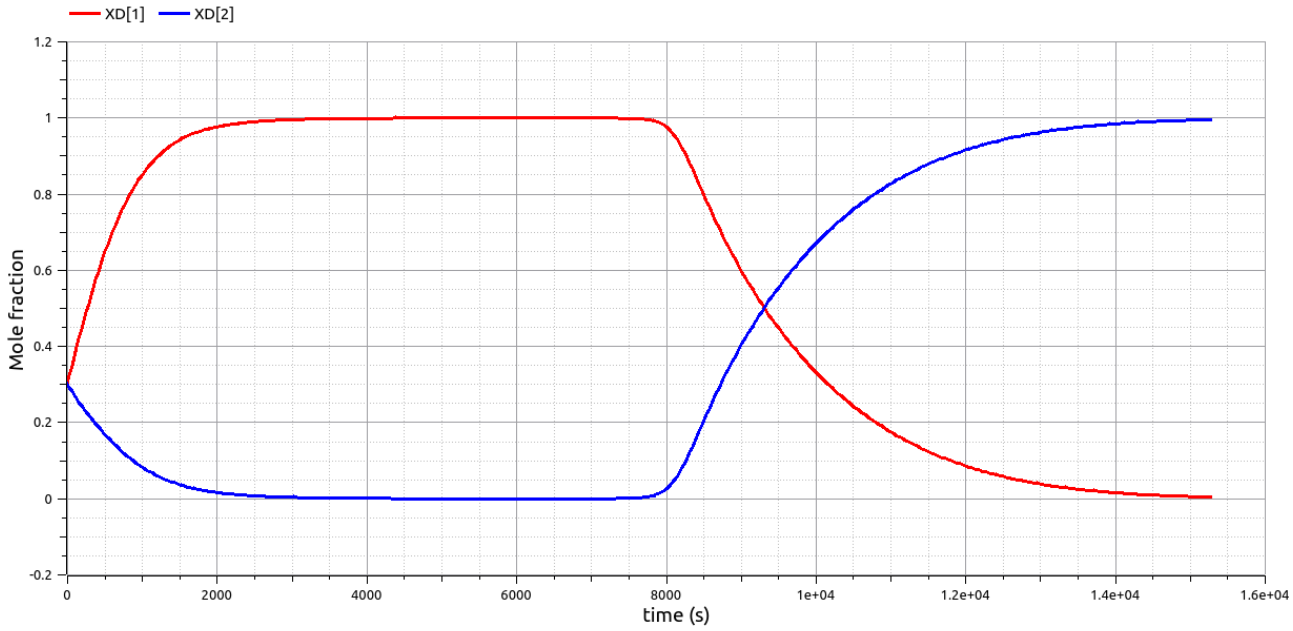

Figure 6.14: Mole fraction of $XD[1]$ and $XD[2]$ in the batch distillation column

When the concentration of $XD[1]$ and $XD2SP$ is below the specified purity levels ($XD[1] < XD1SP$ and $XD[2] < XD2SP$), we will switch ON the $valveSlop[1]$ and divert the distillate stream to $slop[1]$. When the concentration of $XD[2]$ exceeds the specified purity level

($XD2SP = 0.95$), we need to collect the distillate steam by opening the valve for $product[2]$. From Fig. 6.14, one may observe that the mole fraction of $XD[2]$ exceeds 0.95 at approximately 12700 seconds. So, at 12700 seconds, we should expect that $valveProduct[2]$ is switched ON and $product[2]$ is being withdrawn.

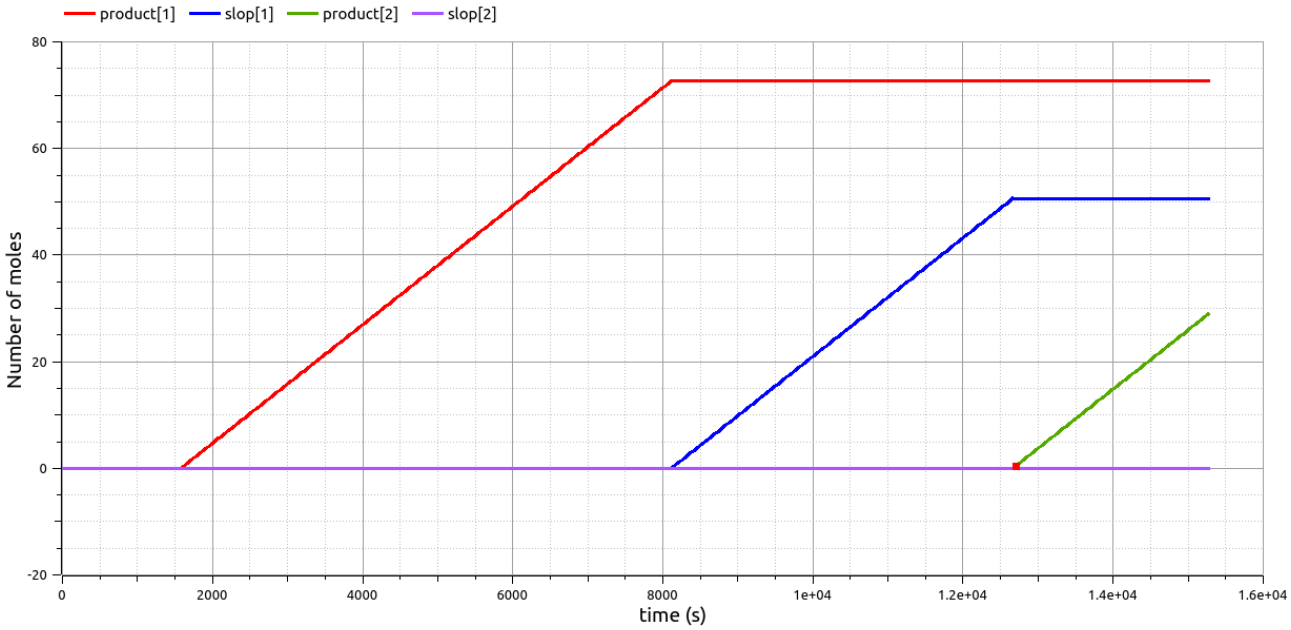

Figure 6.15: Moles of product and slop in the batch distillation column

Fig. 6.15 shows the moles withdrawn in product tanks and slop tanks. As shown in Fig. 6.15, $product[1]$ starts getting accumulated from 1600 seconds and $product[2]$ starts getting accumulated from 12700 seconds. In between, $slop[1]$ is being accumulated. Remember, Table 6.2 shows that the total amount of material charged to the column ($HB0$) is 300 mol. So, our results should add up to that amount. Before that, we would like to see the value of instantaneous holdup in the still pot (HB).

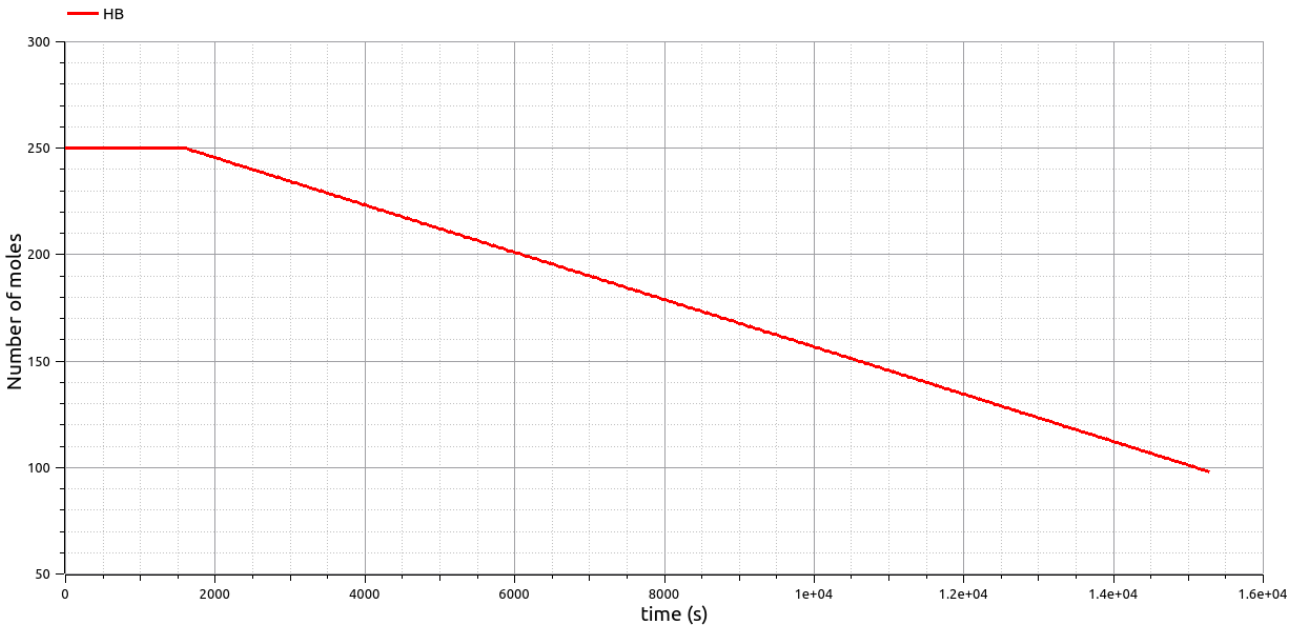

Figure 6.16: Moles of instantaneous holdup in the still pot in the batch distillation column

Fig. 6.16 shows the value (number of moles) of instantaneous holdup in the still pot (HB). As

per the balancing equation, the total number of resulting moles (say, M) can be calculated as follows:

$$\begin{aligned} M &= product[1] + product[2] + slop[1] + slop[2] + HB + HD + (HN \times N) \\ &= 72.6099 + 28.7746 + 50.5968 + 0 + 98.0188 + 10 + (1 \times 40) \\ &= 300.0001 \approx 300 \end{aligned}$$

where the symbols have their meanings as stated in Table 6.2. Had we not added the `when` statement in the model, we would not have received the same number of resulting moles as that being charged to the column.

Now, we will discuss how to control this batch distillation column using a Python OPC UA client. For this, we will first tweak the model presented in App. B.4.5. Remember, we observe the concentration of $XD[1]$, $XD[2]$, and $XB[3]$ in this model. Accordingly, we control the valves of product and slop tanks. As we deploy a Python OPC UA client to control the valves, we will first comment out the controller part from the model. Next, we need to inform OpenModelica that it should expect the controller signal from the simulation environment. That's why we will modify the data type of $valveProduct[C-1]$ and $valveSlop[C-1]$ from `Real` to `input Real`. We used to do this even for the previous systems like single tank, connected tanks, etc. For instance, we changed the data type of $outCtr$ from `Real` to `input Real`. However, in the case of batch distillation column, one may note that $valveProduct[C-1]$ and $valveSlop[C-1]$ are arrays. That's why we need to know how arrays are represented in OPC UA server of OpenModelica. Fig. 6.17 shows that each element of the arrays ($valveProduct[C-1]$, $valveSlop[C-1]$, $XD[C]$) has been assigned an individual node ID. So, we need to design our Python OPC UA client accordingly.

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	BatchDistillationColumn	NS1|Numeric|100000124	XD[1]	0.311820398...	Double	7:52:52.818 PM	7:52:52.818 PM	Good
2	BatchDistillationColumn	NS1|Numeric|100000125	XD[2]	0.295011206...	Double	7:52:52.818 PM	7:52:52.818 PM	Good
3	BatchDistillationColumn	NS1|Numeric|100000126	XD[3]	0.386272191...	Double	7:52:52.818 PM	7:52:52.818 PM	Good
4	BatchDistillationColumn	NS1|Numeric|100000430	valveProduct[1]	0	Double	7:50:43.791 PM	7:50:43.791 PM	Good
5	BatchDistillationColumn	NS1|Numeric|100000431	valveProduct[2]	0	Double	7:50:45.815 PM	7:50:45.815 PM	Good
6	BatchDistillationColumn	NS1|Numeric|100000432	valveSlop[1]	0	Double	7:50:46.911 PM	7:50:46.911 PM	Good
7	BatchDistillationColumn	NS1|Numeric|100000433	valveSlop[2]	0	Double	7:50:48.040 PM	7:50:48.040 PM	Good
8	BatchDistillationColumn	NS0|Numeric|10001	run	true	Boolean	7:52:37.067 PM	7:52:37.067 PM	Good

Figure 6.17: Node IDs of the elements from the model of batch distillation column

The Python OPC UA client to control the batch distillation column is presented in App. B.4.5. In this client, we have implemented ON-OFF controllers to control the valves of product and slop tanks. Connecting this client with the model of batch distillation column throws an error, as shown in Fig. **??**. It shows that the OPC UA server of OpenModelica crashes while simulating the batch distillation column with OPC UA flags enabled. We tried to investigate the reason behind this crash. Since we could not find the reason(s), we reported this issue to the developers of OpenModelica [22]. Dr. Martin Sjölund (who works for the Open Source Modelica Consortium and is developing the OpenModelica Compiler) has also confirmed that the server crashes while simulating this model. He claims that this issue might be due to the error in the library (open62541) itself. Owing to these reasons, we decided to store the simulation data at the client side so that we could analyze the results. Even though we can store all the variables, we will only store the following variables:

- $XD[1]$ and $XD[2]$: It will contain the compositions of distillate.
- $product[1]$ and $product[2]$: It will contain the number of moles collected in product tanks.
- $slop[1]$ and $slop[2]$: It will contain the number of moles collected in slop tanks.

- HB: This is required to store the instantaneous holdup in the still pot.
- *time*: Here, we will store the server time. One may note that OPC UA server of OpenModelica transmits time as one of the variables.

Chapter 7

Automating simulation in OpenModelica

In this chapter, we will discuss how to automate the simulation of a model in OpenModelica. Remember, the interactive OpenModelica environment consists of quite a few components, including UI-based applications and command-line interface (CLI) based applications. OMShell and OMShell-terminal are CLI-based applications, whereas OMEdit is a UI-based application. In this project, we are using OMEdit to simulate a model which is being controlled in OpenModelica itself. On the other hand, we are using OMShell-terminal (on Linux OS) to simulate models which have to be controlled by an external Python OPC UA client. While simulating models in OMShell-terminal, one needs to launch the OMShell-terminal first, followed by issuing a couple of commands needed to execute the model. Fig. 6.2 shows one such execution. As shown in Fig. 6.2, we are simulating a model named `samplePlant.mo` for a duration of 30 seconds over the OPC UA server in real-time. The last command (`simulate` will be modified if one wishes to simulate a model inside a package in OpenModelica). As obvious from Fig. 6.2, we need to type these commands manually whenever we have to simulate a model using OMShell-terminal. Manual typing might be cumbersome given that we might have to simulate different models and that too, again and again. That's why it calls for developing a shell script (only for Linux OS) to execute a model with minimal manual typing.

7.1 Arguments needed to simulate a model

Here, we present the development of a shell script that will help simulate (or execute) a model with minimal manual typing. Before diving deep into the shell script, we will present the steps to simulate a model using OMShell-terminal. We will consider the OpenModelica models as stated in App. A.2, App. B.1, and App. B.2. Let us consider the model given in App. A.2. First of all, we would like to add that it is a package, not a model. The name of the package is `TwoTank`, and in this package, we have several components (or specializations), as shown in Fig. 7.1. Now, if we have to simulate the model named System for 20 seconds of this package,

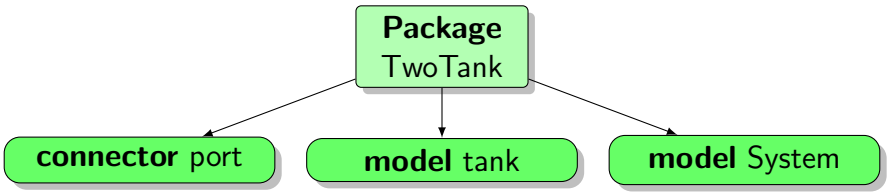

Figure 7.1: Various components in the package TwoTank (App. A.2)

we will have to follow the steps stated in Fig. 7.2.

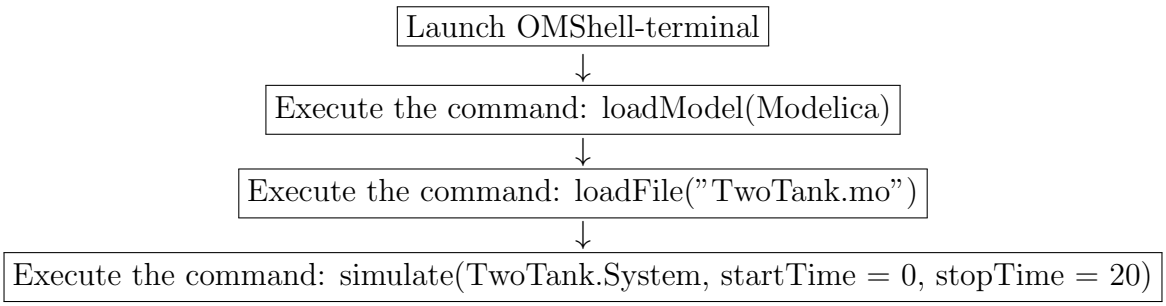

Figure 7.2: Steps to simulate a model in the package TwoTank (App. A.2)

At the same time, if we want to simulate the same model with the OPC UA server activated, we will have to modify the last step as shown in Fig. 6.2. From Fig. 6.2 and Fig. 7.1, one may observe that certain commands like loadModel(Modelica), etc. are same while simulating any model. That's why we can have a shell script that will automatically call these commands.

Now let us consider the sample model given in App. B.1. As one can see, this is a model with no components embedded in it. To simulate this model, we need to follow the steps given in Fig. 6.2. If one does not require the OPC UA server to be enabled, we will omit the `simflags` argument from the `simulate` command. That's why we can infer that the major difference between simulating a model embedded inside a package and simulating a flat model is the execution of `simulate` command.

7.2 Shell script for simulating models

The shell script to simulate the models in OpenModelica is presented in App. C. In this script, we have included the methods for simulating a model with/without OPC UA server. We now discuss how to simulate a model using this shell script. For the sake of convenience, we will cover four simulations, as given below:

1. A model without OPC UA server: Let us consider the sample model given in App. B.1. Assuming that the shell script and the model are present in the working directory and we want to simulate this model for 100 seconds, we will execute the following command in a Linux Terminal:

   ```
   sh om-sim.sh samplePlant.mo 100 1
   ```

   Fig. 7.3 shows the results being produced from the execution of the above command.

2. A model with OPC UA server: Let us consider the same sample model from the previous step. Assuming that the shell script and the model are present in the working directory and we want to simulate this model with OPC UA server for 100 seconds, we will execute the following command in a Linux Terminal:

   ```
   sh om-sim.sh samplePlant.mo 100 2
   ```

3. A model embedded inside a package without OPC UA server: Let us consider the package given in App. A.2. As shown in Fig. 7.1, TwoTank is the package, and System is the

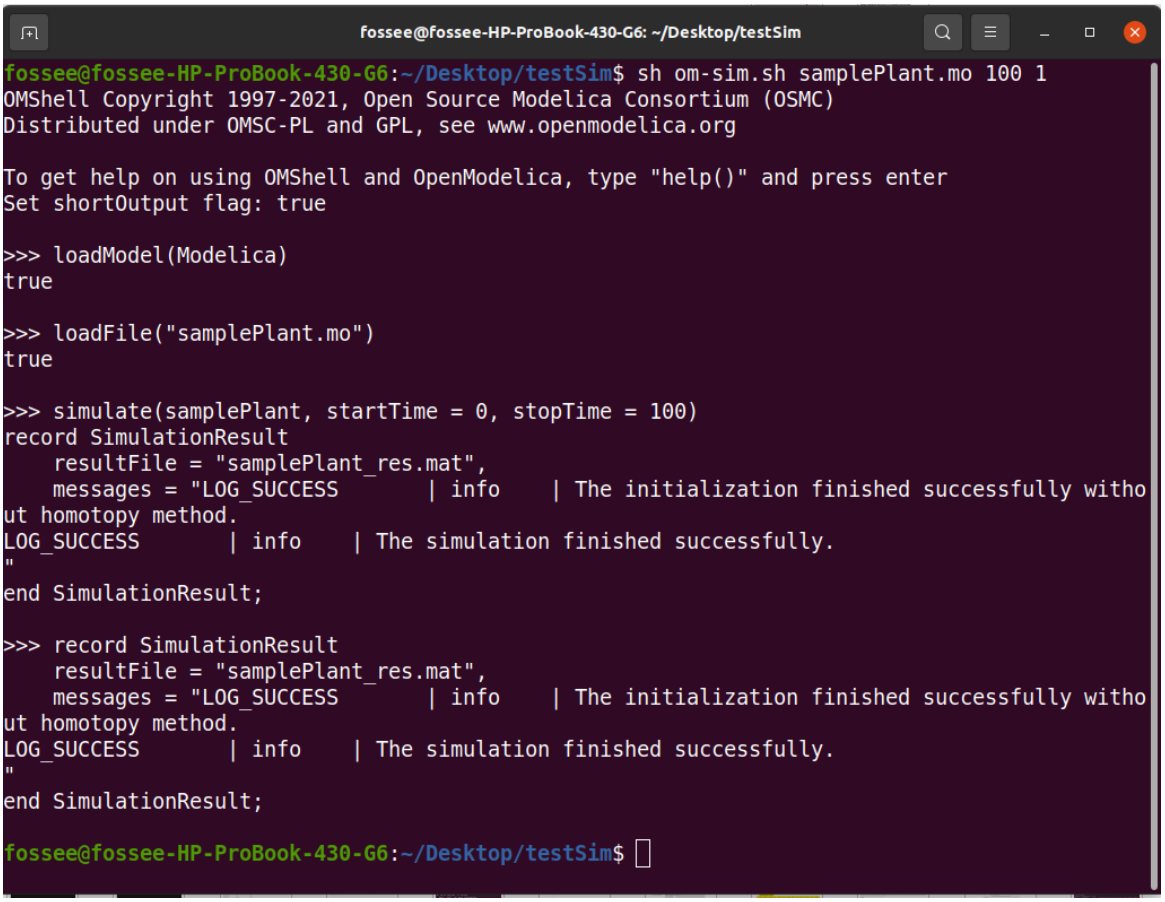

Figure 7.3: Linux Terminal showing the execution of shell script with a model

model we want to simulate. Assuming that the shell script and the package are present in the working directory and we want to simulate the model (named System) for 20 seconds, we will execute the following command in a Linux Terminal:

```
sh om-sim.sh TwoTank.mo System 20 1
```

4. A model with OPC UA server: Let us consider the same sample model from the previous step. Assuming that the shell script and the package are present in the working directory and we want to simulate this model (named System) with OPC UA server for 20 seconds, we will execute the following command in a Linux Terminal:

```
sh om-sim.sh TwoTank.mo System 20 2
```

From the above illustrations, we may conclude that the last argument in the shell script decides whether the OPC UA server has to be enabled or not (1: without OPC UA server, and 2: with OPC UA server). Also, this script is only simulating the model for a specified time. However, certain other parameters like outputFormat, stepSize, etc., can be added as additional arguments in this script. We leave these exercises as future work for this project.

Note: While simulating a model from OMShell-terminal, OpenModelica generates quite a few temporary files, as shown in Fig. 7.4. Out of these files, this project mainly requires the files ending with extensions `.mo` and `.mat`. So, we have modified our shell script (App. C) such

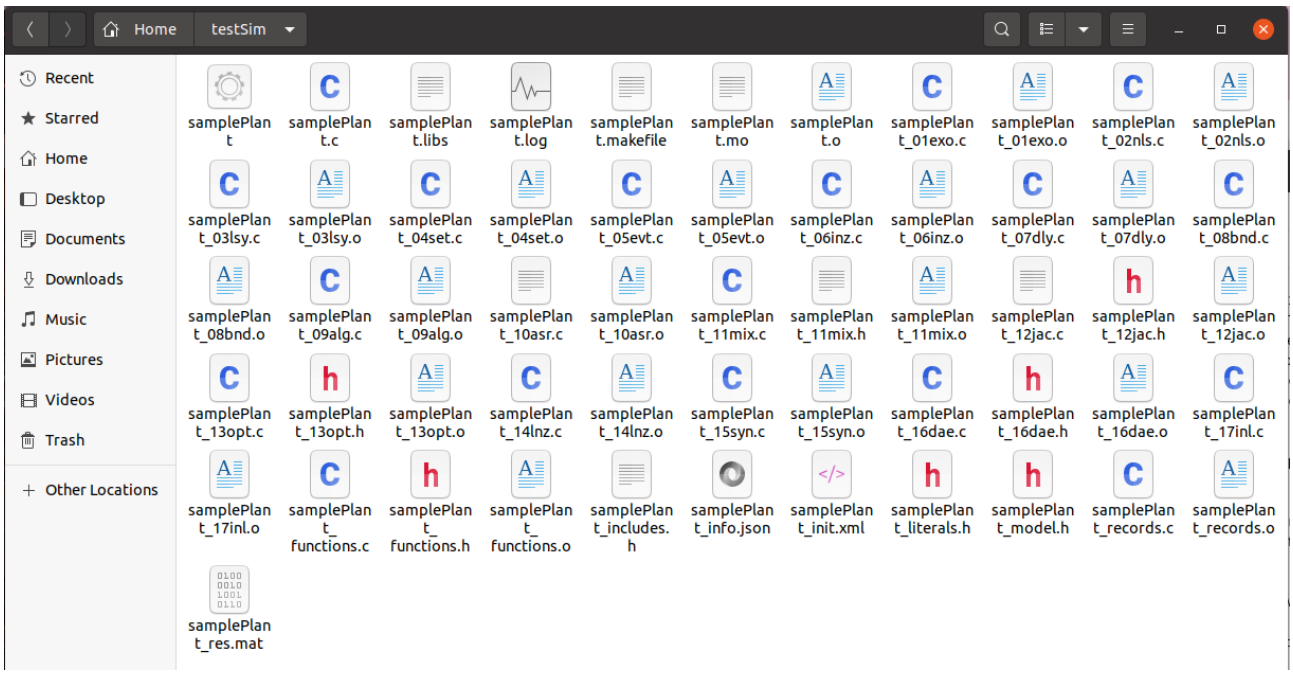

Figure 7.4: Sample folder showing the various files generated after the simulation of model (given in App. B.1) for 10 seconds

that the files other than these two are deleted at the end of the simulation. The following line in the shell script helps us achieve this:

```
rm $(ls -I "*.mo" -I "*.mat" -I "*.sh")
```

As seen in the above-mentioned line, the files other than three extensions - `.mo`, `.mat`, and `.sh` - are being removed at the end of the simulation. Thus, the users are advised to be careful while using this script. Alternatively, they can modify this line of code as per their requirements.

Chapter 8

Investigating the built-in OPC UA server of OpenModelica

In this chapter, we will investigate the built-in OPC UA server of OpenModelica. In the previous chapters, we have shown how to deploy a Python OPC UA client to control/ monitor a model running in OpenModelica. Even though we could control those models, we would like to elaborate on some key factors which one should keep in mind. First, we will summarize the various data types (like an array, float, etc.) which can be read/write on OPC UA server. Next, we will present the data types (like Boolean, Integer, etc.) that cannot be modified on the OPC UA server in OpenModelica. At the same time, we will also compare the OPC UA server of OpenModelica with other OPC UA servers. This would provide an insight into the existing structure of OPC UA server in OpenModelica.

8.1 Data types that can be modified on OPC UA server

In Chapter 6, we first started with controlling a sample system, as given in Sec. 6.1. Next, we extended the technique to control actual systems, like single tank system, connected tanks system, etc. One can revisit Sec. 6.2 and Sec. 6.3 to know more about these techniques. In these three systems, we followed the same modus operandi: read the PV, calculate the error, and modify MV on the server. Apart from this, we would like to mention that both of the variables - PV and MV - in these cases were of `float` data type. For example, in the sample plant system (presented in Sec. 6.1), we had defined both y (PV) and x (MV) as `Real` and `input Real`, respectively, in OpenModelica. Similarly, in the single tank system (stated in Sec. 6.2), we had defined the process variable, height of the liquid in the tank (h), as `Real` and the manipulated variable, controller output ($outCtr$), as `input Real`. From these examples, we infer that one can modify the variables which have been defined as `input Real` in OpenModelica. In the Python OPC UAclient, we used the methods `get_value` to read the PV and `set_value` to modify the MV. Note that MV are those variables which have been defined as `input Real` in OpenModelica.

Consider the sample plant given in Sec. 6.1. The OpenModelica model for this plant is as given below:

```
model samplePlant
  input Real x;
  Real y;

equation
```

```
  y = x;
end samplePlant;
```

This time, we will not implement a controller on this plant. Instead, we will try to modify the value of y using a Python OPC UA client. Remember, OpenModelica automatically assigns a value of zero to the variables which have been defined as `input Real`. So, let us design a Python OPC UA client, which will assign a value of 5 to y. It seems to be an easy task: find the node ID of y and use the method `set_value` to modify the value at that ID. Suppose, `modelicaId` is the dictionary which contains all the children nodes of the client, `writeID` is the node ID of y and `set_point = 5` is the value to be written. So, we can issue the following command:

```
modelicaId[writeID].set_value(set_point)
```

Ideally, this command should work. However, when we executed it with the sample plant running in OpenModelica (with OPC UA server enabled), we got an error as shown in Fig. 8.1.

```
Traceback (most recent call last):
  File "float-test.py", line 56, in <module>
    modelicaId[writeID].set_value(set_point)
  File "/home/fossee/.local/lib/python3.8/site-packages/opcua/common/node.py", line 217, in set_value
    self.set_attribute(ua.AttributeIds.Value, datavalue)
  File "/home/fossee/.local/lib/python3.8/site-packages/opcua/common/node.py", line 263, in set_attribute
    result[0].check()
  File "/home/fossee/.local/lib/python3.8/site-packages/opcua/ua/uatypes.py", line 218, in check
    raise UaStatusCodeError(self.value)
opcua.ua.uaerrors._auto.BadUnexpectedError: "An unexpected error occurred."(BadUnexpectedError)
```

Figure 8.1: Error while writing an integer value on the server

Where did we go wrong? First, we need to check the data type of y. For this, we can execute the following command in the OPC UA client:

```
print(type(modelicaId[writeID].get_value()))
```

This command reveals the fact that the data type of values coming from the server is `float`. And, we are trying to write an integer value (`set_point = 5`) in this variable. Apparently, there is a type mismatch that compels the Python OPC UA client to throw the error as shown in Fig. 8.1. Thus, we change (`set_point = 5`) to (`set_point = 5.0`). Now, we can execute our client, and it should work as desired. From this situation, we can conclude that the variables which are defined as `input Real` in OpenModelicaare interpreted as `float` in the Python OPC UA client. So, it seems that the current implementation of OPC UA server in OpenModelica only supports float values to be written on the server. Since we don't have much control over enabling the OPC UA server in OpenModelica (except that we pass `-embeddedServer=opc-ua` as simflags), we cannot change the data type of variables. That's why we always have to write only float values on the server. This can also be substantiated with the systems (or models), which we have controlled in Chapter 6. In all of these models, we have dealt with a single element at a time.

Apart from single elements, some models in OpenModelica have arrays. For the sake of simplicity, we will only discuss 1-D (one-dimensional) and 2-D (two-dimensional) arrays. A simple example of a 1-D array is a vector, whereas a matrix can be visualized as a 2-D array. Let us consider a situation in which an OpenModelica model has an array, and we want to modify its elements. The OpenModelica model is as given below:

```
model numArrayTest
  parameter Integer C = 3 "Number of rows";
  parameter Integer N('No of Trays') = 2 "Number of columns";
  input Real array1D[C - 1] "Sample 1-D array";
  input Real array2D[N, C] "Sample 2-D array";

equation
end numArrayTest;
```

In this model, we have a 1-D array named `array1D` which has 2 elements. On the other hand, we have defined a 2-D array named `array2D` which has 2 rows and 3 columns - 6 elements. Given that our main focus is on modifying an array on OPC UA server, we have not added any equations in this model. Now, we will proceed towards simulating this model and modifying its variables. We need to know how the array elements are represented on the OPC UA server. For this, we will make use of UaExpert [13]. Fig. 8.2 shows that each element of an array has been assigned an individual node ID. It means that we need to know the node ID of the element which we want to modify. One can expect to have a single node ID for the entire array. However, the existing implementation of OPC UA in OpenModelica flatters an array and assigns an individual node ID to each element. In the upcoming sections, we will compare OPC UA server in OpenModelica with that in other simulation tools.

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	numArrayTest	NS1|Numeric|100000000	array1D[1]	0	Double	3:17:54.720 PM	3:17:54.720 PM	Good
2	numArrayTest	NS1|Numeric|100000001	array1D[2]	0	Double	3:17:55.663 PM	3:17:55.663 PM	Good
3	numArrayTest	NS1|Numeric|100000002	array2D[1,1]	0	Double	3:17:56.617 PM	3:17:56.617 PM	Good
4	numArrayTest	NS1|Numeric|100000003	array2D[1,2]	0	Double	3:17:57.626 PM	3:17:57.626 PM	Good
5	numArrayTest	NS1|Numeric|100000004	array2D[1,3]	0	Double	3:17:58.856 PM	3:17:58.856 PM	Good
6	numArrayTest	NS1|Numeric|100000005	array2D[2,1]	0	Double	3:17:59.990 PM	3:17:59.990 PM	Good
7	numArrayTest	NS1|Numeric|100000006	array2D[2,2]	0	Double	3:18:01.007 PM	3:18:01.007 PM	Good
8	numArrayTest	NS1|Numeric|100000007	array2D[2,3]	0	Double	3:18:01.920 PM	3:18:01.920 PM	Good
9	numArrayTest	NS0|Numeric|10004	time	0	Double	3:20:26.302 PM	3:20:26.302 PM	Good
10	numArrayTest	NS0|Numeric|10000	step	false	Boolean	3:20:27.312 PM	3:20:27.312 PM	Good
11	numArrayTest	NS0|Numeric|10001	run	false	Boolean	3:21:26.800 PM	3:21:26.800 PM	Good
12	numArrayTest	NS0|Numeric|10002	realTimeScalingFactor	1	Double	3:21:28.400 PM	3:21:28.400 PM	Good
13	numArrayTest	NS0|Numeric|10003	enableStopTime	true	Boolean	3:21:29.760 PM	3:21:29.760 PM	Good

Figure 8.2: Node IDs of the elements from the model `numArrayTest`

As of now, we will proceed towards modifying the array elements. Let us say we want to modify `array1D[2]` and `array2D[1, 3]`. We will simulate this model for 20 seconds, and when the server time is greater than 5, we will set `array1D[2]` to `2.5`. Next, when the server time is greater than 10, we will set `array2D[1, 3]` to 5.0 and `array1D[2]` to `0.5`. Remember, we will have to access each element by its individual node ID. Now, we will simulate the model `arrayTest` using the shell script discussed in Chapter 7. Fig. 8.3 shows the plots of the respective elements after the simulation is finished. Thus, we can infer that we need to access the array elements by their individual node IDs.

We have now discussed the two data types - float and array, which can be modified in the OPC UA server of OpenModelica. However, we have observed that certain data types cannot be modified. The next section will cover these data types.

8.2 Data types that can't be modified on OPC UA server

In the previous section, we observed that the variables defined as `input Real` in OpenModelicawere interpreted as a float data type in Python OPC UA client. Accordingly, we could only write float values on that variables. Now, suppose we want to write an integer value

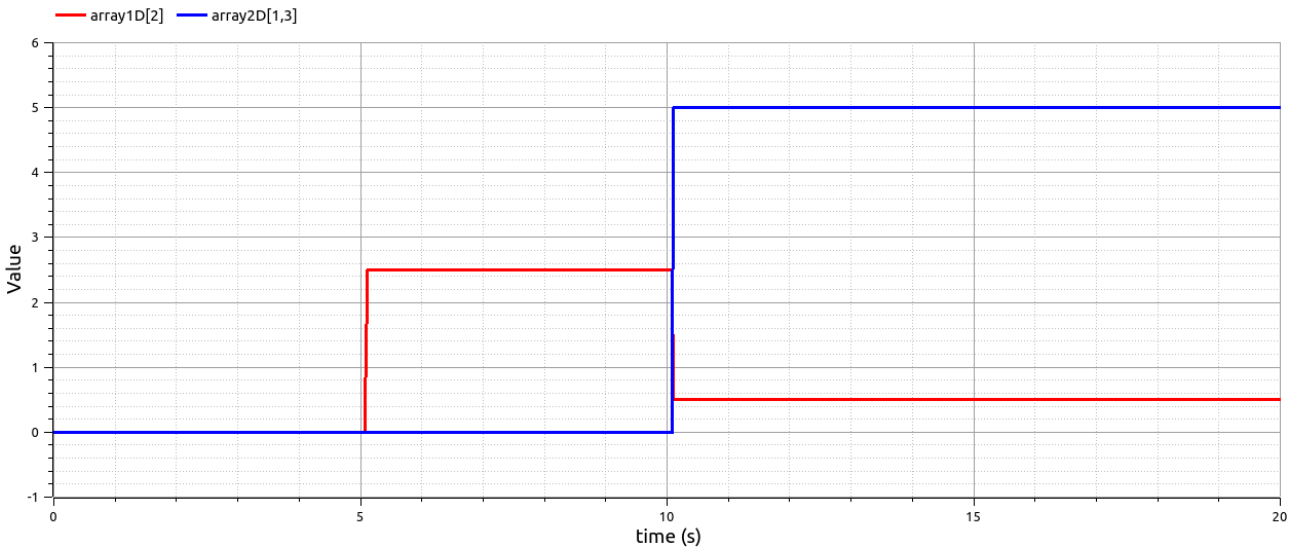

Figure 8.3: Modifying the elements from the model `numArrayTest`

on OPC UA server in OpenModelica. What if we define a variable as `input Integer` in the OpenModelica model? Let us consider the model given below:

```
model integerTest
  input Integer x;
  Real y;

equation
  y = x;
end integerTest;
```

Here, we will try to modify the value of x. As usual, we first need to know the node ID of the variables. Again, we will make use of UaExpert [13] for this task. Fig. 8.4 shows the variables and server parameters being received in the UaExpert. Surprisingly, x (the one defined as `input Integer`) is not being shown on the server. It means that the OPC UA server of OpenModelica does not broadcast the variables defined as `input Integer`. It may be noted that OpenModelica does not show the variables defined as `parameter` as well. So, we need to ensure that we don't define our variables as `input Integer` while working with OPC UA server.

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	integerTest	NS1|Numeric|100000000	y	0	Double	6:12:48.323 AM	6:12:48.323 AM	Good
2	integerTest	NS0|Numeric|10004	time	0	Double	6:12:49.273 AM	6:12:49.273 AM	Good
3	integerTest	NS0|Numeric|10000	step	false	Boolean	6:12:50.026 AM	6:12:50.026 AM	Good
4	integerTest	NS0|Numeric|10001	run	false	Boolean	6:12:50.946 AM	6:12:50.946 AM	Good
5	integerTest	NS0|Numeric|10002	realTimeScalingFactor	1	Double	6:12:51.825 AM	6:12:51.825 AM	Good
6	integerTest	NS0|Numeric|10003	enableStopTime	true	Boolean	6:12:52.874 AM	6:12:52.874 AM	Good

Figure 8.4: Node Ids of the elements from the model `integerTest`

Till now, we have only covered the data structures having numeric elements. Suppose we have an OpenModelica model, which has an array comprising Boolean elements. The model is as shown below:

```
model boolArrayTest
  parameter Integer C = 3 "Number of rows";
  input Boolean array1D[C - 1] "Sample 1-D array";
```

```
equation

end boolArrayTest;
```

In this model, we have a 1-D Boolean array named `array1D` which has 2 elements. Like the model `numArrayTest` discussed earlier, each element of this Boolean array will be assigned an individual node ID on OPC UA server. Fig. 8.5 shows that each element of an array has been assigned an individual node ID. Let us say we want to set `array1D[2] as True`. Using the

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	boolArrayTest	NS1|Numeric|200000000	array1D[1]	false	Boolean	3:32:37.678 PM	3:32:37.678 PM	Good
2	boolArrayTest	NS1|Numeric|200000001	array1D[2]	false	Boolean	3:32:39.090 PM	3:32:39.090 PM	Good
3	boolArrayTest	NS0|Numeric|10004	time	0	Double	3:33:00.596 PM	3:33:00.596 PM	Good
4	boolArrayTest	NS0|Numeric|10000	step	false	Boolean	3:33:01.465 PM	3:33:01.465 PM	Good
5	boolArrayTest	NS0|Numeric|10001	run	false	Boolean	3:33:02.418 PM	3:33:02.418 PM	Good
6	boolArrayTest	NS0|Numeric|10002	realTimeScalingFactor	1	Double	3:33:07.034 PM	3:33:07.034 PM	Good
7	boolArrayTest	NS0|Numeric|10003	enableStopTime	true	Boolean	3:33:08.402 PM	3:33:08.402 PM	Good

Figure 8.5: Node Ids of the elements from the model `boolArrayTest`

node ID of this element (as shown in Fig. 8.5), we can apply the following command in a Python OPC UA client:

```
arr1D_2 = client.get_node("ns = 1; i = 200000001")
arr1D_2.set_value(True)
```

However, we would like to report that this command does not modify the aforesaid element. Upon further investigation, we realized that a Python OPC UA client could not modify the variables, which are defined as `input Boolean` in an OpenModelica model.

From the points discussed in this chapter so far, we can infer that a Python OPC UA client could only modify the variables, which are defined as `input Real` in an OpenModelica model. As shown before, these variables are interpreted as `float` in the Python OPC UA client. That's why we can only write a float value on OPC UA server in OpenModelica. Therefore, we would like to offer the following points to those who wish to work with OPC UA server in OpenModelica:

1. A Python OPC UA client could only modify the variables, which are defined as `input Real` in an OpenModelica model.

2. The existing implementation of OPC UA in OpenModelica flatters an array and assigns an individual node ID to each element. Thus, the current implementation doesn't assign a single node ID to the entire array.

3. A Python OPC UA client cannot write an integer to the variables (in OpenModelica) which are having `float` values. In other words, there should not be a type mismatch while modifying the elements on the OPC UA server.

4. A Python OPC UA client could not modify the variables, which are defined as `input Boolean` in an OpenModelica model.

5. The variables defined as `input Integer` in OpenModelica do not appear on the OPC UA server.

One may consider the above-mentioned points as the scopes for improvement in the existing OPC UA implementation in OpenModelica.

8.3 Comparison of OPC UA in OpenModelica with that in other simulation tools

In the previous sections, we have noticed that the existing implementation of OPC UA server in OpenModelica flatters an array and assigns an individual node ID to each element. In this section, we will create an OPC UA server, with an array, in Python. This server will be similar to the one given in App. A.1 except that we will add two arrays here, as shown below:

```
arr1 = params.add_variable(addspace, "numArray", [1, 2, 6])
arr2 = params.add_variable(addspace, "boolArray", [True, False, True])
```

Now, we will start this server and connect it with UaExpert [13] to check how these arrays are denoted. Fig. 8.6 shows these arrays being received in the UaExpert. As shown in this figure, each array has been assigned a single node ID, unlike in OpenModelica. This representation might be suitable for some of the experiments, as we can access the array elements by their respective indices. Suppose we want to access the second element of `numArray` at the client-

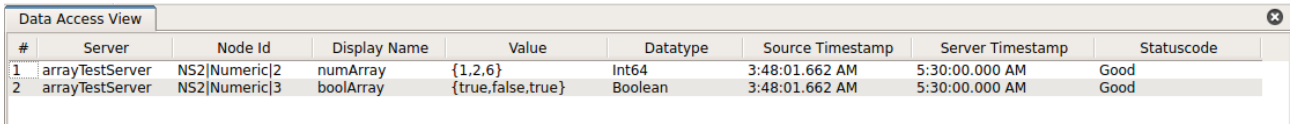

Figure 8.6: Node Ids of the arrays from the Python OPC UA server

side. For this, we can modify the client presented in App. A.1. Remember, Python follows zero-indexing. If we want to access the second element of `numArray` and the third element of `boolArray`, We can add the lines as given below:

```
numArr = client.get_node("ns = 2; i = 2")
numArrVal = numArr.get_value()

boolArr = client.get_node("ns = 2; i = 3")
boolArrVal = boolArr.get_value()

print("Second element of the numeric array: {}".\
    format(numArrVal[1]))
print("Third element of the boolean array: {}".\
    format(boolArrVal[2]))
```

Thus, it is evident that we can access any element by knowing the node ID of the array. However, this representation of the array does not support modifying an element by its index [23]. Suppose we want to modify the `numArray` to `[1, 5.6, 6]` and the `boolArray` to `[True, True, True]`. To do so, we can use the `set_value` method at the client-side, as given below:

```
numArr.set_value([1, 5.6, 6])
boolArr.set_value([True, True, True])
```

In OPC UA server in OpenModelica, we could write the individual array element by knowing its respective node ID. Thus, we can infer that the read/write methods applicable for OPC UA server in OpenModelica are different from other OPC UA servers. That's why one needs to know this difference before using OPC UA for any simulation tool.

After implementing a Python OPC UA server having arrays, we now search for other simulation tools with a built-in OPC UA server like OpenModelicahas. For this study, we considered three tools: DWSIM [24], MATLAB [25], and Scilab [26]. As far as we have learned, none of these

three simulation tools has a built-in OPC UA server [27]. That's why OpenModelica becomes the only choice for those who wish to simulate a model/plant over an OPC UA server. However, each of these tools (Scilab, DWSIM, and MATLAB) has an OPC UA client plugin that can connect to an OPC UA server. The details are as follows:

- DWSIM - The OPC UA client plugin for DWSIM enables the mapping of monitored variables in an OPC UA server to properties of DWSIM flowsheet objects [28].
- Scilab - There is a toolbox that can facilitate the implementation of the OPC client for Scilab [29]. However, this toolbox, created in 2012, is only available for Windows OS.
- MATLAB - OPC Toolbox provides an OPC UA client to connect to OPC UA servers [30].

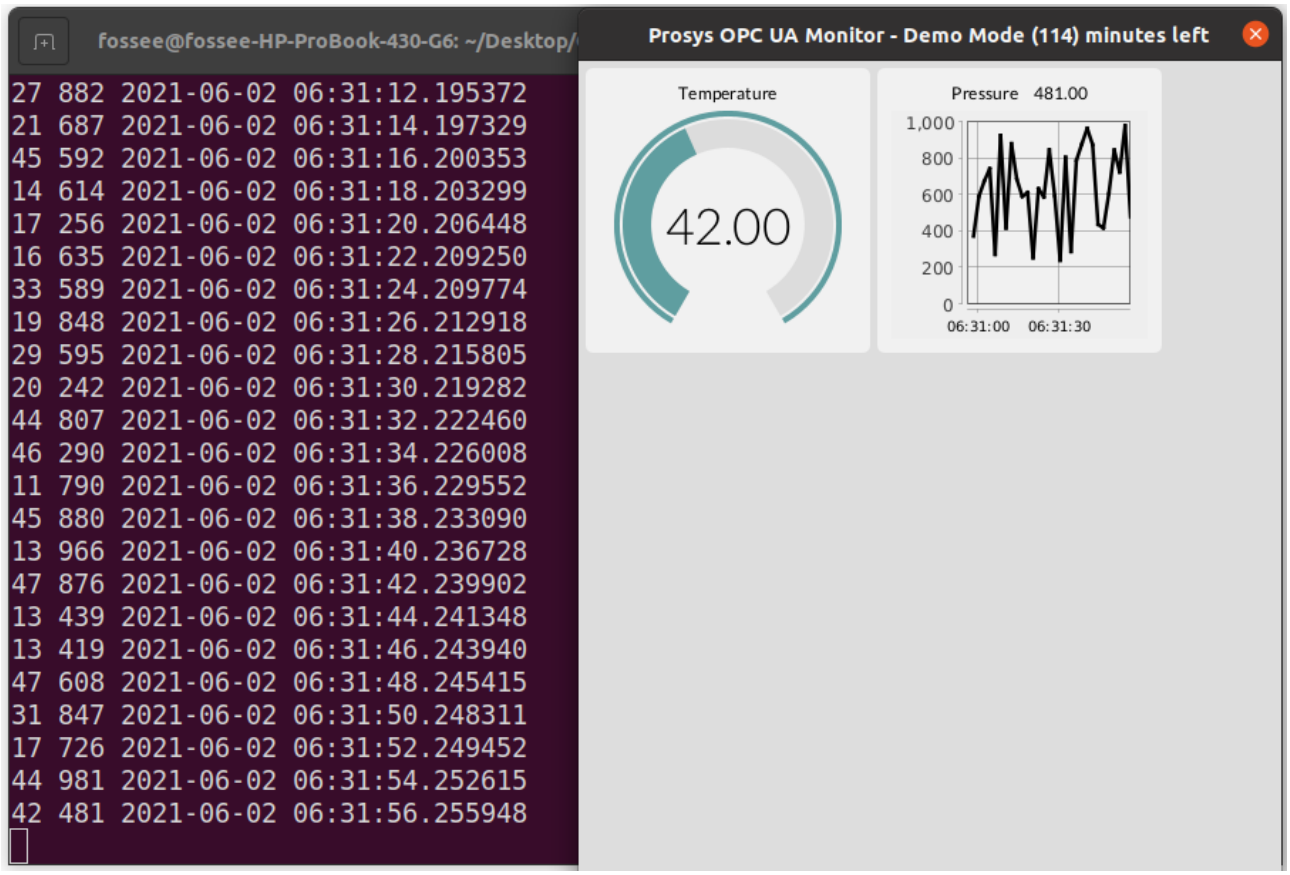

Figure 8.7: Monitoring `Temperature` and `Pressure` using Prosy OPC UA monitor

Table 8.1 summarizes the availability of OPC UA interface for these three simulation tools. As we could not find any simulation tool with a built-in OPC UA server, we opted to work with OPC UA simulation servers. There are quite a few such servers that can be deployed for testing purposes. For this project, we chose to work with Prosys OPC UA simulation server [31] and Unified Automation OPC UA servers [32]. As these are sample servers, we don't get to define our own variables. However, these tools have all possible variables inbuilt.

Prosys provides OPC UA monitor [33] along with OPC UA simulation server [31]. As the name indicates, one can use the monitor to observe the values broadcasted over a OPC UA server. For instance, we can execute the sample server presented in App. A.1 and try monitoring it with Prosys OPC UA monitor. Remember, in the sample server, we have three different variables named, `Temperature`, `Pressure`, and `Time`. Fig. 8.7 shows these values being monitored in

	DWSIM	MATLAB	OpenModelica	Scilab
OPC UA server	✗	✗	✓	✗
OPC UA client	✓	✓	✗	✓

Table 8.1: Availability of OPC UA in various simulation tools

Prosys OPC UA monitor. As shown in this figure, we are monitoring only two parameters. The left part of Fig. 8.7 shows the server being simulated in the Linux Terminal, whereas the right part shows the values being received by Prosys OPC UA monitor. Next, we will use Prosys OPC UA simulation server [31]. In this server, we can only visualize the built-in variables. Fig. 8.8 shows the built-in variables available in Prosys OPC UA simulation server. One can make use of UaExpert [13] to connect with this server and monitor the variables. As the Prosys OPC UA simulation server doesn't support adding array-type variables in the Simulation view [34], we will proceed with Unified Automation OPC UA servers [32].

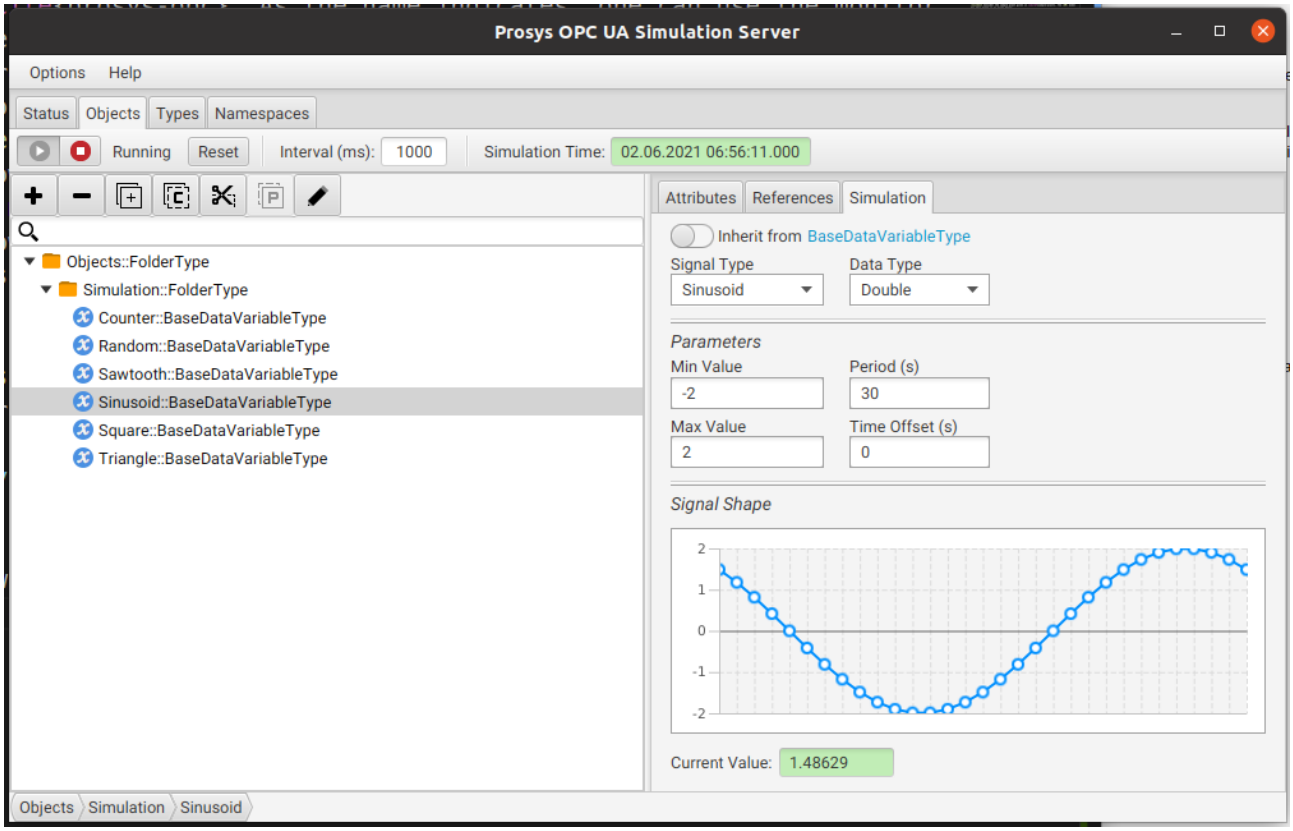

Figure 8.8: Built-in variables available in Prosys OPC UA simulation server

Unlike Prosys tools, Unified Automation OPC UA servers [32] are only available for Windows OS. For this project, we have used OPC UA ANSI C Demo Server v1.9.1 (Windows). Fig. 8.9 shows the screen appearing after this server is launched. As shown in this figure, this server creates quite a few nodes. Along with these nodes, Fig. 8.9 shows the endpoint URL on which the server has been started. We can use this endpoint URL to connect to this server. For this, we can utilize either Prosys OPC UA monitor [33] or UaExpert [13]. For this project, we make use of UaExpert connect to the server shown in Fig. 8.9. After connecting to this server, one can view the list of nodes in the address space. Fig. 8.10 shows the nodes of various data types like `Static`, `Dynamic`, etc. As we are interested in knowing how the arrays are represented, we will drag two dynamic arrays - `Float` and `Boolean` - to the Data Access View of UaExpert.

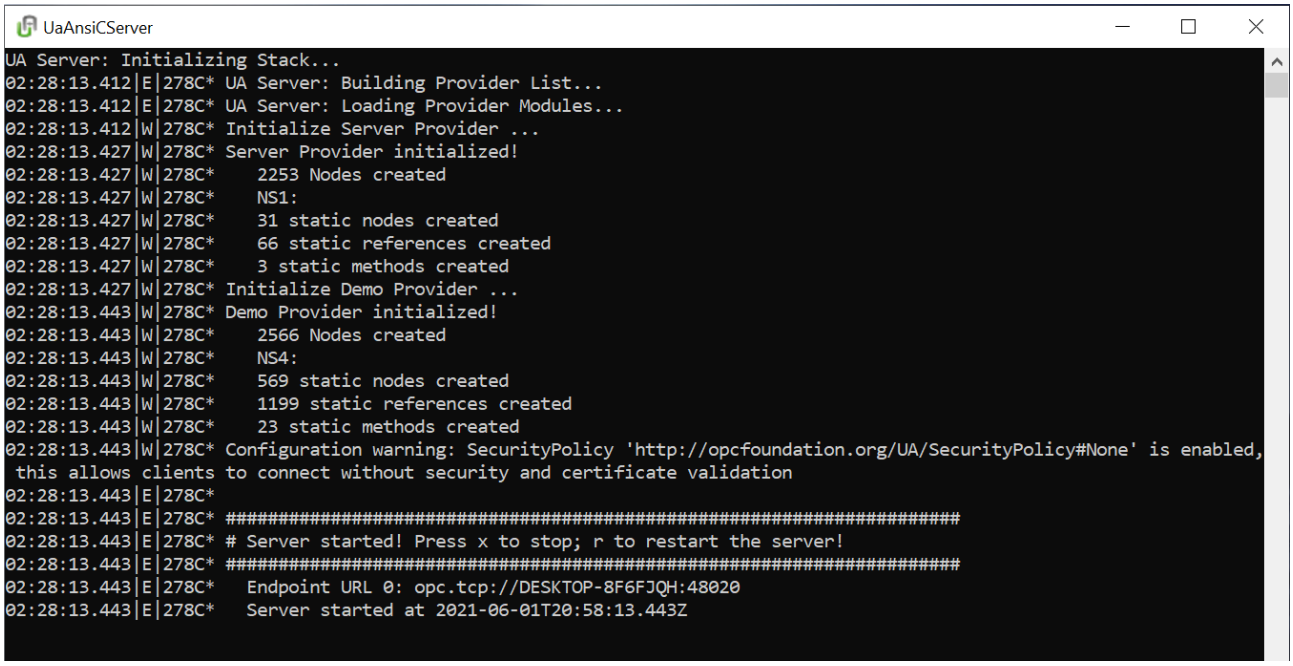

Figure 8.9: Launching Unified Automation OPC UA server

Fig. 8.11 shows these two arrays. As shown in this figure, each of these two arrays have a single node ID. That's why one can access the entire array with the help of a single node ID, unlike OpenModelica.

Thus, we can conclude that Prosys OPC UA simulation server and Unified Automation OPC UA server assign a single node ID to an array, whereas OpenModelica assigns an individual node ID to each element of an array. Therefore, we would like to note the following:

1. Out of the available simulation tools (we only checked for DWSIM, MATLAB, OpenModelica, and Scilab), OpenModelica is the unique choice for those who wish to simulate a model/plant over an OPC UA server.

2. There are quite a few OPC UA simulation servers like Prosys OPC UA simulation server, Unified Automation OPC UA server, etc. which can be utilized for testing the structures of variables.

At last, we discuss the security aspects of OPC UA server in OpenModelica. For enabling the OPC UA server in OpenModelica, we need to add the corresponding simulation flag in `simulate` function as shown in Fig. 6.2. Apart from the flag `-embeddedServer=opc-ua`, we don't have access to other arguments related to OPC UA server in OpenModelica. However, there are OPC UA servers which support adding Security Mode, User authentication, etc. [35].

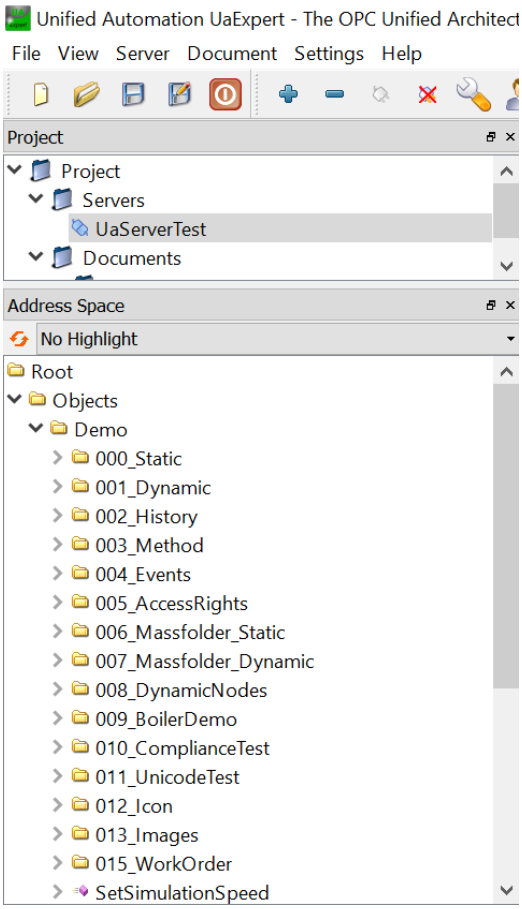

Figure 8.10: Various data types coming from Unified Automation OPC UA server

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	UaServerTest	NS4|String|Demo.Dynamic.Arrays.Float	Float	{6351,6352,6353,6354}	Float	02:35:01.335	02:35:01.335	Good
2	UaServerTest	NS4|String|Demo.Dynamic.Arrays.Boolean	Boolean	{true,false,true}	Boolean	02:34:57.077	02:34:57.077	Good

Figure 8.11: Data Access View of two dynamic arrays coming from Unified Automation OPC UA server

Chapter 9

Conclusion

In this project, we have studied how the industrial revolution has created the need for digital twins for a manufacturing plant. Along with this, we have discussed why we need to devise open-source techniques and tools by which we can simulate or program embedded systems in sync with external hardware. Thus, we have presented a proof-of-concept for implementing Hardware-in-Loop simulations using OpenModelica and a single board computer. To show the capability of open-source tools, we have simulated a few models in OpenModelica. Further, we have discussed how Modelica can simulate a wide range of processes, including electrical, mechanics, fluid, etc. Along with this, we have discussed the environments (like OMEdit OMShell, and OMPlot), which are bundled with OpenModelica. We have also discussed a standard communication protocol named OPC UA. We have also discussed the core differences between OPC-DA and OPC UA. Moreover, we have explored the built-in OPC UAserver of OpenModelica. Due to the availability of the OPC UAserver in OpenModelica, we were encouraged to design an OPC UAclient to establish Hardware-in-Loop simulation efficiently.

After exploring the features of OpenModelica, we have analyzed the subtle differences between a microprocessor and a microcontroller. We have also talked about single board computers, like Raspberry Pi. Subsequently, we have presented the differences between Pi 3 versus Pi 4. We have also briefly discussed why Pi 3 should be sufficient for the project at hand.

Having studied OpenModelica and Pi 3, we have established a client-server communication between OpenModelica and Pi 3 using OPC UAprotocol. To demonstrate this, we have simulated a two-tank system in OpenModelica, where our client could fetch all the values from the server. Next, we have implemented controllers (like P- and PID-controller) at the client-side, capable of monitoring and controlling the plant in OpenModelica.

Chapter 10

Future Work

As a next step, we wish to simulate complex models in OpenModelica. Accordingly, we would like to check whether our OPC UAclient can deal with those models. We also intend to implement advanced controllers like LQR, MPC, etc., on the client-side. Along with this, we plan to use 4diac, which provides an open-source infrastructure for distributed industrial process measurement and control systems. By harnessing the potency of 4diac, we wish to implement customized controllers for the different units of the plant. Moreover, we will explore the idea of having control laws for dealing with various situations in the plants.

Apart from the points mentioned above, we wish to work on some of the challenges we have faced while working with OpenModelica. For instance, we could read the variables and modify those variables (of a plant running in OpenModelica) from a client running on R Pi. However, we have been facing issues in viewing the plots in OMEdit after the simulation is finished. Additionally, we simulated the plant by using OMShell-terminal. In this case, we could view the simulation results in a `.mat` file. Moreover, after loading this `.mat` file in OMEdit (by using the option Open Result Files in File Menu), we could view the desired plots. Thus, we need to figure out how to simulate a plant from OMEdit and how to view the plots after the client-server communication is complete.

Appendix A

Interfacing Raspberry Pi and OpenModelica

A.1 How to establish client-server communication

A.1.1 Python code for sample server

```
1  from opcua import Server
2  from random import randint
3  import datetime
4  import time
5
6  server = Server()
7
8  # Define the IP address and a port number.
9  url = "opc.tcp://192.168.0.3:4840"
10 server.set_endpoint(url)
11
12 # Define our address space
13 name = "OPCUA_Simulation_Server"
14 addspace = server.register_namespace(name)
15
16 # Get the root node defined in our address space
17 node = server.get_objects_node()
18
19 # Define a node to store the parameters
20 params = node.add_object(addspace, "Parameters")
21
22 # Define three variables each with 0 as the initial value
23 Temp = params.add_variable(addspace, "Temperature", 0)
24 Press = params.add_variable(addspace, "Pressure", 0)
25 Time = params.add_variable(addspace, "Time", 0)
26
27 # Make these three variables writable
28 Temp.set_writable()
29 Press.set_writable()
30 Time.set_writable()
31
32 # Start the server
33 server.start()
```

```
34 print("Server started at {}".format(url))
35
36 # Assign random values to the three variables
37 while True:
38     Temperature = randint(10, 50) # random value between 10 and 50
39     Pressure = randint(200, 999) # random value between 200 and 999
40     TIME = datetime.datetime.now() # today's date and time
41
42     print(Temperature, Pressure, TIME)
43
44     # Set the values to the three variables
45     Temp.set_value(Temperature)
46     Press.set_value(Pressure)
47     Time.set_value(TIME)
48
49     # Add a delay of 2 seconds
50     time.sleep(2)
```

A.1.2 Python code for sample client

```
1 from opcua import Client
2 import time
3
4 # Write the URL on which the server is connected
5 url = "opc.tcp://192.168.0.3:4840"
6
7 client = Client(url)
8 client.connect()
9 print("Client connected!")
10
11 # Get the values from the server
12 while True:
13     # Get the node Ids from the UaExpert
14     Temp = client.get_node("ns = 2; i = 2")
15     Temperature = Temp.get_value()
16     print(Temperature)
17
18     Press = client.get_node("ns = 2; i = 3")
19     Pressure = Press.get_value()
20     print(Pressure)
21
22     TIME = client.get_node("ns = 2; i = 4")
23     TIME_Value = TIME.get_value()
24     print(TIME_Value)
25
26     time.sleep(1)
```

A.2 Connecting Pi 3 with OpenModelica

A.2.1 Two-tank model in OpenModelica

```
1 package TwoTank
2
3   model Tank
```

```
4     Real Fi(unit="m3/s") "Input flowrate to the tank";
5     parameter Real x;
6     parameter Real A(unit="m2") "Area of the tank";
7     Real F(unit="m3/s") "Output flowrate from the tank";
8     Real h(unit="m") "Tank level";
9     Real delH(unit="m");
10    parameter Real hinit(unit="m") "Initial level in the tank";
11    parameter Boolean dynamic "true(1) indicates dynamic state";
12  TwoTank.port inlet;
13  TwoTank.port outlet;
14    initial equation
15    if dynamic ==  true then
16    h = hinit;
17    end if;
18  equation
19    inlet.f = Fi;
20    outlet.f = F;
21    if dynamic == true then
22      A * der(h) = Fi - F;
23    else
24      Fi - F = 0;
25    end if;
26    F = x*sqrt(delH);
27  end Tank;
28
29  connector port
30  Real f;
31  end port;
32
33  model System
34  Tank Tank1(A = 3, dynamic = true, hinit = 4, x = 8, F(start=0.01),
       delH(start=1));
35  Tank Tank2(A = 1, dynamic = true, hinit = 2, x = 4, F(start=0.01),
       delH(start=1));
36  equation
37  connect(Tank1.outlet,Tank2.inlet);
38  Tank1.Fi = 16;
39  Tank1.delH = Tank1.h - Tank2.h;
40  Tank2.delH = Tank2.h;
41  end System;
42
43 end TwoTank;
```

A.2.2 OPC-UA client connecting to the two-tank model

```
1 from opcua import Client
2 from opcua import ua
3 import time
4
5 # Define the URL on which the server is broadcasting
6 url = "opc.tcp://192.168.0.3:4841"
7
8 if __name__ == "__main__":
9     client = Client(url)
```

```
10
11      try:
12          client.connect()
13          print("Client connected!")
14
15          enableStopTime = client.get_node(ua.NodeId(10003, 0))
16          enableStopTime.set_value(False)
17          print("Current state of enableStopTime :
                {}".format(enableStopTime.get_value()))
18
19          run = client.get_node(ua.NodeId(10001, 0))
20          run.set_value(True)
21          print("Current state of run : {}".format(run.get_value()))
22
23          root = client.get_root_node()
24          print("Root node is : ", root)
25
26          objects = client.get_objects_node()
27          print("Objects\' node is : ", objects)
28
29          while True:
30              modelicaId = {}
31              modelicaVariables = {}
32              tmp = {}
33
34              modelicaId = objects.get_children()
35
36              for i in range(len(modelicaId)):
37                  modelicaVariables[i] =
                        modelicaId[i].get_display_name().Text
38                  # modelicaVariables[i] = modelicaId[i].get_browse_name()
39
40                  # till index 5, we have values like server, run, step,
                        enableStopTime, etc.
41                  # So, we begin with 6 to have the actual parameters of
                        the plant.
42                  if (i > 5):
43                      tmp[i] = modelicaId[i].get_value()
44                      print(i, modelicaVariables[i], tmp[i])
45
46                  else:
47                      print(i, modelicaVariables[i])
48              time.sleep(2)
49              print("="*40)
50
51
52          # print(objects.get_children()[6].get_display_name().Text)
53
54      except KeyboardInterrupt:
55          print("Stopping sequence!")
56
57      finally:
58          print("Done!")
```

```
59          client.disconnect()
```

Appendix B

Implementing controllers on client

B.1 Controlling a sample system

B.1.1 Sample plant in OpenModelica

```
model samplePlant
  input Real x;
  Real y;

equation
  y = x;
end samplePlant;
```

B.1.2 OPC UA client with P controller to control the sample plant

```
from opcua import Client
from opcua import ua
import time
import logging

# Define the URL on which the server is broadcasting
url = "opc.tcp://192.168.0.3:4841"

# 0 Server
# 1 step
# 2 run
# 3 realTimeScalingFactor
# 4 enableStopTime
# 5 time
# 6 x 0.0
# 7 y 0.0


if __name__ == "__main__":
    client = Client(url)
    logging.basicConfig(level=logging.WARN)

    try:
        client.connect()
        print("Client connected!")

```

```
27          enableStopTime = client.get_node(ua.NodeId(10003, 0))
28          # enableStopTime.set_value(False)
29          print("Current state of enableStopTime :
                {}".format(enableStopTime.get_value()))
30
31          run = client.get_node(ua.NodeId(10001, 0))
32          run.set_value(True)
33          print("Current state of run : {}".format(run.get_value()))
34
35          root = client.get_root_node()
36          print("Root node is : ", root)
37
38          objects = client.get_objects_node()
39          print("Objects\' node is : ", objects)
40
41          # Find the IDs for MV, PV
42          writeID = 6 # x
43          readID = 7 # y
44
45          modelicaId = {}
46          modelicaId = objects.get_children()
47
48          # Desired setpoint
49          set_point = 10
50
51          while True:
52              # Evaluate the PV and MV
53              error = set_point - modelicaId[readID].get_value()
54              valWrite = 0.7 * error
55              print("Error = {}, valWrite = {}".format(error, valWrite))
56
57              if (error > 0):
58                  modelicaId[writeID].set_value(valWrite)
59
60              else:
61                  print("Setpoint achieved!\n")
62
63              print("y value is: ", modelicaId[readID].get_value())
64              print("x value is: ", modelicaId[writeID].get_value())
65              time.sleep(1)
66              print("="*40)
67
68      except KeyboardInterrupt:
69          print("Stopping sequence!")
70
71      finally:
72          print("Done!")
73          client.disconnect()
```

B.1.3 OPC UA client with PID controller to control the sample plant

```
1 from opcua import Client
2 from opcua import ua
```

```
3 import time
4 import logging
5
6
7 # Define the URL on which the server is broadcasting
8 url = "opc.tcp://192.168.0.3:4841"
9
10 def PID(Kp, Ki, Kd, MV_bar = 0):
11
12     # Initialize the stored data
13     e_prev, t_prev = 0, 0
14     I = 0
15
16     MV = MV_bar
17
18     while True:
19         # yield MV, wait for new t, PV, SP
20         t, PV, SP = yield MV
21
22         # Error calculation
23         e = SP - PV
24
25         P = Kp * e
26         I = I + Ki * e * (t - t_prev)
27         D = Kd * (e - e_prev) / (t - t_prev)
28
29         MV = MV_bar + P + I + D
30
31         # update stored data for next iteration
32         e_prev, t_prev = e, t
33
34
35 if __name__ == "__main__":
36     client = Client(url)
37     logging.basicConfig(level=logging.WARN)
38
39     try:
40         client.connect()
41         print("Client Connected")
42
43         run = client.get_node(ua.NodeId(10001, 0))
44         run.set_value(True)
45         print("Current state of run : {}".format(run.get_value()))
46
47         # root = client.get_root_node()
48         # print("Root node is : ", root)
49
50         objects = client.get_objects_node()
51         # print("Objects\' node is : ", objects)
52
53         # Find the IDs for MV, PV
54         writeID = 6 # x
55         readID = 7 # y
```

```
56
57          modelicaId = {}
58          modelicaId = objects.get_children()
59
60          # Desired setpoint
61          SP = 10
62
63          # Initialize the controller
64          controller = PID(0.2, 0.1, 0.2)
65          controller.send(None)
66
67          t = 0 # Just a counter
68
69          while True:
70              t = t + 1
71              # Evaluate the PV and MV
72              PV = modelicaId[readID].get_value()
73              MV = controller.send([t, PV, SP])
74
75              modelicaId[writeID].set_value(MV)
76
77              print("y value is: ", modelicaId[readID].get_value())
78              print("x value is: ", modelicaId[writeID].get_value())
79              time.sleep(1)
80              print("="*40)
81
82      except KeyboardInterrupt:
83          print("Stopping sequence!")
84
85      finally:
86          print("Done!")
87          client.disconnect()
```

B.2 Controlling a single tank system

B.2.1 Single tank system in OpenModelica

```
1  package singleTank
2    model Tank
3      // Tank related variables and parameters
4      parameter Real flowLevel(unit = "m3/s") = 0.02;
5      parameter Real A(unit = "m2") = 1 "Area of the tank";
6      parameter Real flowGain(unit = "m2/s") = 0.05;
7      Real h(start = 0, unit = "m") "Tank level";
8      Real Qi(unit = "m3/s") "Flow through input valve";
9      Real Qo(unit = "m3/s") "Flow through output valve";
10     // Controller related variables and parameters
11     parameter Real K = 2 "Gain";
12     parameter Real T(unit = "s") = 10 "Time constant";
13     parameter Real minV = 0, maxV = 10;
14     // Limits for flow output
15     Real ref = 0.50 "Reference level for control";
16     Real error "Deviation from reference level";
```

```
    Real outCtr "Control signal without limiter";
    Real x "State variable for controller";
  equation
    assert(minV >= 0, "minV must be greater than or equal to zero");
    der(h) = (Qi - Qo) / A;
// Mass balance equation
    Qi = if time > 150 then 3 * flowLevel else flowLevel;
    Qo = singleTank.LimitValue(minV, maxV, -flowGain * outCtr);
    error = ref - h;
    der(x) = error / T;
    outCtr = K * (error + x);
  end Tank;

  function LimitValue
    input Real pMin;
    input Real pMax;
    input Real p;
    output Real pLim;
  algorithm
    pLim := if p > pMax then pMax
            else if p < pMin then pMin
            else p;
  end LimitValue;
end singleTank;
```

B.2.2 Single tank system (to be controlled via Python OPC UA client) in OpenModelica

```
package singleTankExt
  model Tank
    // Tank related variables and parameters
    parameter Real flowLevel(unit = "m3/s") = 0.02;
    parameter Real A(unit = "m2") = 1 "Area of the tank";
    parameter Real flowGain(unit = "m2/s") = 0.05;
    Real h(start = 0, unit = "m") "Tank level";
    Real Qi(unit = "m3/s") "Flow through input valve";
    Real Qo(unit = "m3/s") "Flow through output valve";
    // Controller related variables and parameters
    //  parameter Real K = 2 "Gain";
    //  parameter Real T(unit = "s") = 10 "Time constant";
    parameter Real minV = 0, maxV = 10;
    // Limits for flow output
    Real ref = 0.50 "Reference level for control";
    //  Real error "Deviation from reference level";
    input Real outCtr "Control signal without limiter";
    //  Real x "State variable for controller";
  equation
    assert(minV >= 0, "minV must be greater than or equal to zero");
    der(h) = (Qi - Qo) / A;
// Mass balance equation
    Qi = if time > 150 then 3 * flowLevel else flowLevel;
    Qo = singleTankExt.LimitValue(minV, maxV, -flowGain * outCtr);
//  error = ref - h;
//  der(x) = error / T;
```

```
27 //  outCtr = K * (error + x);
28   end Tank;
29
30   function LimitValue
31     input Real pMin;
32     input Real pMax;
33     input Real p;
34     output Real pLim;
35   algorithm
36     pLim := if p > pMax then pMax
37             else if p < pMin then pMin
38             else p;
39   end LimitValue;
40 end singleTankExt;
```

B.2.3 OPC UA client with PI controller to control the single tank system

```
1 from opcua import Client
2 from opcua import ua
3 import time
4 import logging
5
6
7 # Define the URL on which the server is broadcasting
8 url = "opc.tcp://192.168.0.3:4841"
9
10 def PI(Kp, Ki, MV_bar = 0):
11
12     # Initialize stored data
13     t_prev, I = 0, 0
14     MV = MV_bar
15
16     while True:
17         # yield MV, wait for new t, PV, SP
18         t, PV, SP = yield MV
19
20         # Error calculation
21         error = SP - PV
22
23         P = Kp * error
24         I = I + Ki * error * (t - t_prev)
25
26         MV = MV_bar + P + I
27
28         # update stored data for next iteration
29         t_prev = t
30
31
32 if __name__ == "__main__":
33     client = Client(url)
34     logging.basicConfig(level=logging.WARN)
35
36     try:
```

```
37          client.connect()
38          print("Client Connected")
39
40          run = client.get_node(ua.NodeId(10001, 0))
41          run.set_value(True)
42          print("Current state of run : {}".format(run.get_value()))
43
44          objects = client.get_objects_node()
45
46          # Find the IDs for MV, PV
47          writeID = 10 # Controller output
48          readID = 6 # h
49
50          modelicaId = {}
51          modelicaId = objects.get_children()
52
53          # Desired height of the liquid in the tank
54          SP = 0.50
55
56          # Initialize the controller
57          controller = PI(2, 0.2)
58          controller.send(None)
59
60          t = 0 # Just a counter
61
62          while True:
63              t = t + 1
64              # Evaluate the PV and MV
65              PV = modelicaId[readID].get_value()
66              MV = controller.send([t, PV, SP])
67
68              modelicaId[writeID].set_value(MV)
69
70              print("h value is: ", modelicaId[readID].get_value())
71              print("Controller output is: ",
                    modelicaId[writeID].get_value())
72
73              time.sleep(1)
74              print("="*40)
75
76      except KeyboardInterrupt:
77          print("Stopping sequence!")
78
79      finally:
80          print("Done!")
81          client.disconnect()
```

B.3 Controlling a connected tanks system

B.3.1 Connected tanks system in OpenModelica

```
1 package connectedTanks
2   model TwoTanks
```

```
3   // Tank related variables and parameters
4     parameter Real flowLevel(unit = "m3/s") = 0.02;
5     parameter Real tank1_A(unit = "m2") = 1;
6     parameter Real tank2_A(unit = "m2") = 1.3;
7     parameter Real flowGain(unit = "m2/s") = 0.05;
8
9     Real tank1_h(start = 0.0, unit = "m"), tank2_h(start = 0.0, unit =
          "m")  "Level of the two tanks";
10    Real tank1_Qi(unit = "m3/s"), tank2_Qi(unit = "m3/s") "Flow through
          input valves of the two tanks";
11    Real tank1_Qo(unit = "m3/s"), tank2_Qo(unit = "m3/s") "Flow through
          output valves of the two tanks";
12
13    // Controller related variables and parameters
14    parameter Real K = 2 "Gain";
15    parameter Real T(unit = "s") = 10 "Time constant";
16    parameter Real minV = 0, maxV = 10;      // Limits for output valve
          flow
17
18    // Limits for flow output
19    Real tank1_ref = 0.25, tank2_ref = 0.40 "Reference levels for
          control";
20    Real tank1_error, tank2_error "Deviation from reference level";
21
22    Real tank1_outCtr, tank2_outCtr "Control signal without limiter";
23
24    Real tank1_x, tank2_x "State variables for the two controllers";
25    Real tank1_y, tank2_y "State variables for the two controllers";
26
27  equation
28    assert(minV >= 0, "minV - minimum valve level must be greater than
          or equal to zero");
29
30    der(tank1_h) = (tank1_Qi - tank1_Qo) / tank1_A;
31    der(tank2_h) = (tank2_Qi - tank2_Qo) / tank2_A;
32
33    tank1_Qi = if time > 150 then 3 * flowLevel else flowLevel;
34    tank2_Qi = tank1_Qo;
35
36    tank1_Qo = connectedTanks.LimitValue(minV, maxV, -flowGain *
          tank1_outCtr);
37    tank2_Qo = connectedTanks.LimitValue(minV, maxV, -flowGain *
          tank2_outCtr);
38
39    tank1_error = tank1_ref - tank1_h;
40    tank2_error = tank2_ref - tank2_h;
41
42    der(tank1_x) = tank1_error / T;
43    tank1_y = T * der(tank1_error);
44    tank1_outCtr = K * (tank1_error + tank1_x + tank1_y);
45
46    der(tank2_x) = tank2_error / T;
47    tank2_y = T * der(tank2_error);
```

```
48      tank2_outCtr = K * (tank2_error + tank2_x + tank2_y);
49
50    end TwoTanks;
51
52    function LimitValue
53      input Real pMin;
54      input Real pMax;
55      input Real p;
56      output Real pLim;
57    algorithm
58      pLim := if p > pMax then pMax
59              else if p < pMin then pMin
60              else p;
61    end LimitValue;
62  end connectedTanks;
```

B.3.2 Connected tanks system (to be controlled via Python OPC UA client) in OpenModelica

```
1  package connectedTanksExt
2    model TwoTanks
3    // Tank related variables and parameters
4      parameter Real flowLevel(unit = "m3/s") = 0.02;
5      parameter Real tank1_A(unit = "m2") = 1;
6      parameter Real tank2_A(unit = "m2") = 1.3;
7      parameter Real flowGain(unit = "m2/s") = 0.05;
8
9      Real tank1_h(start = 0.0, unit = "m"), tank2_h(start = 0.0, unit =
         "m")  "Level of the two tanks";
10     Real tank1_Qi(unit = "m3/s"), tank2_Qi(unit = "m3/s") "Flow through
         input valves of the two tanks";
11     Real tank1_Qo(unit = "m3/s"), tank2_Qo(unit = "m3/s") "Flow through
         output valves of the two tanks";
12
13     // Controller related variables and parameters
14   //  parameter Real K = 2 "Gain";
15   //  parameter Real T(unit = "s") = 10 "Time constant";
16     parameter Real minV = 0, maxV = 10;      // Limits for output valve
         flow
17
18     // Limits for flow output
19     Real tank1_ref = 0.25, tank2_ref = 0.40 "Reference levels for
         control";
20   //  Real tank1_error, tank2_error "Deviation from reference level";
21
22     input Real tank1_outCtr, tank2_outCtr "Control signal without
         limiter";
23
24   //  Real tank1_x, tank2_x "State variables for the two controllers";
25   //  Real tank1_y, tank2_y "State variables for the two controllers";
26
27   equation
28     assert(minV >= 0, "minV - minimum valve level must be greater than
         or equal to zero");
```

```
29
30      der(tank1_h) = (tank1_Qi - tank1_Qo) / tank1_A;
31      der(tank2_h) = (tank2_Qi - tank2_Qo) / tank2_A;
32
33      tank1_Qi = if time > 150 then 3 * flowLevel else flowLevel;
34      tank2_Qi = tank1_Qo;
35
36      tank1_Qo = connectedTanksExt.LimitValue(minV, maxV, -flowGain *
            tank1_outCtr);
37      tank2_Qo = connectedTanksExt.LimitValue(minV, maxV, -flowGain *
            tank2_outCtr); //  tank1_error = tank1_ref - tank1_h;
38  //  tank2_error = tank2_ref - tank2_h;
39  //  der(tank1_x) = tank1_error / T;
40  //  tank1_y = T * der(tank1_error);
41  //  tank1_outCtr = K * (tank1_error + tank1_x + tank1_y);
42  //  der(tank2_x) = tank2_error / T;
43  //  tank2_y = T * der(tank2_error);
44  //  tank2_outCtr = K * (tank2_error + tank2_x + tank2_y);
45    end TwoTanks;
46
47    function LimitValue
48      input Real pMin;
49      input Real pMax;
50      input Real p;
51      output Real pLim;
52    algorithm
53      pLim := if p > pMax then pMax else if p < pMin then pMin else p;
54    end LimitValue;
55  end connectedTanksExt;
```

B.3.3 OPC UA client with PID controller to control the connected tanks system

```
1  from opcua import Client
2  from opcua import ua
3  import time
4  import logging
5
6
7  # Define the URL on which the server is broadcasting
8  url = "opc.tcp://192.168.0.3:4841"
9
10 class PID:
11      def __init__(self, Kp, Ki, Kd, MV_bar = 0):
12          self.Kp = Kp
13          self.Ki = Ki
14          self.Kd = Kd
15          self.MV_bar = MV_bar
16
17      def PID_cntr(self):
18
19          # Initialize stored data
20          t_prev, PV_prev = 0, 0
21          # error_prev, t_prev = 0, 0
```

```
22
23          I = 0
24
25          MV = self.MV_bar
26
27          while True:
28              # yield MV, wait for new t, PV, SP
29              t, PV, SP = yield MV
30
31              # Error calculation
32              error = SP - PV
33
34              P = self.Kp * error
35              I = I + self.Ki * error * (t - t_prev)
36              D = - self.Kd * (PV - PV_prev) / (t - t_prev)
37              # D = Kd * (error - error_prev) / (t - t_prev)
38
39              MV = self.MV_bar + P + I + D
40
41              # update stored data for next iteration
42              # error_prev = error
43              t_prev = t
44              # error_prev = error
45              PV_prev = PV
46
47
48  if __name__ == "__main__":
49      client = Client(url)
50      logging.basicConfig(level=logging.WARN)
51
52      try:
53          client.connect()
54          print("Client Connected")
55
56          run = client.get_node(ua.NodeId(10001, 0))
57          run.set_value(True)
58          print("Current state of run : {}".format(run.get_value()))
59
60          objects = client.get_objects_node()
61
62          # Find the IDs for MV, PV
63          tank1_h, tank2_h = 6, 7
64          tank1_outCtr, tank2_outCtr = 11, 15
65
66          modelicaId = {}
67          modelicaId = objects.get_children()
68
69          # Desired heights of the liquid in each tank
70          tank1_ref, tank2_ref = 0.25, 0.40
71
72          # Initialize the controllers
73          controller1 = PID(2, 0.2, 20)
74          var1 = controller1.PID_cntr()
```

```
        var1.send(None)

        controller2 = PID(2, 0.2, 20)
        var2 = controller2.PID_cntr()
        var2.send(None)

        t = 0  # Just a counter

        while True:
            t = t + 1
            # Evaluate PVs for each of the two tanks
            PV1 = modelicaId[tank1_h].get_value()
            PV2 = modelicaId[tank2_h].get_value()

            # Evaluate and modify the MVs for each of the two tanks
            MV1 = var1.send([t, PV1, tank1_ref])
            MV2 = var2.send([t, PV2, tank2_ref])

            modelicaId[tank1_outCtr].set_value(MV1)
            modelicaId[tank2_outCtr].set_value(MV2)

            print("Tank 1: h value is: ",
                modelicaId[tank1_h].get_value())
            print("Tank 1: Controller output is: ",
                modelicaId[tank1_outCtr].get_value())

            print("Tank 2: h value is: ",
                modelicaId[tank2_h].get_value())
            print("Tank 2: Controller output is: ",
                modelicaId[tank2_outCtr].get_value())

            time.sleep(1)
            print("="*40)

    except KeyboardInterrupt:
        print("Stopping sequence!")

    finally:
        print("Done!")
        client.disconnect()
```

B.4 Modeling hybrid systems

B.4.1 Single tank system in OpenModelica with discrete PI-controller

```
package singleTankDis
  model Tank
    // Tank related variables and parameters
    parameter Real flowLevel(unit = "m3/s") = 0.02;
    parameter Real A(unit = "m2") = 1 "Area of the tank";
    parameter Real flowGain(unit = "m2/s") = 0.05;
    Real h(start = 0, unit = "m") "Tank level";
    Real Qi(unit = "m3/s") "Flow through input valve";
```

```
    Real Qo(unit = "m3/s") "Flow through output valve";
    // Controller related variables and parameters
    parameter Real K = 2 "Gain";
    parameter Real T(unit = "s") = 10 "Time constant";
    parameter Real Ts(unit = "s") = 0.1 "Time between samples";
    parameter Real minV = 0, maxV = 10;
    // Limits for flow output
    parameter Real ref = 0.50 "Reference level for control";
    Real error "Deviation from reference level";
    Real outCtr "Control signal without limiter";
    discrete Real x "State variable for discrete controller";
  equation
    assert(minV >= 0, "minV must be greater than or equal to zero");
    der(h) = (Qi - Qo) / A;
  // Mass balance equation
    Qi = if time > 150 then 3 * flowLevel else flowLevel;
    Qo = singleTankDis.LimitValue(minV, maxV, -flowGain * outCtr);
    error = ref - h;
    when sample(0, Ts) then
      x = pre(x) + (error * Ts / T);
      outCtr = K * (error + x);
    end when;
  end Tank;

  function LimitValue
    input Real pMin;
    input Real pMax;
    input Real p;
    output Real pLim;
  algorithm
    pLim := if p > pMax then pMax
            else if p < pMin then pMin
            else p;
  end LimitValue;
end singleTankDis;
```

B.4.2 Single tank system (to be controlled via Python OPC UA client) in OpenModelica

```
package singleTankDisExt
  model Tank
    // Tank related variables and parameters
    parameter Real flowLevel(unit = "m3/s") = 0.02;
    parameter Real A(unit = "m2") = 1 "Area of the tank";
    parameter Real flowGain(unit = "m2/s") = 0.05;
    Real h(start = 0, unit = "m") "Tank level";
    Real Qi(unit = "m3/s") "Flow through input valve";
    Real Qo(unit = "m3/s") "Flow through output valve";
    // Controller related variables and parameters
    parameter Real K = 2 "Gain";
    parameter Real T(unit = "s") = 10 "Time constant";
    parameter Real Ts(unit = "s") = 0.1 "Time between samples";
    parameter Real minV = 0, maxV = 10;
    // Limits for flow output
```

```
16      parameter Real ref = 0.50 "Reference level for control";
17      //  Real error "Deviation from reference level";
18      input Real outCtr "Control signal without limiter";
19      //  discrete Real x "State variable for discrete controller";
20    equation
21      assert(minV >= 0, "minV must be greater than or equal to zero");
22      der(h) = (Qi - Qo) / A;
23  // Mass balance equation
24      Qi = if time > 150 then 3 * flowLevel else flowLevel;
25      Qo = singleTankDisExt.LimitValue(minV, maxV, -flowGain * outCtr);
26    //  error = ref - h;
27    //  when sample(0, Ts) then
28    //    x = pre(x) + (error * Ts / T);
29    //    outCtr = K * (error + x);
30    //  end when;
31    end Tank;
32
33    function LimitValue
34      input Real pMin;
35      input Real pMax;
36      input Real p;
37      output Real pLim;
38    algorithm
39      pLim := if p > pMax then pMax
40              else if p < pMin then pMin
41              else p;
42    end LimitValue;
43  end singleTankDisExt;
```

B.4.3 OPC UA client with discrete PI controller to control the single tank system

```
1  from opcua import Client
2  from opcua import ua
3  from tclab import clock
4  import time
5  import logging
6
7
8  # Define the URL on which the server is broadcasting
9  url = "opc.tcp://192.168.0.3:4841"
10
11 def PI(Kp, Ki, MV_bar = 0):
12
13     # Initialize stored data
14     t_prev, I = 0, 0
15     MV = MV_bar
16
17     while True:
18         # yield MV, wait for new t, PV, SP
19         t, PV, SP = yield MV
20
21         # Error calculation
22         error = SP - PV
```

```
23
24          P = Kp * error
25          I = I + Ki * error * (t - t_prev)
26
27          MV = MV_bar + P + I
28
29          # update stored data for next iteration
30          t_prev = t
31
32
33  if __name__ == "__main__":
34      client = Client(url)
35      logging.basicConfig(level=logging.WARN)
36
37      try:
38          client.connect()
39          print("Client Connected")
40
41          run = client.get_node(ua.NodeId(10001, 0))
42          run.set_value(True)
43          print("Current state of run : {}".format(run.get_value()))
44
45          objects = client.get_objects_node()
46
47          # Find the IDs for MV, PV
48          writeID = 10 # Controller output
49          readID = 6 # h
50
51          modelicaId = {}
52          modelicaId = objects.get_children()
53
54          # Desired height of the liquid in the tank
55          SP = 0.50
56
57          # Initialize the controller
58          controller = PI(2, 0.2)
59          controller.send(None)
60
61          tfinal = 350
62          Ts = 0.1
63
64          for t in clock(tfinal, Ts):
65              # Evaluate the PV and MV
66              PV = modelicaId[readID].get_value()
67              MV = controller.send([t, PV, SP])
68
69              modelicaId[writeID].set_value(MV)
70
71              print("h value is: ", modelicaId[readID].get_value())
72              print("Controller output is: ",
                    modelicaId[writeID].get_value())
73
74              print("="*40)
```

```
75
76      except KeyboardInterrupt:
77          print("Stopping sequence!")
78
79      finally:
80          print("Done!")
81          client.disconnect()
```

B.4.4 Connected tanks system in OpenModelica with discrete PID controller

```
1  package connectedTanksDis
2    model TwoTanks
3    // Tank related variables and parameters
4      parameter Real flowLevel(unit = "m3/s") = 0.02;
5      parameter Real tank1_A(unit = "m2") = 1;
6      parameter Real tank2_A(unit = "m2") = 1.3;
7      parameter Real flowGain(unit = "m2/s") = 0.05;
8
9      Real tank1_h(start = 0.0, unit = "m"), tank2_h(start = 0.0, unit =
         "m")  "Level of the two tanks";
10     Real tank1_Qi(unit = "m3/s"), tank2_Qi(unit = "m3/s") "Flow through
         input valves of the two tanks";
11     Real tank1_Qo(unit = "m3/s"), tank2_Qo(unit = "m3/s") "Flow through
         output valves of the two tanks";
12
13     // Controller related variables and parameters
14     parameter Real K = 2 "Gain";
15     parameter Real T(unit = "s") = 10 "Time constant";
16     parameter Real Ts(unit = "s") = 0.1 "Time between samples";
17     parameter Real minV = 0, maxV = 10;      // Limits for output valve
         flow
18
19     // Limits for flow output
20     Real tank1_ref = 0.25, tank2_ref = 0.40 "Reference levels for
         control";
21     Real tank1_error, tank2_error "Deviation from reference level";
22
23     Real tank1_outCtr, tank2_outCtr "Control signal without limiter";
24
25     discrete Real tank1_x, tank2_x "State variables for the two
         controllers";
26     discrete Real tank1_y, tank2_y "State variables for the two
         controllers";
27
28   equation
29     assert(minV >= 0, "minV - minimum valve level must be greater than
         or equal to zero");
30
31     der(tank1_h) = (tank1_Qi - tank1_Qo) / tank1_A;
32     der(tank2_h) = (tank2_Qi - tank2_Qo) / tank2_A;
33
34     tank1_Qi = if time > 150 then 3 * flowLevel else flowLevel;
35     tank2_Qi = tank1_Qo;
```

```
36
37      tank1_Qo = connectedTanksDis.LimitValue(minV, maxV, -flowGain *
           tank1_outCtr);
38      tank2_Qo = connectedTanksDis.LimitValue(minV, maxV, -flowGain *
           tank2_outCtr);
39
40      tank1_error = tank1_ref - tank1_h;
41      tank2_error = tank2_ref - tank2_h;
42
43      when sample(0, Ts) then
44    //controller equations for tank1
45        tank1_x = pre(tank1_x) + (tank1_error * Ts / T);
46        tank1_y = T * (tank1_error - pre(tank1_error));
47        tank1_outCtr = K *(tank1_error + tank1_x + tank1_y);
48    //controller equations for tank2
49        tank2_x = pre(tank2_x) + (tank2_error * Ts / T);
50        tank2_y = T * (tank2_error - pre(tank2_error));
51        tank2_outCtr = K *(tank2_error + tank2_x + tank2_y);
52      end when;
53
54    end TwoTanks;
55
56    function LimitValue
57      input Real pMin;
58      input Real pMax;
59      input Real p;
60      output Real pLim;
61    algorithm
62      pLim := if p > pMax then pMax else if p < pMin then pMin else p;
63    end LimitValue;
64  end connectedTanksDis;
```

B.4.5 Batch distillation column in OpenModelica

```
1  model FlatControlTBReal
2  parameter Integer C = 3 "No of Components", N('No of Trays') = 40 "No of
       Trays";
3    parameter Real XB0[C] = {0.3, 0.3, 0.4} "Initial Mole Fraction",
        alpha[C] = {9, 3, 1} "Relative Votality";
4    parameter Real HB0 = 300 "Initial Charge to Still", HD = 10 "Reflux
        Drum Hold Up", HN = 1 "Tray Hold Up", V = 100 / HR "Vapor Boil up
        rate", XD1SP = 0.95, XD2SP = 0.95, XB3SP = 0.95 "Specified Product
        Purity", RR = 1.22 "Reflux Ratio", HR = 3600 "TIme Conversion
        factor", Epsilon = 1e-5, simulationScalingFactor = 1;
5    Real HB "Still Hold up", XB[C](each min = 0, each max = 1) "Mole
        fraction in still", YB[C](each min = 0, each max = 1), L "Liquid
        Flow Rate", X[N, C](each min = 0, each max = 1) "Comp on Trays",
        Y[N, C](each min = 0, each max = 1), XD[C](each min = 0, each max =
        1) "Composition of Distillate", D "Distillate flow rate";
6    //=============================================================================
7    Real flagProduct[C - 1](each start = -1), flagSlop[C - 1](each start =
        -1) "flagProduct signalling collection of Product 1 and Product 2.
        flagSlop signalling collection of Slop 1 and Slop 2";
8    Real timeProduct[C](each start = 0), timeSlop[C - 1](each start = 0)
```

```
      "Times at which products and slops are withdrawn";
  Real product[C - 1](each start = 3e-5), slop[C - 1](each start = 3e-5)
      "Holdup tanks for Products and slops";
  Real productComponents[C - 1, C] "Amounts collected in Product tanks";
  Real heavyKeyComponentMoles "Quantity (moles) of h.k. component
      collected in still pot";
  Real sumProduct[C - 1] "Total number of moles collected in product
      tanks";
  Real averageProductMoleFraction[C - 1, C] "Average concentrations
      comp. in product tanks";
  Real slopComponents[C - 1, C] "Amounts of comp collected in slop
      tanks";
  Real sumSlop[C - 1] "Total number of moles collected in slop tanks";
  Real averageSlopMoleFraction[C - 1, C] "Average concentrations comp in
      slop tanks";
//  Boolean valveProduct[C - 1](each start = false), valveSlop[C -
    1](each start = false) "On/Off valves for the product and slop tanks";
  Real valveProduct[C - 1], valveSlop[C - 1] "On/Off valves for the
      product and slop tanks";
  //=============================================================================
initial equation
  for i in 1:C - 1 loop
    for j in 1:C loop
      productComponents[i, j] = Epsilon "Initial amounts(moles) present
          in product tanks.";
    end for;
  end for;
  for i in 1:C - 1 loop
    for j in 1:C loop
      slopComponents[i, j] = Epsilon "Initial amounts(moles) present in
          slop tanks.";
    end for;
  end for;
//-----------------------------------------------------------------------------
  HB = HB0 - HD - N*HN - D;
  XB[1] = 0.3;
  XB[2] = 0.3;
  XB[3] = 0.4;
  for n in 2:N - 1 loop
    X[n, 1] = 0.3;
    X[n, 2] = 0.3;
    X[n, 3] = 0.4;
  end for;
  XD[1] = 0.3;
  XD[2] = 0.3;
  XD[3] = 0.4;
  X[N, 1] = 0.3;
  X[N, 2] = 0.3;
  X[N, 3] = 0.4;
  X[1, 1] = 0.3;
  X[1, 2] = 0.3;
  X[1, 3] = 0.4;
//=============================================================================
```

```
51 equation
52 /* ====Conditions for getting Time of Products and amount of Heavy
       Key==== */
53   for i in 1:C - 1 loop
54     when flagProduct[i] == 1 then
55       timeProduct[i] = time;
56     end when;
57     when flagSlop[i] == 1 then
58       timeSlop[i] = time;
59     end when;
60   end for;
61   when valveSlop[C - 1] == 0.0 and timeSlop[C - 1] > 0 then
62     timeProduct[C] = time;
63   end when;
64   when timeProduct[C] > 0 then
65     heavyKeyComponentMoles = HB * XB[3] "Amount of H.K. component
         recovered";
66   end when;
67 //-------------------------------------------------------------------------
68 /* ========== "Amounts ofcomp collected in tank product1 at any point
       in time"========*/
69   for i in 1:C - 1 loop
70     if valveProduct[i] == 1.0 then
71 // Product valve is open.
72       flagProduct[i] = 1 "Removing product 1";
73       flagSlop[i] = -1;
74       der(product[i]) = D;
75       der(slop[i]) = 0;
76       for j in 1:C loop
77         der(productComponents[i, j]) = D * XD[j];
78         der(slopComponents[i, j]) = 0;
79       end for;
80     elseif valveSlop[i] == 1.0 then
81 // Slop Valve is open.
82       flagProduct[i] = -1;
83       flagSlop[i] = 1 "Removing Slop 1";
84       der(product[i]) = 0;
85       der(slop[i]) = D;
86       for j in 1:C loop
87         der(productComponents[i, j]) = 0;
88         der(slopComponents[i, j]) = D * XD[j];
89       end for;
90     else
91       flagProduct[i] = -1;
92       flagSlop[i] = -1;
93       der(product[i]) = 0;
94       der(slop[i]) = 0;
95       for j in 1:C loop
96         der(productComponents[i, j]) = 0;
97         der(slopComponents[i, j]) = 0;
98       end for;
99     end if;
100  end for;
```

```
101 //-------------------------------------------------------------------------------
102 /*============= Total Amount of Products and Components in Slop and
      Products Tanks=================*/
103   for i in 1:C - 1 loop
104     sumProduct[i] = sum(productComponents[i, :]);
105     sumSlop[i] = sum(slopComponents[i, :]);
106     for j in 1:C loop
107       averageProductMoleFraction[i, j] = productComponents[i, j] /
            sumProduct[i] "Average concentrations inside product tanks";
108       averageSlopMoleFraction[i, j] = slopComponents[i, j] / sumSlop[i]
            "Average concentrations inside slop tanks";
109     end for;
110   end for;
111 //-------------------------------------------------------------------------------
112 /* ============================ Conditions to Open/Close Valves of Slop
      and Products ==========================================*/
113   if XD[1] >= XD1SP then
114     valveProduct[1] = 1.0;
115     valveSlop[1] = 0.0;
116     valveProduct[2] = 0.0;
117     valveSlop[2] = 0.0;
118   elseif D > 0 and XD[1] < XD1SP and XD[2] < XD2SP and product[2] <=
        3e-5 then
119     valveProduct[1] = 0.0;
120     valveSlop[1] = 1.0;
121     valveProduct[2] = 0.0;
122     valveSlop[2] = 0.0;
123   elseif D > 0 and XD[2] >= XD2SP then
124     valveProduct[1] = 0.0;
125     valveSlop[1] = 0.0;
126     valveProduct[2] = 1.0;
127     valveSlop[2] = 0.0;
128   elseif product[2] > 3e-5 and XD[2] < XD2SP and XB[3] < XB3SP then
129     valveProduct[1] = 0.0;
130     valveSlop[1] = 0.0;
131     valveProduct[2] = 0.0;
132     valveSlop[2] = 1.0;
133   else
134     valveProduct[1] = 0.0;
135     valveSlop[1] = 0.0;
136     valveProduct[2] = 0.0;
137     valveSlop[2] = 0.0;
138   end if;
139 /*
      ================================================================================
      */
140 /* ======================  Material Balance on Column
      ==========================================*/
141 //Initially operated at Total reflux
142   when XD[1] >= XD1SP then
143     D = 40/HR;//V / (1 + RR);
144   end when;
145 //-------------------------------------------------------------------------------
```

```
146 /* ========== STILL POT
    ==========*/
147   der(HB) = -D;
148   for j in 1:C loop
149     der(HB * XB[j]) = L * X[1, j] - V * YB[j];
150     YB[j] = alpha[j] * XB[j] / (alpha[1] * XB[1] + alpha[2] * XB[2] +
          alpha[3] * XB[3]);
151   end for;
152 /* ========== N TRAY(Excluding Top and
    Bottom)==========*/
153   for n in 2:N - 1 loop
154     for j in 1:C loop
155       HN * der(X[n, j]) = L * (X[n + 1, j] - X[n, j]) + V * (Y[n - 1, j]
            - Y[n, j]);
156       Y[n, j] = alpha[j] * X[n, j] / (alpha[1] * X[n, 1] + alpha[2] *
            X[n, 2] + alpha[3] * X[n, 3]);
157     end for;
158   end for;
159 /* ========== Bottom Tray
    ==========*/
160   for j in 1:C loop
161     HN * der(X[1, j]) = L * (X[2, j] - X[1, j]) + V * (YB[j] - Y[1, j]);
162     Y[1, j] = alpha[j] * X[1, j] / (alpha[1] * X[1, 1] + alpha[2] * X[1,
          2] + alpha[3] * X[1, 3]);
163   end for;
164 /* ========== Top Tray
    ==========*/
165   for j in 1:C loop
166     HN * der(X[N, j]) = L * (XD[j] - X[N, j]) + V * (Y[N - 1, j] - Y[N,
          j]);
167     Y[N, j] = alpha[j] * X[N, j] / (alpha[1] * X[N, 1] + alpha[2] * X[N,
          2] + alpha[2] * X[N, 3]);
168   end for;
169 /* ========== REFLUX DRUM
    ==========*/
170   for j in 1:C loop
171     HD * der(XD[j]) = V * Y[N, j] - (L + D) * XD[j];
172   end for;
173   L = V - D;
174
175   when XB[3] >= XB3SP then
176     terminate("done");
177   end when;
178
179 end FlatControlTBReal;
```

B.4.6 Python OPC UA client to control the batch distillation column

```
1 # XD[1] 129, XD1SP, XD2SP, D, XD[2] 130, product[2] 133, XB[3] 128, XB3SP
2 # valveProduct[1] 435, valveSlop[1] 437, valveProduct[2] 436,
      valveSlop[2] 438
3
4 from opcua import Client, ua
```

```
5  import time
6  import logging
7  from tclab import clock
8  import pandas as pd
9
10 # Define the URL on which the server is broadcasting
11 url = "opc.tcp://192.168.0.3:4841"
12
13 if __name__ == "__main__":
14     client = Client(url)
15     logging.basicConfig(level=logging.WARN)
16
17     try:
18         client.connect()
19         print("Client Connected")
20
21         enableStopTime = client.get_node(ua.NodeId(10003, 0))
22         # enableStopTime.set_value(False)
23         print("Current state of enableStopTime :
               {}".format(enableStopTime.get_value()))
24
25         run = client.get_node(ua.NodeId(10001, 0))
26         run.set_value(True)
27         print("Current state of run : {}".format(run.get_value()))
28
29         objects = client.get_objects_node()
30
31         XB3_ID, XD1_ID, XD2_ID = 129, 130, 131 # readIDs
32         D_ID = 440 # readIDs
33         product1_ID, product2_ID = 133, 134 # readIDs
34         slop1_ID, slop2_ID = 141, 142 # readIDs
35         HB_ID = 6 # readIDs
36
37         valveProduct1_ID, valveProduct2_ID = 436, 437 # writeIDs
38         valveSlop1_ID, valveSlop2_ID = 438, 439 # writeIDs
39         readTime_ID = 5 # time ID
40
41         modelicaId = {}
42         modelicaId = objects.get_children()
43         # print(modelicaId)
44
45         XD1SP, XD2SP, XB3SP = 0.95, 0.95, 0.95
46
47         tfinal = 10
48         stepsize = tfinal / 500
49         XD_1, XD_2, product_1, product_2, slop_1, slop_2, HB = [], [],
               [], [], [], [], []
50         t = 0
51         while True:
52             # t = t + stepsize
53             print("Local time is {}".format(t))
54             print("Server time is
                   {}".format(modelicaId[readTime_ID].get_value()))
```

```
55
56              XD_1.append(modelicaId[XD1_ID].get_value())
57              XD_2.append(modelicaId[XD2_ID].get_value())
58
59              product_1.append(modelicaId[product1_ID].get_value())
60              product_2.append(modelicaId[product2_ID].get_value())
61
62              slop_1.append(modelicaId[slop1_ID].get_value())
63              slop_2.append(modelicaId[slop2_ID].get_value())
64
65              HB.append(modelicaId[HB_ID].get_value())
66
67              # timeVal.append(t)
68
69              if(modelicaId[XD1_ID].get_value() >= XD1SP):
70                  print("I am in first loop")
71                  modelicaId[valveProduct1_ID].set_value(1.0)
72                  modelicaId[valveSlop1_ID].set_value(0.0)
73                  modelicaId[valveProduct2_ID].set_value(0.0)
74                  modelicaId[valveSlop2_ID].set_value(0.0)
75
76              elif(modelicaId[D_ID].get_value() > 0.0 and
                    modelicaId[XD1_ID].get_value() < XD1SP and \
77                  modelicaId[XD2_ID].get_value() < XD2SP and
                        modelicaId[product2_ID].get_value() <= 3e-5):
78                  print("I am in second loop")
79                  modelicaId[valveProduct1_ID].set_value(0.0)
80                  modelicaId[valveSlop1_ID].set_value(1.0)
81                  modelicaId[valveProduct2_ID].set_value(0.0)
82                  modelicaId[valveSlop2_ID].set_value(0.0)
83
84              elif(modelicaId[D_ID].get_value() > 0.0 and
                    modelicaId[XD2_ID].get_value() >= XD2SP):
85                  print("I am in third loop")
86                  modelicaId[valveProduct1_ID].set_value(0.0)
87                  modelicaId[valveSlop1_ID].set_value(0.0)
88                  modelicaId[valveProduct2_ID].set_value(1.0)
89                  modelicaId[valveSlop2_ID].set_value(0.0)
90
91              elif(modelicaId[product2_ID].get_value() > 3e-5 and
                    modelicaId[XD2_ID].get_value() < XD2SP \
92                  and modelicaId[XB3_ID].get_value() < XB3SP):
93                  print("I am in the 4th loop")
94                  modelicaId[valveProduct1_ID].set_value(0.0)
95                  modelicaId[valveSlop1_ID].set_value(0.0)
96                  modelicaId[valveProduct2_ID].set_value(0.0)
97                  modelicaId[valveSlop2_ID].set_value(1.0)
98
99              else:
100                 print("I am in the last loop")
101                 modelicaId[valveProduct1_ID].set_value(0.0)
102                 modelicaId[valveSlop1_ID].set_value(0.0)
103                 modelicaId[valveProduct2_ID].set_value(0.0)
```

```
                modelicaId[valveSlop2_ID].set_value(0.0)

            time.sleep(1)
            print("="*40)

    except KeyboardInterrupt:
        print("Stopping sequence!")

    finally:
        dict = {'XD[1]': XD_1, 'XD[2]': XD_2, \
            'product[1]': product_1, 'product[2]': product_2, \
            'slop[1]': slop_1, 'slop[2]': slop_2, \
            'HB': HB}
        df = pd.DataFrame(dict)
        df.to_csv('batchDistExt.csv', index = False)
        print("Done!")
        client.disconnect()
```

Appendix C

Shell script for simulating models

```
#!/bin/bash
var1="$1"
var1=${var1%.mo}
if [ "$3" = 1 ]; then
    printf '%s\n' 'loadModel(Modelica)' 'loadFile("'$1'")'
       'simulate('$var1', startTime = 0, stopTime = '$2')' |
       OMShell-terminal
rm $(ls -I "*.mo" -I "*.mat" -I "*.sh")


elif [ "$3" = 2 ]; then
    printf '%s\n' 'loadModel(Modelica)' 'loadFile("'$1'")'
       'simulate('$var1', startTime = 0, stopTime = '$2', simflags =
       "-rt=1.0 -embeddedServer=opc-ua")' | OMShell-terminal
rm $(ls -I "*.mo" -I "*.mat" -I "*.sh")


elif [ "$4" = 1 ]; then
    printf '%s\n' 'loadModel(Modelica)' 'loadFile("'$1'")'
       'simulate('$var1.$2', startTime = 0, stopTime = '$3')' |
       OMShell-terminal
rm $(ls -I "*.mo" -I "*.mat" -I "*.sh")


elif [ "$4" = 2 ]; then
    printf '%s\n' 'loadModel(Modelica)' 'loadFile("'$1'")'
       'simulate('$var1.$2', startTime = 0, stopTime = '$3', simflags =
       "-rt=1.0 -embeddedServer=opc-ua")' | OMShell-terminal
rm $(ls -I "*.mo" -I "*.mat" -I "*.sh")

else
    echo 'Input Command is not Proper'

fi
```

Bibliography

[1] Brenna Sniderman, Monika Mahto, Mark J Cotteleer, et al. Industry 4.0 and manufacturing ecosystems: Exploring the world of connected enterprises. *Deloitte Consulting*, 1:3–14, 2016.

[2] László Monostori. *Cyber-Physical Systems*, pages 1–8. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.

[3] Martin Wollschlaeger, Thilo Sauter, and Juergen Jasperneite. The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0. *IEEE industrial electronics magazine*, 11(1):17–27, 2017.

[4] Kevin Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[5] Wojciech Grega. Hardware-in-the-loop simulation and its application in control education. In *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011*, volume 2, pages 12B6–7. IEEE, 1999.

[6] Priyam Nayak, Pravin Dalve, Rahul Anandi Sai, Rahul Jain, Kannan M Moudgalya, PR Naren, Peter Fritzson, and Daniel Wagner. Chemical process simulation using openmodelica. *Industrial & Engineering Chemistry Research*, 58(26):11164–11174, 2019.

[7] Marco Bonvini, Filippo Donido, and Alberto Leva. Modelica as a design tool for hardware-in-the-loop simulation. In *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009*, number 043, pages 378–385. Linköping University Electronic Press, 2009.

[8] Liu Liu and Georg Frey. Feasibility analysis for networked control systems by simulation in modelica. In *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, pages 729–732. IEEE, 2008.

[9] Carl Sandrock and Philip L de Vaal. Dynamic simulation of chemical engineering systems using openmodelica and cape-open. In *Computer Aided Chemical Engineering*, volume 26, pages 859–864. Elsevier, 2009.

[10] OpenModelica Connection Editor (OMEdit). https://www.openmodelica.org/. Seen on 9 May 2021.

[11] Raspberry Pi Documentation. https://www.raspberrypi.org/documentation/faqs/. Seen on 8 May 2021.

[12] OPC UA overview. https://www.rtautomation.com/technologies/opcua/. Seen on 9 May 2021.

[13] UaExpert - A Full-Featured OPC UA Client. https://www.unified-automation.com/products/development-tools/uaexpert.html. Seen on 10 May 2021.

[14] Python OPC UA package. https://github.com/FreeOpcUa/python-opcua. Seen on 5 May 2021.

[15] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach*. John Wiley & Sons, 2014.

[16] Effect of derivative term. https://controlstation.com/blog/derivative-affect-pid-controller-performance/. Seen on 28 April 2021.

[17] Michael S. Branicky. *Introduction to Hybrid Systems*, pages 91–116. Birkhäuser Boston, Boston, MA, 2005.

[18] tclab PyPI. https://pypi.org/project/tclab/. Seen on 21 May 2021.

[19] S.E. LeBlanc and D.R. Coughanowr. *Process Systems Analysis and Control*. McGraw-Hill chemical engineering series. McGraw-Hill Higher Education, 2009.

[20] William L Luyben. Multicomponent batch distillation. 1. ternary systems with slop recycle. *Industrial & engineering chemistry research*, 27(4):642–647, 1988.

[21] Development of Modelica Library for Batch Distillation. https://www.openmodelica.org/events/openmodelica-workshop/openmodelica-program-2021. Seen on 5 June 2021.

[22] Error while simulating models with 2D arrays in OpenModelica over OPC-UA server. https://github.com/OpenModelica/OpenModelica/issues/7467. Seen on 15 June 2021.

[23] How to access an array from an OPC UA client. https://github.com/FreeOpcUa/python-opcua/discussions/1304. Seen on 5 May 2021.

[24] DWSIM - Open Source Chemical Process Simulator. https://dwsim.inforside.com.br/new/. Seen on 5 May 2021.

[25] MATLAB. https://www.mathworks.com/products/matlab.html. Seen on 1 June 2021.

[26] Scilab. https://www.scilab.org/. Seen on 15 May 2021.

[27] Can MATLAB act as an OPC server? https://in.mathworks.com/matlabcentral/answers/102774-can-matlab-act-as-an-opc-server. Seen on 30 May 2021.

[28] OPC Client Plugin - DWSIM Chemical Process Simulator. https://dwsim.inforside.com.br/wiki/index.php?title=OPC_Client_Plugin. Seen on 30 May 2021.

[29] OPC Client - OLE for Process Control Toolbox for Scilab. https://atoms.scilab.org/toolboxes/opc_client/1.3.1. Seen on 30 May 2021.

[30] OPC UA Components - MATLAB & Simulink. https://www.mathworks.com/help/opc/ug/opc-ua-components.html. Seen on 30 May 2021.

[31] Prosys OPC UA Simulation Server - Prosys OPC. https://www.prosysopc.com/products/opc-ua-simulation-server/. Seen on 28 May 2021.

[32] OPC UA Servers - Unified Automation. https://www.unified-automation.com/downloads/opc-ua-servers.html. Seen on 29 May 2021.

[33] Prosys OPC UA Monitor - Prosys OPC. https://www.prosysopc.com/products/opc-ua-monitor/. Seen on 28 May 2021.

[34] How to create an array in OPC UA Simulation Server - Prosys Forum. https://forum.prosysopc.com/forum/opc-ua-simulation-server/how-to-create-an-array-in/. Seen on 2 June 2021.

[35] How to Use OPC UA for Secure Communications. https://www.automationworld.com/home/blog/13318228/how-to-use-opc-ua-for-secure-communications. Seen on 2 June 2021.