Università degli Studi di Torino
SCUOLA DI SCIENZE DELLA NATURA
Corso di Laurea Triennale in Informatica

Bachelor's Thesis

Design and Development of the Digital Twin of a Greenhouse

SUPERVISOR
Prof. Damiani Ferruccio

CO-SUPERVISORS
Prof. Johnsen Einar Broch
Dr. Schlatte Rudolf
Dr. Kamburjan Eduard

CANDIDATE
Occhipinti Eduard Antonovic
947847

Academic Year 2022/2023

Declaration of Originality

" I declare to be responsible for the content I'm presenting in order to obtain the final degree, not to have plagiarized in all or part of, the work produced by others and having cited original sources in consistent way with current plagiarism regulations and copyright. I am also aware that in case of false declaration, I could incur in law penalties and my admission to final exam could be denied "

ACKNOWLEDGMENTS

I would like to thank my supervisor, prof. Ferruccio Damiani, and the team at the research laboratory in Oslo, in particular the prof. Einar Broch Johnsen, dr. Rudolf Schlatte and dr. Eduard Kamburjan, for grating me the opportunity to work on this project and for the support they provided me during the internship. A special note of appeciation goes to Chinmayi Baramashetru for always being available to help us during the internship to figure out the electronics and Gianluca Barmina and Marco Amato with whom I collaborated closely on this project.

I express my gratitude to all the friends who shared this journey and many long study sessions over these three years with me. I am sure a big part of my achievements is to be attributed to them. In particular Miriam Lamari, that has been and still is always inviting us to study together, Marco Molica that I had great pleasure studying and working with in several assignments, and Dennis Gobbi that always pushed me to do my best and has always been available to explain concepts I had trouble with. I would also like to thank the friends from Oslo that really contributed in making my last semester at UniTO memorable.

Finally I give an heartfelt thanks to my mother for always supporting me throughout the studies.

Abstract

In this thesis we will talk about what digital twins are and how they can be used in a range of scenarios, we will introduce some concepts of the Semantic Web that will serve as a basis for our work. We will also introduce a novel programming language, SMOL, developed to facilitate the way to interface with digital twins. We will talk about the work of myself and my colleagues in the process of building the physical twin with a focus on the structure and the way the responsibilities of the different components are modularized. Finally, we will talk about the software components that we wrote as part of this project, including the code to interact with the sensors and the actuators - with a focus on the Python code and the way it's structured - and the SMOL code that serves as a proof of concept for the automation of the greenhouse.

Keywords

Digital Twins, Raspberry Pi, SMOL, Python

Contents

1 Introduction

The following project was realized as an internship project during my Erasmus semester abroad in Oslo, Norway in collaboration with the University of Oslo and the Sirius Research Center.

The central focus lies in the development of a digital twin (see Section 3) of a greenhouse and the optimization of the environmental conditions for maximum growth potential. Using SMOL, a programming language developed in-house at the research center (see Section 4), we can interface with the simulation to predict optimal watering timing and the correct quantity of water to use.

1.a Learning Outcome

The internship required several theoretical and practical competencies to be acquired, such as:

- Understanding the concept of digital twins.
- Learning the syntax and semantics of SMOL to more easily interface with semantic technologies and test it in a real-world scenario.
- Learning in depth the concept of semantic technologies (in particular RDF, OWL, SPARQL) and their usage in the context of digital twins.
- Learning the basics of the FMI standard.
- Learning basic concepts of electronics for the connection of sensors and actuators to a Raspberry Pi.

The project also required the usage of several tools such as:

- Raspberry Pi and how to work with them on both a software and hardware level.
- InfluxDB for efficient data storage of the sensors' readings.
- Python, Java, and SMOL as programming languages.

By modeling a small-scale model of a digital twin we were able to build the basis for the analysis of a larger-scale problem. Other than the competencies listed above, it was also important to be able to work in a team by dividing tasks and efficiently managing the codebases of the various subprojects (by setting up build tools and CI pipelines in the best way possible, for example). Documentation was also important given that the project is still ongoing and currently being picked up by another student.

Electronic prototyping was also a big part of the project. We had to learn how to connect sensors and actuators to a Raspberry Pi (which required the use of breadboards, analog-to-digital converters, and relays for some of them) and how to translate the signal in a way that could be used as data whenever the output was merely a voltage reading.

1.b My Contribution

As a team we split our roles evenly and divided the project into sub-tasks but worked together on most of the main ones, my involvement was primarily focused on the following areas:

- The hardware side of the project and the setup and configuration of the Linux environment.
- The Python side of the codebase, in particular the code for the communication with the sensors.

1.c Results

A functioning prototype was built with the potential to be easily scaled both in terms of size and complexity. Adding new shelves with new plants is as easy as adding a new component to the model and connecting the sensors and actuators to the Raspberry Pi. The model is easily extendable to add new components and functionalities such as additional actuators for the control of the temperature and humidity or separate pumps dedicated to the dosage of fertilizer. New sensors can also be easily added (a PH sensor would prove useful for certain plants) both in the asset model and the code, each sensor and actuator is represented as a class and they are all being read independently from each other.

1.d Thesis Contents

The thesis from a theoretical point of view focuses on the concept of digital twins in Section 3 and the way the SMOL language (in Section 4) is closely tied with them. In Section 2 we will discuss notions regarding semantic technologies in addition to all the tools and technolgies we used, such as NoSQL databases and different programming languages and paradigms.

In Section 6 we will threat the hardware side of the project and how the sensors interface on the software side. In Section 7 we will take a deep dive into the codebase and have a closer look at the implementation and structure of the various programs.

2 Tools and Technologies

In this chapter, we will explore the tools and technologies used in the project. In particular, in Section 2.a we will introduce InfluxDB and after that, we will touch on the three main programming languages that we had to work with. After that, we'll briefly touch on the concepts of the Semantic Web [1] from Section 2.f onwards, which is an extension of the World Wide Web through standards that aim to make internet data machine-readable. These concepts serve as the underlying foundation for our project.

2.a InfluxDB

InfluxDB [2] is a powerful open-source time-series database that in our context is used to store the data collected by the *data collectors* (see Section 7.d.III). It is designed to handle high volume and high frequency time-stamped data and provides a SQL-like language with specific time-centric functions useful for querying a data structure primarily composed of measurements, streams of data, and points [3]. Being a NoSQL database, it is schemaless and it allows for a flexible data model.

InfluxDB is also directly integrated with the SMOL language, in the `access` expression (see Section 7.c for an example of how we use it in practice), an InfluxDB query with a single answer variable can be directly executed on an external database without the need for external tooling.

The way our data is organized is in structures called *buckets* that are used to store the *measurements*. Each *measurement* is composed of:

- *measurement name*
- *tag set*
  - Used as keys to index the data
- *field set*
  - Used to store the actual data
- *timestamp*

For example, the following query will return the last moisture measurement, given a measurement range of 30 days, from the bucket named *greenhouse*:

```
from(bucket: "greenhouse")
  |> range(start: -30d)
  |> filter(fn: (r) => r["_measurement"] ==
"ast:pot")
  |> filter(fn: (r) => r["_field"] ==
"moisture")
  |> filter(fn: (r) => r["plant_id"] == %1)
  |> keep(columns: ["_value"])
  |> last()
```

Chronograf [4] is a web application included in the InfluxDB 2.0 package, it is the tool that we chose to visualize the data stored in the database and to create dashboards. It can be also used to create alerts and automation rules.

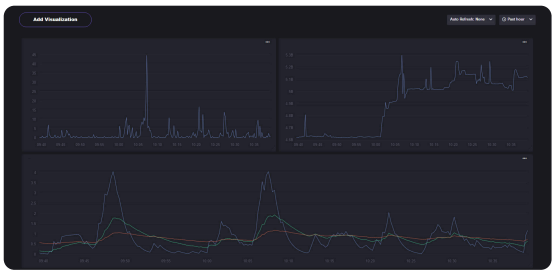


Figure 1: An example of the visualization tools offered by the Chronograf Dashboard [5]

2.b Raspberry Pi

Every part of the system is either managed by or communicates with a Raspberry Pi, in particular, we used a series of Raspberry Pi 4, small single-board computers that are extremely versatile due to their open design, low cost and modularity. The integrated J8 header provides a standard pin pinout for interfacing with sensors and such [6].

2.c Python

When working with the Raspberry Pi 4 the obvious choice for a programming language is Python as it is the most widely used language for the Raspberry Pi due to the amount of libraries available for it and the wide community support. We will talk more in detail in Section 7 about the specifics of the code.

The goal was to make it extremely modular allowing for easy integration of new sensors and actuators, the field of digital twins is modular by nature.

2.d SMOL

SMOL is an Object Oriented programming language in its early development stages, it allows us to:

- Interact with the Influx Database and read data from it, directly without the need for third-party libraries.
- Read and query the asset model (that we discuss in Section 7.b), mapping the data to objects in the heap.
- Map the program state to a knowledge graph through semantic lifting (a concept that will be further discussed in Section 4.b), the program state can be then queried to extract information about the state of the system.
- Represent and run the simulation and interact with FMU simulation files, something that we didn't cover in this project but is the next logical step that should be taken. In Section 3.d we briefly will cover the theory behind FMUs.

SMOL programs are compiled to Java bytecode and can be run natively on the JVM.

2.e Java

We use Java to interface with the SMOL program by making use of the Kotlin (a language designed to fully interoperate with Java) classes that are the building blocks of the REPL (Read-Evel-Print-Loop) interactive interpreter for SMOL [7]. The REPL is used to query the lifted state of the SMOL program.

The Java program is also responsible for sending commands to the Raspberry Pi through SSH in the local network we set up by using the `JSch` library.

2.f RDF

RDF is a standard model for data interchange on the Web. The acronym stands for Resource Description Framework, RDF has features that facilitate data merging even when the underlying schemas differ and it specifically supports the evolution of schemas over time without requiring all the data consumers to be changed [8].

Each RDF statement represents knowledge graphs as a set of triples in the form *<subject> <predicate> <object>* [9].

More specifically the triplets are composed of three basic elements [10]:

- **Resources** - the thing described

- **Properties** - the relationship between things
- **Classes** - the bucket used to group the things

An example related to our knowledge graph is the formalization of the value `50` as ideal moisture for the individual encoded as `basilicum1`:

```
<basilicum1> <hasIdealMoisture> <50>
```

The three-part structure consists of resources identified by a URI (Universal Resource Identifier). RDF extends the linking structure of the Web by using URIs to name the relationship between things, this method allows the mixing of structured and semi-structured data.

These characteristics make RDF a flexible and efficient way to represent knowledge. When thinking about large-scale applications, data can be read quickly due to its structure not being linear like a traditional database and not being hierarchical like XML.

2.F.I Namespaces

Namespaces are used to define a set of short strings to represent long URIs (Uniform Resource Identifiers) that are used in RDF data.

For example instead of:

```
<http://www.w3.org/2002/07/owl#ObjectProperty>
```

we can more concisely write `<owl:ObjectProperty>` and define the namespace `owl` as:

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
```

by defining it at the top of the Turtle document, the aforementioned is a language used to write down RDF graphs in a compact textual form [11].

2.G OWL

The W3C Web Ontology Language (OWL) is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things [12]. OWL is based on RDF and extends it with more vocabulary for the description of properties and classes.

We use OWL in the asset model of the greenhouse to create a formal description of its physical structure and formalize the relationships between the different components.

2.G.I Ontologies

In the context of RDF and OWL, an ontology is a formal description providing users with a shared understanding of a given domain [13]. They define the concepts and entities that exist in a domain and the relationships between them.

2.H SPARQL

After having understood the basics of RDF and OWL, we can now talk about SPARQL, which is the query language for RDF [14]. For example, the following query will return the ideal moisture for each plant:

```
PREFIX ast: <http://www.semanticweb.org/gianl/
ontologies/2023/1/sirius-greenhouse#>

SELECT ?plantId ?idealMoisture
WHERE {
  ?plant rdf:type ast:Plant ;
    ast:hasPlantId ?plantId ;
    ast:hasIdealMoisture ?idealMoisture .
}
```

In the query, we prefix variables with a question mark `?`.

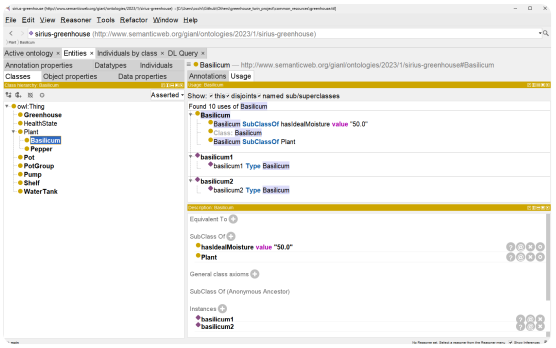
2.1 Protégé

Protégé is a free and open-source ontology editor and framework for building intelligent systems [15].

It is developed by a team from Stanford University and it provides a suite of tools to construct domain models and knowledge-based applications with ontologies.

Protégé fully supports the OWL 2 Web Ontology Language and RDF specifications and that made it the tool of choice for the project,

the following is an image of the interface in which we can see details about the Basilicum class and related instances.



3 Digital Twins

NASA's definition of digital twin

An integrated multiphysics, multiscale, probabilistic simulation of a vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its flying twin. It is ultra-realistic and may consider one or more important and interdependent vehicle systems [16]

A digital twin is a live replica of a Physical System and is connected to it in real-time. An ontology-based asset model is often used to connect the state of the DT (Digital Twin) with the PT (Physical Twin) (see Figure 3). Digital Twins are meant as tools to understand and control assets in nature, industry, or society at large, they are meant to adapt as the underlying assets evolve with time [17]. They are categorized into three distinct definitions depending on their level of integration with each being encapsulated into the next [18]:

Digital Model acts as a digital copy of an existing model or system, changes in the digital copy do not affect the physical system.

Digital Shadow extends the digital copy by automating the exchange of data from the physical system to the digital shadow, but not the opposite.

Digital Twin defines the highest level of integration in which the exchange of data is bidirectional with both systems affecting one another.

NASA was one of the first to introduce the concept of digital twins, however, this research field still lacks any form of standardization, NASA's approach is a monolithic one, what we're trying to do with SMOL provides a more flexible approach and the basis for a standard way to approach this kind of problems.

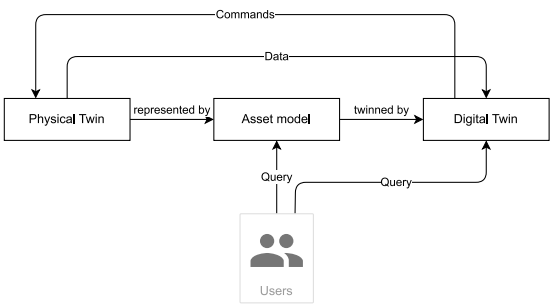


Figure 3: High-level representation of a typical digital twin architecture [19].

3.a Applications

Digital Twins are already extensively employed in a wide range of fields, ranging from power generation equipment (like large engines, and power generation turbines) to establishing timeframes for regularly scheduled maintenance, to the health industry where they can be used to profile patients and help track a variety of health indicators. [20]

For instance M. Wiens, T. Meyer, and P. Thomas describes [18], in the context of introducing the FMI (Section 3.d) standard in light of the need for a full-system simulation model, a digital twin of a hydrogen generator based on wind-turbine energy, a field where a high degree of modularity is observed. In this case, the digital twin allows for operational optimizations and error detection.

3.b Twinning by Construction

Digital twins that mirror a structure that does not change over time, also referred to as static digital twins, are said to be *twinned-by-construction* if the initialization of the digital system ensures the twinning property [19]. When the PT evolves over time, self-adaptation will play a crucial role in ensuring the twinning property.

More specifically the asset model can be used as the base to construct the digital twin, we see our implementation later on in Section 7.b, this kind of DT is what we define as *twinned-by-construction* and the process of establishing a connection between PT and DT via the asset model is what we call *twinning-by-construction*.

3.c Self-Adaptation and MAPE-K

Self-adaptation is the process that is triggered when we observe a deviation of the DT from the PT and/or when the asset model changes. In trying to standardize the self-adaptation process the MAPE-K process was introduced.

MAPE-K loops are feedback control loops that are used in self-adaptive systems to organize their behavior [21], the acronym summarizes the steps needed and stands for:

Monitor the step in which streams of data are collected from the physical and digital systems.

Analyze the step in which we process the data and express expectations on the data stream that is being fed for the digital twin to work correctly, an error might be raised if an unexpected value is registered or if the difference between the collected and simulated data surpasses a certain threshold.

Plan the step in which model search techniques are used to tune the parameters to realign the DT with the PT.

Execute based on Knowledge the step in which the DT is reset with the updated parameters that were computed in the preceding step.

3.d FMI

FMI stands for Functional Mockup Interface, it's a standard created to standardize the model exchange and the co-simulation format in the field of digital twins [18]. The interface encapsulates simulators in FMUs (Functional Mockup Units) which are software components that are used for exchanging and simulating dynamic system models. As formalized by Gomes, Lúcio, and Vangheluwe [22], simulation units are a

tuple $(S, U, Y, \texttt{set}, \texttt{get}, \texttt{doStep})$ with S as the domain of internal states, U the set of inputs, Y the set of outputs, $\texttt{set} : S \times U \times \mathcal{V} \to S$ the function to set the values of the inputs to some values of domain

$\mathcal{V}$, $\texttt{get} : S \times Y \to \mathcal{V}$ the function to get the results and $\texttt{doStep} : S \times \mathbb{R}^+ \to \mathbb{R}$ the function to perform the simulation for a given amount of time [9].

4 SMOL

SMOL (Semantic Micro Object Language) is an imperative, object-oriented language with integrated semantic state access. It can be used as a framework for creating digital twins. The interpreter can be used to examine the state of the system with SPARQL, SHACL, and OWL queries. It can be seen conceptually as a subset of Java.

SMOL uses *Functional Mock-Up Objects* (FMOs) as a programming layer to encapsulate simulators compliant with the FMI standard into object-oriented structures [17]. One of the key features of SMOL is the choice to treat FMUs as first-order concepts and integrate them into its type system.

The project is in its early stages of development, during our internship one of our objectives was to demonstrate the capabilities of the language and help with its development by being some of the first users and submitting issues (and eventually fixes) when they arise on the main repository.

Details on our usage of the language can be found in Section 7.c.

4.a A Functional Mock-Up Object Language

To develop cyber-physical systems, it's imperative to have language and tool support for the core technologies involved, in particular simulators (integrated as FMUs) and knowledge graphs. To achieve this, one needs to leverage a language that encompasses both of those technologies. SMOL, as presented by Kamburjan et al. in 2021, is a syntactically standard, statically typed, object-oriented language that achieves support for both [9].

SMOL goes beyond what would be achievable by extending any traditional language with packages or libraries by incorporating FMUs as Functional Mock-Up Objects in its runtime and treating them as first-order types.

In an FMO the in and out ports of the corresponding FMI become the fields of the object and the functions are converted in class methods, for example, the `doStep` operation Section 3.d is translated to a method that accepts a floating point number as parameter (the time). For instance, to initialize an FMO `gr` in SMOL that accepts a Boolean as input and returns a Double as output, one would write:

```
FMO[in Boolean i, out Double o] gr = simulate(
  "Greenhouse.fmu",
  i = True
);
```

The number of inputs and outputs is not limited and their name has to match the name of the corresponding variable in the FMU. Each one has a `tick` method that accepts a floating point number as a parameter to advance the simulation by a given time.

4.b Semantical Lifting

Semantical Lifting maps a program state to a knowledge graph, to see the program state through the lens of domain knowledge [9].

At any point at runtime, the state of the program can be queried and lifted states can be used to check the consistency of the program with the domain constraints.

5 Overview of the Greenhouse

The specific greenhouse we're working on has the following characteristics:

- It is divided into two shelves
- Each shelf is composed of two groups of plants
- Each group of plants is watered by a single water pump
- Each group of plants is composed of two plants
- Each plant is associated with a pot

5.a Assets

5.a.I Sensors

The following sensors, each associated with a specific asset, are used to monitor the environmental conditions of the greenhouse and the plants:

Greenhouse one webcam used to measure the light level, can be replaced with a light sensor that would also provide an accurate lux measurement.

Shelves one `DHT22` sensor is used to measure the temperature and humidity.

Pots each pot has one capacitive soil moisture sensor used to measure the moisture of the soil.

Plants a `Raspberry Pi Camera Module v2 NoIR` is used to take pictures of the plants and measure their growth by calculating the `NDVI` (see Section 6.d.IV).

5.a.II Actuators

There is one pump for watering but a second bucket is also present as it enables us to add a second pump for liquid fertilizers.

Figure 4: the bottom shelf of our greenhouse, not in the picture the router, host, and lights

6 Raspberry Pi Responsibilities and Physical Setup

In our greenhouse, we use a total of 4 Raspberry Pi 4 but in theory, only one is strictly necessary. The responsibility that each PC has is as follows:

- a *Controller*
- an *Actuator*
- a Router
- a Host

We will refer to the *Controller* also as *Data Collector* because its responsibility is to collect data using sensors attached to the board and send the measurements to the time series database.

In case one wants to replicate our setup, any computer can be used as the host machine, these paragraphs will assume the host runs a Debian-based Linux distribution but any major OS should work with minimal changes.

The *Controller* and *Actuator* can be run on the same Raspberry with minimal changes.

A more in-depth guide to exactly replicate our setup was published at https://github.com/N-essuno/greenhouse_twin_project/tree/main/physical-setup.

To make it easier to configure the Raspberry Pis we wrote some bash scripts that make it easy to install the necessary dependencies with one click. For example to configure the host (see Section 6.b) we can just run the following script that is written to be architecture-independent, given that the host computer does not need to interface with any sensor directly:

```
#!/bin/bash

git clone https://www.github.com/N-essuno/smol_scheduler

influxdbversion="2.7.3"
arch=$(dpkg --print-architecture)

# Checks the architecture and continues the setup only if arch is
# either arm64 or amd64
case "$arch" in
amd64)
  wget https://dl.influxdata.com/influxdb/releases/influxdb2-client-$influxdbversion-linux-amd64.tar.gz
  tar xvzf influxdb2-client-$influxdbversion-linux-amd64.tar.gz
  sudo cp influxdb2-client-$influxdbversion-linux-amd64/influx /usr/local/bin
  ;;
arm64)
  wget https://dl.influxdata.com/influxdb/releases/influxdb2-client-$influxdbversion-linux-arm64.tar.gz
  tar xvzf influxdb2-client-$influxdbversion-linux-arm64.tar.gz
  sudo cp influxdb2-client-$influxdbversion-linux-arm64/influx /usr/local/bin
```

```
  ;;
*)
  echo "Unsupported architecture: $arch"
  exit 1
  ;;
esac

# Sets up influxdb with the standard config tool
influx setup

exit 0
```

6.A OS Choice

We used the, at the time, latest distribution of `Raspberry Pi OS 64bit`. Any compatible operating system will work in practice, but it's necessary to use a 64-bit distribution at least for the host computer. It is also recommended to have a desktop environment on the host computer for simpler data analysis.

We used, and recommend using the Raspberry Pi Imager to mount the OS image on the microSD card.

6.B Host Setup

The only thing that is necessary to install and configure on the host computer is an instance of InfluxDB, after that, it's sufficient to clone the repository containing the SMOL scheduler.

6.C Router Setup

We used a separate Raspberry Pi 4 as a router but that's not strictly necessary, one could simply use an off-the-shelf router for this purpose or give the host the responsibility of being the wireless access point. For our purpose we used `hostapd` and `dnsmasq` but it can also be done via GUI on Raspberry Pi OS very easily, nonetheless in the guide aforementioned we provide a step-by-step tutorial.

6.D Controller Setup

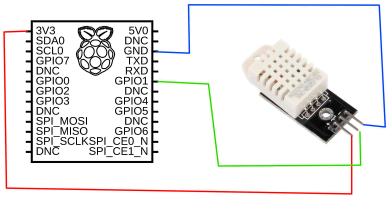
The data points from the sensors that the controller needs to manage are the following:

- Temperature
- Humidity
- Moisture
- Light Level
- NDVI

For the temperature and moisture, we used a `DHT22` sensor, which is very common and that reflects in very good software support.

6.D.I DHT22

The following schematics mirror what we did the connect the sensor to the board



We used the circuitpython libraries, which provide a more modern implementation compared to the standard adafruit libraries.

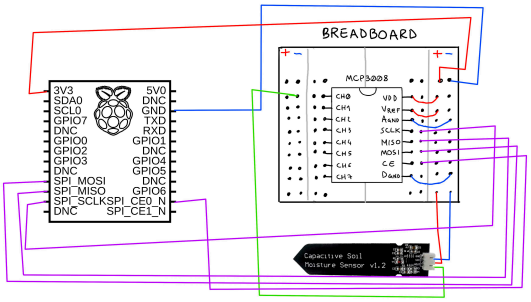
The following is a minimal script that illustrates how we read the data from the sensor.

```
import time
import board
import adafruit_dht

# Initialize the dht device, pass the GPIO pin
# it's connected to as an argument
dhtDevice = adafruit_dht.DHT22(board.D4)

try:
  temperature_c = dhtDevice.temperature
```

```
    humidity = dhtDevice.humidity

    # Print the values to the console
    print(
      "Temp: {:.1f} C    Humidity: {}% ".format(
        temperature_c, humidity
      )
    )

  except RuntimeError as error:
    # Errors happen fairly often
    # DHT's are hard to read
    print(error.args[0])
  except Exception as error:
    dhtDevice.exit()
    raise error

  dhtDevice.exit()
```

6.D.II Moisture

We used a generic capacitive soil moisture sensor, to convert the analog signal we need to use an analog-to-digital converter. For our purpose we used the `MCP3008` ADC which features eight channels, thus making it possible to extend our setup with a decent number of other sensors (for example a PH-meter or a LUX-meter). The following schematics illustrate how we connected the ADC to the board and the moisture sensor to the ADC.



We used `SpiDev` as the library of choice to communicate with the ADC, the following class aids in the readout of the connected sensor:

```
from spidev import SpiDev


class MCP3008:
  """
  Uses the SPI protocol to communicate
  with the Raspberry Pi.
  """

  def __init__(self, bus=0, device=0) -> None:
    self.bus, self.device = bus, device
    self.spi = SpiDev()
    self.open()

    # The default value of 125MHz is not
    # sustainable on a Raspberry Pi
    self.spi.max_speed_hz = 1000000  # 1MHz

  def open(self):
    self.spi.open(self.bus, self.device)
    self.spi.max_speed_hz = 1000000  # 1MHz

  def read(self, channel=0) -> float:
    adc = self.spi.xfer2([
      1, # speed in hertz
      (8 + channel) << 4, # delay in µs
      0 # bits per word
    ])

    data = ((adc[1] & 3) << 8) + adc[2]
    return data / 1023.0 * 3.3

  def close(self):
    self.spi.close()
```

A minimal example of how one would go about reading from an ADC is the following:

```
from MCP3008 import MCP3008

adc = MCP3008()
value = adc.read(channel = 0)
print("Applied voltage: %.2f" % value)
```

6.D.III Light Level

The unavailability of a LUX meter meant we had to get creative and use a webcam to approximate the light level readings. This means that the data is only meaningful when compared to the first measurement. We used the `OpenCV` library to interface with the USB webcam due to it being better supported compared to `picamera2`, another popular library used for similar contexts.

A minimal example of the scripts we used is the following:

```
import cv2

from time import sleep
```

```
cap = cv2.VideoCapture(0)

sleep(2) #lets the webcam adjust its exposure

# Turn off automatic exposure compensation,
# this means that the measurements are only
# significant when compared to the first one,
# to get proper lux reading one should use a
# proper light sensor
cap.set(cv2.CAP_PROP_AUTO_EXPOSURE, 0)

while True:
  ret, frame = cap.read()
  grey = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
  avg_light_level = cv2.mean(grey)[0]
  print(avg_light_level)

  if cv2.waitKey(1) & 0xFF == ord('q'):
    break

  sleep(1)

cap.release()
cv2.destroyAllWindows()
```

6.D.IV NDVI

What is NDVI?

The Landsat Normalized Difference Vegetation Index (NDVI) is used to quantify vegetation greenness and is useful in understanding vegetation density and assessing changes in plant health. NDVI is calculated as a ratio between the red (R) and near-infrared (NIR) values in traditional fashion: [23]

$$\frac{\text{NIR} - R}{\text{NIR} + R}$$

To better understand how healthy our plants were we decided to use an infrared camera to quantify the NDVI. In our small-scale application, the data were not too helpful because of the color noise surrounding the plants but it will work very well if used on a camera mounted on top of the plants and focussed in an area large enough that the image is filled with only vegetation.

Connecting the camera is very straightforward given that we used the `Raspberry Pi NoIR` module.

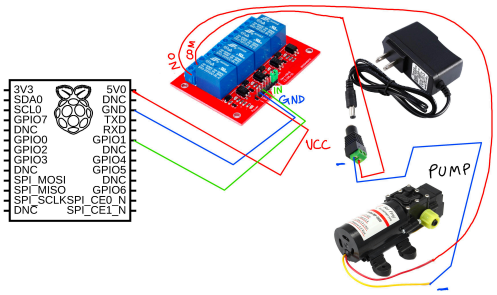
The following is an excerpt of the class we use to calculate the index:

```
import cv2
import numpy as np
from picamera2 import Picamera2


class NDVI:
  def __init__(
    self, format: str = "RGB888"
  ) -> None:
  """Initializes the camera, a config can be
      passed as a dictionary

  Args:
    config (str, optional): refer to
    https://datasheets.raspberrypi.com/camera/
picamera2-manual.pdf
    for all possible values, the default is
    RGB888 which is mapped to an array of
    pixels [B,G,R]"

    transform (libcamera.Transform, optional):
    useful to flip the camera if needed.
    Defaults to Transform(vflip=True).
  """

    self.camera = Picamera2()
    self.camera.still_configuration.main.format
= (format)
    self.camera.configure("still")
    self.camera.start()

  def contrast_stretch(self, image):
    """Increases contrast of the image to
facilitate
       NDVI calculation
    """

    # Find top 5% and bottom 5% of pixel values
    in_min = np.percentile(image, 5)
    in_max = np.percentile(image, 95)

    # Defines min and max brightness values
    out_min = 0
    out_max = 255

    out = image - in_min
    out *= (out_min - out_max) / (
      in_min - in_max
    )
    out += in_min

    return out

  def calculate_ndvi(self, image):
    b, g, r = cv2.split(image)
    bottom = r.astype(float) + b.astype(float)
    bottom[
```

```
      bottom == 0
    ] = 0.01  # Make not to divide by zero!
    ndvi = (r.astype(float) - b) / bottom

    return ndvi

  def read(self):
    return np.mean(
      self.calculate_ndvi(
        self.contrast_stretch(
          self.camera.capture_array()
        )
      )
    )

  def stop(self):
    self.camera.stop()
    self.camera.close()
```

6.e Actuator Setup

In general, to connect actuators (like pumps, light switches, or fans) we can rely on a relay. To connect the relay we can refer to the following schematics:



In our project, we just connected one pump but it's trivial to extend the project to multiple pumps (for example we plan to add one dedicated to pumping fertilized water) or other devices. An example of the code used to interact with the pump is the following:

```
import RPi.GPIO as GPIO
from time import sleep


def pump_water(sec, pump_pin):
  print(
    "Pumping water for {} seconds ..."
    .format(sec)
  )

  # set GPIO mode and set up the pump pin
  GPIO.setmode(GPIO.BCM)
  GPIO.setup(pump_pin, GPIO.OUT)

  try:
    # turn the pump off for 0.25 seconds
    GPIO.output(pump_pin, GPIO.LOW)
    sleep(0.25)

    # turn the pump on for the given time
    GPIO.output(pump_pin, GPIO.HIGH)
    sleep(sec)

    # turn the pump off
    GPIO.cleanup()

    print("Done pumping water.")

  except KeyboardInterrupt:
    # stop pump when ctrl-c is pressed
    GPIO.cleanup()
```

7 Software Components

The following is an overview of the software components it was necessary to write to achieve our goals. The codebase is encompassed in different modules with each being responsible for a distinct task, in particular, we have:

- A scheduler and controller module that is responsible for scheduling the execution of the SMOL program and remotely controlling the actuators that we'll treat in Section 7.a.
- A data collector module that is responsible for the collection of data collected by the different sensors that we'll treat in Section 7.d.
- A module that contains the actuators scripts and is remotely called by the scheduler module, we'll treat it in Section 7.e.
- The asset model used to formally describe the greenhouse and that we'll treat in Section 7.b.
- The SMOL program serves as a proof of concept for the language and will be treated in Section 7.c.

7.a SMOL Scheduler

The SMOL scheduler is the main program that controls the execution of the SMOL code and, in doing so, schedules the actuators. It's also responsible for starting the data collectors. All of these operations are done remotely via the SSH protocol on a local network.

More precisely, the scheduler is a Java program that interfaces with the SMOL Twinning Program and extracts the data from a knowledge graph generated by SMOL via semantical lifting of the program state.

For example, when searching for the plants that need to be watered, the scheduler will query the lifted state and save the results in a `ResulSet` object. The `ResultSet` object contains the IDs of the plants that need to be watered.

```
var repl = new REPL(settings);
var query =
  "PREFIX prog: <https://github.com/Edkamb/Sema
nticObjects/Program#>\n" +
  "SELECT ?plantID " +
  "WHERE { ?plantsToWater prog:Plant_plantId ?
plantID}";

var plantsToWater = repl
  .getInterpreter()
  .query(query);
```

where `Plant` is a class defined in the SMOL program that has a `plantId` property.

At this point, the ResultSet can be passed to a method `waterControl` that will start the actuators responsible for the plants listed.

```
waterControl(plantsToWater);

private static void waterControl(
  ResultSet plantsToWater
) {
  var reader = new GreenhouseModelReader(
    greenhouseAssetModelFile,
```

```
    ModelTypeEnum.ASSET_MODEL
  );

  // list of pump pins to activate
  var pumpPins = new ArrayList<Integer>();

  while (plantsToWater.hasNext()) {
    var plantToWater = plantsToWater.next();
    var plantId = plantToWater
      .get("?plantId")
      .asLiteral()
      .toString();

    pumpPins
      .add(reader.getPumpPinForPlant(plantId));
  }
  startWaterActuator(pumpPins);
}
```

`startWaterActuator` will simply iterate over the pins and via SSH send the command:

```
cd /home/pi/greenhouse_actuator && python3 -m
actuator water <pin>
```

In the SMOL program, the check for the plants that need to be watered is done with a simple while loop that compares the current moisture of the plant by querying the Influx database for the ideal moisture that is associated with the individual in the asset model and deciding based on the difference whether the plant should be watered or not. The next step would be to automate the process by running a simulation (by designing an FMU) of the greenhouse and using the results of the simulation to decide when to water the plants and update the simulation accordingly when the predicted values diverge from the real ones.

7.b Greenhouse Asset Model

The asset model serves as an interface between the physical system - the greenhouse - and the digital system - the digital twin -. It contains an organized description of the physical system, including its components, their properties, and their relationships. To describe the greenhouse we used the `OWL` language, a semantic web language that follows the `RDF` specification.

We can see how the data is organized by comparing the graph of the relationships between the `Basilicum`, the `Plant`, and the `HealthState` classes.

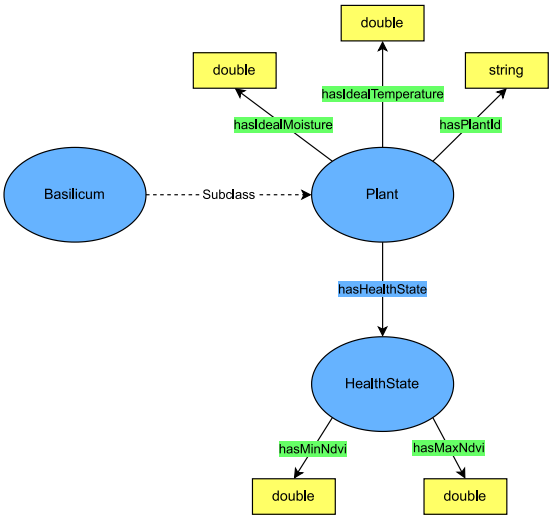


Figure 5: Relationships between the Basilicum, Plant, and HealthState classes

In our asset model, we would formalize the relationships as follows:

```
@prefix : <http://www.semanticweb.org/gianl/
ontologies/2023/1/sirius-greenhouse#> .
@prefix ast: <http://www.semanticweb.org/gianl/
ontologies/2023/1/sirius-greenhouse#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/
namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#
> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-
schema#> .
@base <http://www.semanticweb.org/gianl/
ontologies/2023/1/sirius-greenhouse> .

<http://www.semanticweb.org/gianl/ontologies/
2023/1/sirius-greenhouse> rdf:type
owl:Ontology .

#################################################
#    Object Properties
#################################################

ast:hasHealthState
    rdf:type owl:ObjectProperty ;
    rdfs:subPropertyOf owl:topObjectProperty ;
    rdfs:domain ast:Plant ;
    rdfs:range ast:HealthState .
```

```
###############################################
#    Classes
###############################################

ast:Basilicum rdf:type owl:Class ;
              rdfs:subClassOf ast:Plant .

ast:Plant rdf:type owl:Class ;
          owl:hasKey ( ast:hasPlantId ) .

ast:HealthState rdf:type owl:Class .

###############################################
#    Data Properties
###############################################

ast:hasMaxNdvi rdf:type owl:DatatypeProperty ;
               rdfs:domain ast:HealthState ;
               rdfs:range xsd:double .

ast:hasMinNdvi rdf:type owl:DatatypeProperty ;
               rdfs:domain ast:HealthState ;
               rdfs:range xsd:double .

ast:hasIdealMoisture rdf:type
owl:DatatypeProperty ;
                     rdfs:domain ast:Plant ;
                     rdfs:range xsd:double .

ast:hasIdealTemperature
    rdf:type owl:DatatypeProperty ;
    rdfs:domain ast:Plant ;
    rdfs:range xsd:double .

ast:hasPlantId rdf:type owl:DatatypeProperty ;
               rdfs:domain ast:Plant ;
               rdfs:range xsd:string .
```

We can see that we define the namespaces at the top of the document to make the rest of the definition more readable, the namespaces enable us to use standard definitions, for example when we use the `owl:ObjectProperty` class what we are using is the

```
http://www.w3.org/2002/07/owl#ObjectProperty
```

class that is defined as

```
owl:ObjectProperty a rdfs:Class ;
  rdfs:label "ObjectProperty" ;
  rdfs:comment "The class of object
properties." ;
  rdfs:isDefinedBy <http://www.w3.org/2002/07/
owl#> ;
  rdfs:subClassOf rdf:Property .
```

We have also defined the `ast` namespace that we use to define the classes and properties that are specific to our asset model.

7.c SMOL Twinning program

The `SMOL` program is run by the host computer and is responsible for the creation of the digital twin of the greenhouse.

It achieves this in the following steps:

1. It reads the `asset model` from the `OWL` file
2. It generates `SMOL` objects from the asset model individuals
3. For each asset object it retrieves the sensor data associated with that specific asset from the database (i.e. the moisture of a pot)
4. After retrieving the data it performs the semantic lifting of the program state, creating a knowledge graph that represents the state of the assets in the greenhouse

To interface with the Asset Model we created a SMOL class with methods that retrieve the instances of the classes in the asset model. The class is called `AssetModel` and is defined as follows:

```
/**
* Retrieves data from the asset model and
* converts it to SMOL objects
*/
class AssetModel()
  // get pot instances from the asset model
  List<Pot> getPots()
    List<Pot> pots = construct("
    PREFIX ast: <http://www.semanticweb.org/
gianl/ontologies/2023/1/sirius-greenhouse#>
    SELECT ?shelfFloor ?groupPosition ?
potPosition
    WHERE {
      ?pot rdf:type ast:Pot ;
        ast:hasShelfFloor ?shelfFloor ;
        ast:hasGroupPosition ?groupPosition ;
        ast:hasPotPosition ?potPosition .
    }");
    return pots;
    // edit query to get new pots
  end
```

```
  // get shelf instances from the asset model
  List<Shelf> getShelves()
    List<Shelf> shelves = construct("
      PREFIX ast: <http://www.semanticweb.org/
gianl/ontologies/2023/1/sirius-greenhouse#>
      SELECT ?shelfFloor
      WHERE {
        ?shelf rdf:type ast:Shelf ;
          ast:hasShelfFloor ?shelfFloor .
      }
  ");
    return shelves;
  end

  // get pump instances from the asset model
  List<Pump> getPumps()
    List<Pump> pumps = construct("
      PREFIX ast: <http://www.semanticweb.org/
gianl/ontologies/2023/1/sirius-greenhouse#>
      SELECT ?shelfFloor ?groupPosition
      WHERE {
        ?pump rdf:type ast:Pump ;
          ast:hasShelfFloor ?shelfFloor ;
          ast:hasGroupPosition ?groupPosition.
      }
    ");
    return pumps;
  end

    // get plant instances from the asset model
  List<Plant> getPlants()
    List<Plant> plants = construct("
      PREFIX ast: <http://www.semanticweb.org/
gianl/ontologies/2023/1/sirius-greenhouse#>
      SELECT ?plantId ?idealMoisture
      WHERE {
        ?plant rdf:type ast:Plant ;
          ast:hasPlantId ?plantId ;
          ast:hasIdealMoisture ?idealMoisture .
      }
    ");
    return plants;
  end

  // get health state instances from the asset
model
  List<HealthState> getHealthStates()
    List<HealthState> healthStates =
construct("
      PREFIX ast: <http://www.semanticweb.org/
gianl/ontologies/2023/1/sirius-greenhouse#>
      SELECT ?name ?minNdvi ?maxNdvi
      WHERE {
        ?healthState rdf:type ast:HealthState ;
          ast:hasName ?name ;
          ast:hasMinNdvi ?minNdvi ;
          ast:hasMaxNdvi ?maxNdvi .
      }
    ");
    return healthStates;
  end

  Unit printAssetModelData()
    ...
  end
end
```

Here we see that the syntax of `SMOL` is very intuitive, with the `end` keyword used to delineate code blocks.

We see that each class has a method that retrieves the instances of that class from the asset model. The `construct` top-level expression constructs a list of new objects from a `SPARQL` query [17]. An example of how the classes are defined in `SMOL` is the following:

```
/**
* Represents a physical Plant. Should be
retrieved
* from the asset model via
AssetModel.getPlants()
* Each plant is contained in a Pot. The Pot
contains
* the information about which plant it
contains.
*/
class Plant(String plantId, Double
idealMoisture, String healthState)
  Double getNdvi()
    Double healthState = 0.0;
    List<Double> influxReturn = null;

    influxReturn = access(
      "from(bucket: \"greenhouse_test\")
        |> range(start: -30d)
        |> filter(fn: (r) =>
r[\"_measurement\"] == \"ast:plant\")
        |> filter(fn: (r) => r[\"_field\"] ==
\"ndvi\")
        |> filter(fn: (r) => r[\"plant_id\"] ==
%1)
        |> keep(columns: [\"_value\"])
        |> last()",
    INFLUXDB("config_local.yml"),
    this.plantId);

    healthState = influxReturn.get(0);
    return healthState;
  end

  Double getPotMoisture()
    Double moisture = 0.0;
    List<Double> influxReturn = null;

    influxReturn = access(
      "from(bucket: \"greenhouse_test\")
        |> range(start: -30d)
        |> filter(fn: (r) =>
r[\"_measurement\"] == \"ast:pot\")
        |> filter(fn: (r) => r[\"_field\"] ==
\"moisture\")
        |> filter(fn: (r) => r[\"plant_id\"] ==
%1)
        |> keep(columns: [\"_value\"])
        |> last()",
    INFLUXDB("config_local.yml"),
    this.plantId);
```

```
    moisture = influxReturn.get(0);
    return moisture;
  end
end
```

Here we see a second top-level expression, the `access expression`, also called `query expression`, used to retrieve a list of literals or lifted objects using a query mode `SPARQL` to access the semantically lifted state or `INFLUXDB` to access an external InfluxDB database [17]. In our case, we use the `INFLUXDB` query mode to retrieve the values of the sensors from the database.

What is loaded into the program state is a digital twin of the physical greenhouse, with objects that mirror the physical objects and their properties.

7.d Data Collector Program

To collect data from the sensors connected to the greenhouse we wrote a Python program that retrieves the data and uploads them to InfluxDB.

The repository is located at the address https://github.com/N-essuno/greenhouse-data-collector.

The project directory structure is organized as follows:

```
.
├── .github
│   └── workflows
│       └── black.yml
├── collector
│   ├── __init__.py
│   ├── __main__.py
│   ├── assets
│   │   ├── __init__.py
│   │   ├── asset.py
│   │   ├── greenhouse_asset.py
│   │   ├── measurement_type.py
│   │   ├── plant_asset.py
│   │   ├── pot_asset.py
│   │   ├── pump_asset.py
│   │   ├── shelf_asset.py
│   │   └── utils.py
│   ├── config.ini.example
│   ├── config.py
│   ├── demo
│   │   ├── __init__.py
│   │   └── demo_influx.py
│   ├── influx
│   │   ├── __init__.py
│   │   └── influx_controller.py
│   ├── sensors
│   │   ├── __init__.py
│   │   ├── humidity.py
│   │   ├── interpreter.py
│   │   ├── light_level.py
│   │   ├── mcp3008.py
│   │   ├── moisture.py
│   │   ├── ndvi.py
│   │   ├── temperature.py
│   │   └── water_level.py
│   └── tests
│       ├── __init__.py
│       ├── config_test.ini
│       ├── random_measurements.py
│       ├── test_config_eval.py
│       └── test_influx_controller.py
├── .gitignore
├── .pre-commit-config.yaml
├── README.md
├── requirements.txt
└── scripts
    └── load_random_data.py
```

The project features a `requirements.txt` file that lists all the dependencies needed to run the program.

There is a script `load_random_data.py` that can be used to load random data into the database. It can be used to test the program without having to connect the sensors to the Raspberry Pi.

The `pre-commit-config.yaml` file is used to configure the pre-commit hooks that are run before every commit by leveraging the pre-commit Python framework. The hooks are used to format the code, reorder imports, and statically check for errors.

The `.github` folder contains the CI configuration file that is used to check that the code is formatted correctly and can be extended to run the tests in the future.

In the `test` folder there are some tests that verify the correct functionality of the program.

7.d.I The Collector Module

The program is structured as a Python module. The main module is the `collector` module. It contains the `__main__.py` file which is the entry point of the program. It also contains the `config.py` file that is responsible for reading the configuration file and making it available to the rest of the program. The configuration file is in a `.ini` format, in the main module we include an example file that can be used as a template. The configuration file enables the user to configure the following parameters:

```
[influx2]
url=
org=
token=

[pots]
# moisture_adc_channel refers to the channel in
the Analog to Digital Converter
pot_1={"shelf_floor":"1",
"group_position":"left", "pot_position":"left",
"moisture_adc_channel":1, "plant_id":"1"}
pot_2={"shelf_floor":"1",
"group_position":"left",
"pot_position":"right", "moisture_adc_channel":
2, "plant_id":"2"}

[shelves]
shelf_1={"shelf_floor":"1",
"humidity_gpio_pin":4, "temperature_gpio_pin":
4}

[plants]
plant_1={"plant_id":"1"}
plant_2={"plant_id":"2"}

[sensor_switches]
use_infrared_sensor=false
use_light_sensor=false

[moisture_values]
XP=[2.45, 1.2]

[water_level_values]
XP=[1.1, 2.0]

[light_level_values]
XP=[80, 180]
```

The numbers in the configuration file are what we used for our sensors after calibrating them, `pots`, `shelves`, and `plants` are read as dictionaries that represent the asset model of the greenhouse, and `sensor_switches` is used to enable or disable the readout of some of the sensors, `moisture_values`, `water_level_values` and `light_level_values` are the values used to calibrate the sensors, we'll talk more about this later on in Section 7.d.IV.

The collector module is composed of the following submodules:

- `assets`
- `demo`
- `influx`
- `sensors`
- `tests`

7.d.II Assets

Used to represent the various assets in our database as classes, having used the Python `ABC` module to help us with better class inheritance, we can get an idea of how each of them works just by looking at an excerpt of the `Asset` class.

```
import logging
import sys
import threading
import time
import traceback
from abc import ABC, abstractmethod

from influxdb_client import Point

from collector.influx.influx_controller import
InfluxController


class Asset(ABC):
  stop_flag = threading.Event()
  influx_controller = InfluxController()
  sensor_read_interval = 5

  def set_sensor_read_interval(
    self, sensor_read_interval: int
  ) -> None:
    """
    Sets the sensor read interval in seconds
    """
    self.sensor_read_interval =
sensor_read_interval

  @abstractmethod
  def to_point(self) -> Point:
    """
    Convert the asset to a point. Returns a
    point with the fields and tags of the asset
    """
    pass

  @abstractmethod
  def stop_sensor(self):
    pass
```

```
  def read_sensor_data(self) -> None:
    """
    Read sensor data and write a point to
    influxdb. The point is created by the
    to_point() method. Repeat every
    sensor_read_interval seconds.
    """
    try:
      bucket =
self.influx_controller.get_bucket("greenhouse")

      while not self.stop_flag.is_set():
        point = self.to_point()
        self.influx_controller.write_point(
          point, bucket
        )
        time.sleep(self.sensor_read_interval)
    except Exception as e:
      self.stop_sensor()
      sys.exit(1)

    self.stop_sensor()
    sys.exit(0)

  def stop_thread(self):
    self.stop_flag.set()

  def reset_stop_flag(self):
    self.stop_flag.clear()
```

We can see the way that the `to_point` and `stop_sensor` methods are implemented by looking, as an example, at an excerpt of the `ShelfAsset` class.

```
from dataclasses import dataclass ...


@dataclass
class ShelfAsset(Asset):
  """
  Class representing The Shelf Asset

  Attributes:
    sh_floor (str): floor of the shelf, 1 or 2
    humidity_sensor (Humidity)
    temp_sensor (Temperature)
  """

  sh_floor: str
  hum_sensor: Humidity
  temp_sensor: Temperature

  _typ =
MeasurementType.SHELF.get_measurement_name()

  def __post_init__(self):
    if self.sh_floor ≠ "1" and
       self.sh_floor ≠ "2":
      raise ValueError(
        "sh_floor must be 1 or 2"
      )

  def to_point(self) -> Point:
    return (
      Point(self._typ)
      .tag("shelf_floor", self.sh_floor)
      .field(
        "temperature",
        self.temp_sensor.read()
      )
      .field(
        "humidity",
        self.hum_sensor.read()
      )
    )

  def stop_sensor(self):
    self.temperature_sensor.stop()
```

In general with each asset, we initialize some class variables and verify that the value they are initialized with is valid (using the `__post_init__` method). Here `_typ` turns out to be equivalent to the string `ast:shelf` which is the asset type prefixed by the namespace `ast`, required in InfluxDB. The point is then decorated with the tags and fields that are specific to the asset.

This way of structuring the Asset classes makes it very easy to extend the program to support new assets.

7.D.III Influx

The `influx` module is a wrapper around the `influxdb_client` library. It contains a class `InfluxController` that is responsible for creating the client and the bucket and for writing points to the database.

The class is treated as a singleton so that only one instance of it can be created. This is done by using the `__new__` method, executed at the object's allocation in memory.

```
from typing import Iterable, Optional, Union ...


class InfluxController:
  """
  Singleton, handles the connection to InfluxDB
  Attributes:
    _instance: the singleton instance
```

```
    _client: the InfluxDB client used to
interact with the database
  """

  _instance = None
  _client = InfluxDBClient
            .from_config_file(CONFIG_PATH)

  def __new__(cls):
    """
    Create a new instance of the class if it
    does not exist, otherwise, return the
    existing one
    """
    if cls._instance is None:
      cls._instance = super(
        InfluxController, cls
      ).__new__(cls)

    return cls._instance

  def delete_bucket(
    self,
    bucket_name: str
  ) -> bool:
    ...

  def get_bucket(
    self, bucket_name: str
  ) -> Optional[Bucket]:
    ...

  def create_bucket(
    self,
    bucket_name: str
  ) -> Bucket:
    ...

  def write_point(
    self,
    point: Union[Point, Iterable[Point]],
    bucket: Bucket
  ) -> bool:
    ...

  def close(self):
    self._client.close()
```

7.D.IV Sensors

The `sensors` module contains the classes that represent the sensors connected to the Raspberry Pi. Each sensor class has a method `read` that returns the value measured, in cases such as the moisture sensor, the value needs to be converted to a number that makes sense as a measurement, in this case a percentage. When such a conversion is needed, the class feeds the value to an interpreter as we can see below.

```
from collector.sensors.interpreter import
Interpreter
from collector.sensors.mcp3008 import MCP3008


class Moisture:
  def __init__(
    self, adc: MCP3008, channel: int
  ) -> None:
    """Initializes the Moisture sensor.

    Args:
      adc (MCP3008): the analog to digital
converter
      channel (int): the channel of the ADC to
which the sensor is connected to
    """
    self.interpret = Interpreter("moisture")
                     .interpret

    self.adc = adc
    self.channel = channel

  def read(self) -> float:
    return self.interpret(
      self.adc.read(self.channel)
    )

  def stop(self):
    self.adc.close()
```

The `Interpreter` is a class that utilizes the NumPy `interp` function that performs one-dimensional linear interpolation of monotonically increasing points. Here we can see how the values in the configuration file are used to convert the raw values.

Having the function cached as a class variable enables us to reuse it for every value that needs to be converted, leading to a noticeable performance improvement.

```
import json
import numpy as np

from collector.config import CONFIG_PATH

try:
  # >3.2
  from configparser import ConfigParser
except ImportError:
  # python27
  # Refer to the older SafeConfigParser as
ConfigParser
  from configparser import SafeConfigParser as
ConfigParser
```

```python
class Interpreter:
  """
  Class that interprets raw values from sensors
  to meaningful values. Uses linear inter-
  polation, a variable number of points can
  be used to define the interpolation function
  """

  def __init__(
    self, sensor: str, range: tuple = (0, 100)
  ):
    conf = ConfigParser()
    conf.read(CONFIG_PATH)

    self.XP = json.loads(
      conf[sensor + "_values"]["XP"]
    )
    self.FP = np.linspace(
      range[0], range[1], len(self.XP)
    )

    # If the first value is greater than the
    # last one, reverse the arrays
    if self.XP[0] > self.XP[-1]:
      self.XP = self.XP[::-1]
      self.FP = self.FP[::-1]

  def interpret(self, value: float) -> float:
      return np.interp(value, self.XP, self.FP)
```

Instead of reading the values from the configuration file as standard single values, we read them as a JSON, this enables us to submit multiple values and calculate the interpolation function even for sensors that don't change voltage linearly.

The NumPy `linspace` function is used to create an array of values that are linearly spaced between the first and the last value of the `range` tuple with a length equal to the number of values in the `XP` array.

The arrays `XP` and `FP`, respectively the x-coordinates of the data points and the y-coordinates of the data points, are reversed if the values are not in ascending order, this way it's possible to use the `interp` function even for sensors like the moisture sensor that decreases in voltage as the moisture increases.

7.D.V ___main___

The main class serves to start the data collection, it operates in a multithreaded fashion, starting a thread for each asset. It also handles the stopping of the threads when the program is interrupted.

7.E Actuators Script

As a separate project, we have created a Python module that serves to run the various scripts for the actuators. Actuators can be pumps (for watering or fertilization), can be electronic switches for the lights, can be fans for air circulation, etc...

Being separated into a separate module makes it so that the actuators can be run remotely from the server running the SMOL scheduler. For instance, to activate the pump it's sufficient to call the

```python
GPIO.output(pump_pin, GPIO.HIGH)
```

function from the RPi.GPIO package and let the voltage be on `HIGH` for a set amount of time before resetting it to `LOW`.

8 Conclusions

By the end of the internship, a working model of the greenhouse was delivered, the sensors and the Python program enabled it to continuously monitor the parameters we discussed and save the information on a time series database. After the internship, I learned several concepts related to semantic technologies and the semantic web. It was a very interesting experience working with a novel language, SMOL, that was tailor-made for the specific use case. The project allowed me to affine my skills in Python and to have the opportunity to structure a highly modular project. I had to learn some electronics to interface the Raspberry Pis with the sensors and working with them was a learning experience. We hope our work will help in case the team at the University of Oslo decides to expand on and scale up the project.

References

[1] "Semantic Web". [Online]. Available: https://www.w3.org/standards/semanticweb/

[2] "InfluxDB". [Online]. Available: https://www.influxdata.com/products/influxdb-overview/

[3] "InfluxDB Wikipedia Page". [Online]. Available: https://en.wikipedia.org/wiki/InfluxDB

[4] "Chronograf". Available: https://github.com/influxdata/chronograf

[5] "Chronograf". [Online]. Available: https://connect.adfab.fr/wp-content/uploads/2016/09/chronograf.jpg

[6] "Raspberry Pi Wikipedia Page". [Online]. Available: https://en.wikipedia.org/wiki/Raspberry_Pi

[7] "Semantic Objects". Available: https://github.com/smolang/SemanticObjects/

[8] "Resource Description Framework (RDF)". [Online]. Available: https://www.w3.org/RDF/

[9] Eduard Kamburjan and Einar Broch Johnsen, "Knowledge Structures Over Simulation Units", *2022 Annual Modeling and Simulation Conference (ANNSIM)*, pp. 78–89, 2022.

[10] Rachel Lovinger, "RDF and OWL". [Online]. Available: https://www.slideshare.net/rlovinger/rdf-and-owl

[11] "Turtle Language". [Online]. Available: https://www.w3.org/TR/turtle/

[12] "OWL Web Ontology Language". [Online]. Available: https://www.w3.org/OWL/

[13] "Webdam Project". [Online]. Available: http://webdam.inria.fr/Jorge/html/wdmch8.html

[14] "SPARQL Query Language for RDF". [Online]. Available: https://www.w3.org/TR/rdf-sparql-query/

[15] "Protégé". [Online]. Available: https://protege.stanford.edu/

[16] Edward H. Glaessgen and Doane Stargel, *The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles*. 2012. Available: https://api.semanticscholar.org/CorpusID:110572580

[17] Eduard Kamburjan and Rudolf Schlatte, "The SMOL Language". [Online]. Available: https://smolang.org/

[18] Marcus Wiens, Tobias Meyer, and Philipp Thomas, *The Potential of FMI for the Development of Digital Twins for Large Modular Multi-Domain Systems*. 2021. doi: 10.3384/ecp21181235.

[19] Eduard Kamburjan, Crystal Chang Din, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Einar Broch Johnsen, *Twinning-by-Construction: Ensuring Correctness for Self-adaptive Digital Twins*. 2022.

[20] IBM, "What is a Digital Twin?". [Online]. Available: https://www.ibm.com/topics/what-is-a-digital-twin

[21] Arcaini, Paolo and Riccobene, Elvinia and Scandurra, and Patrizia, *Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation*. 2015. doi: 10.1109/SEAMS.2015.10.

[22] Gomes, Cláudio and Thule, Casper and Broman, David and Larsen, Peter Gorm and Vangheluwe, and Hans, "Co-Simulation: A Survey", vol. 51. Association for Computing Machinery, 2018. doi: 10.1145/3179993.

[23] USGS, "What is the NDVI?". [Online]. Available: https://www.usgs.gov/landsat-missions/landsat-normalized-difference-vegetation-index