

Università degli Studi di Torino

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Triennale in Informatica



Academic Year 2022/2023

Design and Development of the Digital Twin of a Greenhouse

candidate

Eduard Antonovic
Occhipinti
947847

supervisor

Prof. Ferruccio
Damiani

co-supervisors

Prof. Einar Broch
Johnsen

Prof. Rudolf Schlatte

Prof. Eduard
Kamburjan

ACKNOWLEDGMENTS

I would like to express my gratitude to all the friends with whom I shared this journey and many long study sessions during these three years. Special thanks to Chinmayi Bp for always being available to help us during the internship to figure out the electronics and Gianluca Barmina and Marco Amato with whom I worked together on the project

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"

ABSTRACT

We will see how digital twins can be used and applied in a range of scenarios, we'll introduce the language 'SMOL', created specifically for this purpose, and talk about the work of me and my colleagues in the process of designing and implementing a digital twin of a greenhouse.

CONTENTS

Abstract	3
1 Introduction	7
1.a Learning Outcome	7
1.b My Contribution	7
1.c Results	7
1.d Thesis Contents	8
2 Tools and Technologies	10
2.a InfluxDB	10
2.b Python	10
2.c SMOL	10
2.d Java	11
2.e RDF	11
2.e.I Namespaces	11
2.f OWL	11
2.f.I Ontologies	11
2.g SPARQL	11
2.h Protégé	12
3 Digital Twins	14
3.a Applications	14
3.b Twinning by Construction	14
3.c Digital Shadows	14
4 SMOL	16
4.a A Functional Mock-Up Object Language	16
4.b Co-Simulation	16
4.c MAPE-K	16
4.d Semantical Lifting	16
5 Overview of the Greenhouse	18
5.a Assets - Sensors	18
6 Raspberry Responsibilities and Physical Setup	20
6.a OS Choice	21
6.b Host Setup	21
6.c Router Setup	21
6.d Controller Setup	21
6.d.I DHT22	21

6.d.II Moisture	22
6.d.III Light Level	22
6.d.IV NDVI	23
6.e Actuator Setup	23
7 Software Components	26
7.a SMOL Scheduler	26
7.b Greenhouse Asset Model	27
7.c SMOL Twinning program	28
7.d Data Collector Program	29
7.d.I The Collector Module	30
7.d.II Assets	30
7.d.III Influx	31
7.d.IV Sensors	32
7.d.V __main__	33
7.e Actuators Script	33
8 Conclusions	35
References	37

INTRODUCTION

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

The project consists of the creation of a digital twin of a greenhouse and the optimization of the environmental conditions for maximum growth. Using SMOL we can interface with a simulation to predict when to water the plants and how much water to use.

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, a programming language designed by a research group at the University of Oslo to more easily interface with semantic technologies.
- 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 electronics to connect 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 data storage.
- 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.

MY CONTRIBUTION

As a team we split our roles but worked together on most tasks, my involvement was mainly in the following areas:

- 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.

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 actuators for the control of the temperature and humidity or separate pumps dedicated to the dosage of fertilizer. Sensors are also easily added (a PH sensor would prove useful for certain plants) both in terms of the asset model and in terms of code, each sensor/actuator is represented as a class and they are all being read independently from each other.

THESIS CONTENTS

The thesis from a theoretical point of view focuses on the concept of digital twins in Section 3 and the way they are linked with the SMOL language in Section 4. In Section 2 we will discuss notions regarding semantic technologies.

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 deep dive into the codebase and take a look on the implementation and structure of the various programs.

TOOLS AND TECHNOLOGIES

In this chapter, we will explore the tools and technologies used in the project, in particular the concepts of the Semantic Web that serve as the underlying foundation of the project.

INFLUXDB

InfluxDB is a powerful open-source time-series database that is used to store the data collected by the *data collectors*. It is designed to handle high volume and high frequency time-stamped data. Being a NoSQL database, it is schemaless and it allows for a flexible data model.

It serves as the main data storage for the greenhouse.

It's organized in *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 is 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.

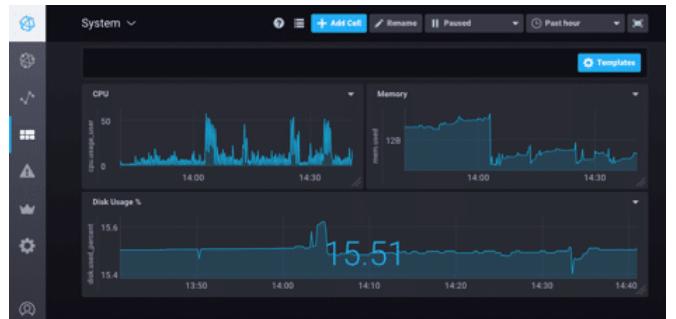


Figure 1: An example of the visualization tools offered by the Chronograf Dashboard

PYTHON

When working with the Raspberry Pi 4 the obvious choice for a programming language is Python, it is the most widely used language for the Raspberry Pi and it has a lot of support and libraries available.

The goal was to make it extremely modular to be able to add new sensors and actuators with ease.

SMOL

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

- Interact with the `InfluxDB` and read data from the database, directly without the need for third-party libraries
- Read and query the knowledge graph, mapping the data to objects in the heap
- Map the program state to a knowledge graph through semantic lifting, the program state can be then queried to extract information about the state of the system

- Represent and run the simulation and interact with **Modelica**

It will be treated in more detail in its dedicated section.

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

JAVA

We use Java to interface with the SMOL program by making use of the Kotlin classes that are the building blocks of the REPL (Read-Evel-Print-Loop) interactive interpreter for SMOL [1]. 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 by using the **JSch** library.

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 [2].

Each RDF statement represents knowledge graphs as a set of triples in the form `<subject> <predicate> <object>` [3].

More specifically the triplets are composed of three basic elements [4]:

- **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 ideal moisture for the Basilicum plant number 1:

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

The three-part structure consists of resources identified by a URI. 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, on a large scale data can be read quickly due to it not being linear like a traditional database and not being hierarchical like XML.

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#> .
```

at the top of the `.ttl` file.

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 [5]. OWL is based on RDF and extends it with more vocabulary for describing 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.

ONTOLOGIES

In the context of RDF and OWL, an ontology is a formal description providing users with a shared understanding of a given domain [6]. They define the concepts and entities that exist in a domain and the relationships between them.

SPARQL

After having understood the basics of RDF and OWL, we can now talk about SPARQL, which is the query language for RDF [7]. 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 .  
}
```

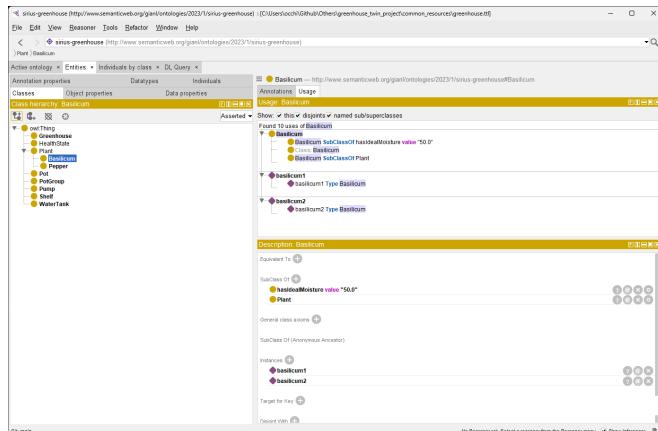
In the query, we prefix variables with a question mark ?.

PROTÉGÉ

Protégé is a free and open-source ontology editor and framework for building intelligent systems [8].

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.



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"

A digital twin is a live replica of a Physical System and is connected to it in real-time, application. Digital Twins are meant to understand and control assets in nature, industry, or society at large, they are meant to adapt as the underlying assets evolve with time. [9]

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 is provide a more flexible approach and the basis for a standard way to approach this kind of problems.

APPLICATIONS

Digital Twins are already extensively used in a wide range of fields, ranging from power generation equipment - like large engines, and power generation turbines - to establish timeframes for regularly scheduled maintenance, to the health industry where they can be used to profile patients and help tracking a variety of health indicators. [10]

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 [11].

DIGITAL SHADOWS

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.

SMOL uses *Functional Mock-Up Objects* (FMOs) as a programming layer to encapsulate simulators compliant with the [FMI standard](#) into object-oriented structures [9].

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 the first users.

Details on the implementation can be found in the [SMOL Twinning Program](#) section.

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 [3]. SMOL goes beyond what would be achievable by extending any traditional language by incorporating FMUs as Functional Mock-Up Objects in its runtime.

CO-SIMULATION

MAPE-K

SEMANTICAL LIFTING

Semantical Lifting maps a program state to a knowledge graph, to see the program state through the lens of domain knowledge [3].

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

ASSETS - SENSORS

The following sensors are used to monitor the environmental conditions of the greenhouse and the plants:

Greenhouse

- 1 `webcam` used to measure the light level, can be replaced with a light sensor that would also provide an accurate lux measurement

Shelves

- 1 `DHT22` sensor used to measure the temperature and humidity

Pots

- 1 capacitive soil moisture sensor used to measure the moisture of the soil

Plants

- 1 `Raspberry Pi Camera Module v2 NoIR` used to take pictures of the plants and measure their growth by calculating the `NDVI`



Figure 3: the bottom shelf of our greenhouse, not in the picture the router, host, and lights

RASPBERRY 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, one can use whichever computer they want as the host, these paragraphs will assume the host computer runs a Debian-based Linux distribution but any 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 [here](#).

To make it easier to configure the Raspberry we wrote some bash scripts that make it easy to install the necessary dependencies with one click. For example to configure the Host 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
```

```

;;
esac

# Sets up influxdb with the standard config tool
influx setup

exit 0

```

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.

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](#).

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.

CONTROLLER SETUP

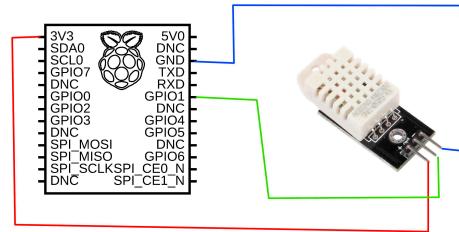
The data points from the sensors that it needs to manage are the following:

- [Temperature](#)
- [Humidity](#)
- [Moisture](#)
- [Light Level](#)
- [NDVI](#)

For the temperature and moisture, we used a [DHT22](#) sensor, which is very common and results in very good software support.

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

```

```

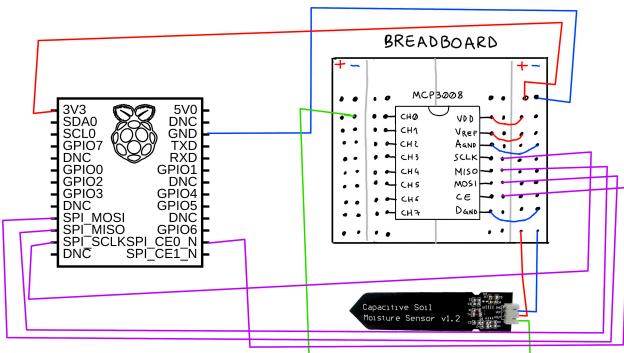
# DHT's are hard to read
print(error.args[0])
except Exception as error:
    dhtDevice.exit()
    raise error

dhtDevice.exit()

```

MOISTURE

We used a generic moisture 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 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 microseconds
        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)

```

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 library OpenCV to interface with the USB webcam because it is better supported compared to picamera2.

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()

```

NDVI

What is it?

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: [12]

$$\frac{\text{NIR} - R}{\text{NIR} + R} \quad (1)$$

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 extract of the class we use to calculate the index:

```

import cv2
import numpy as np
from picamera2 import Picamera2

class NDVI:
    def __init__(self, config: 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]. Defaults to "RGB888".

```

```

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

    # Defines minimum and maximum 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 sure we don't divide by zero!
    ndvi = (r.astype(float) - b) / bottom

    return ndvi

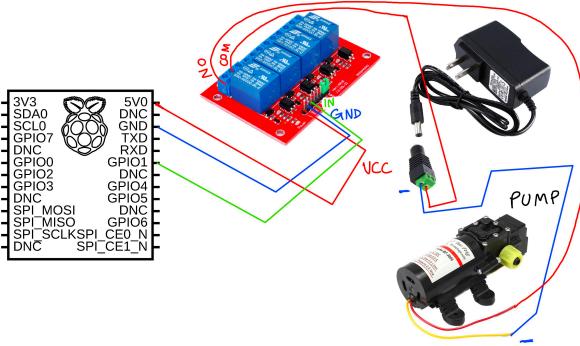
def read(self):
    return np.mean(
        self.calculate_ndvi(
            self.contrast_stretch(
                self.camera.capture_array()
            )
        )
    )

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

```

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()

```


SOFTWARE COMPONENTS

The following is an overview of the software components it was necessary to write to achieve our goals.

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 that is done remotely via SSH 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 ResultSet 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/
  SemanticObjects/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 with the ideal moisture that is associated with the individual in the asset model. 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.

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.

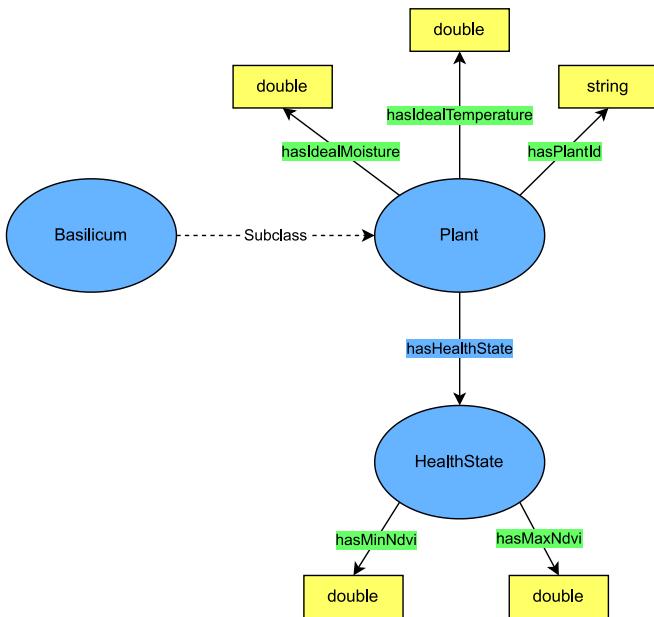


Figure 4: Relationships between the `Basilicum`, `Plant`, and `HealthState` classes

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 the 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.

SMOL TWINNING PROGRAM

The `SMOL` program is run by the host computer and is responsible for creating 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 convert 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 [9]. 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 getNdv()
        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 [9]. In our case, we use the `INFLUXDB` query mode to retrieve the values of the sensors from the database.

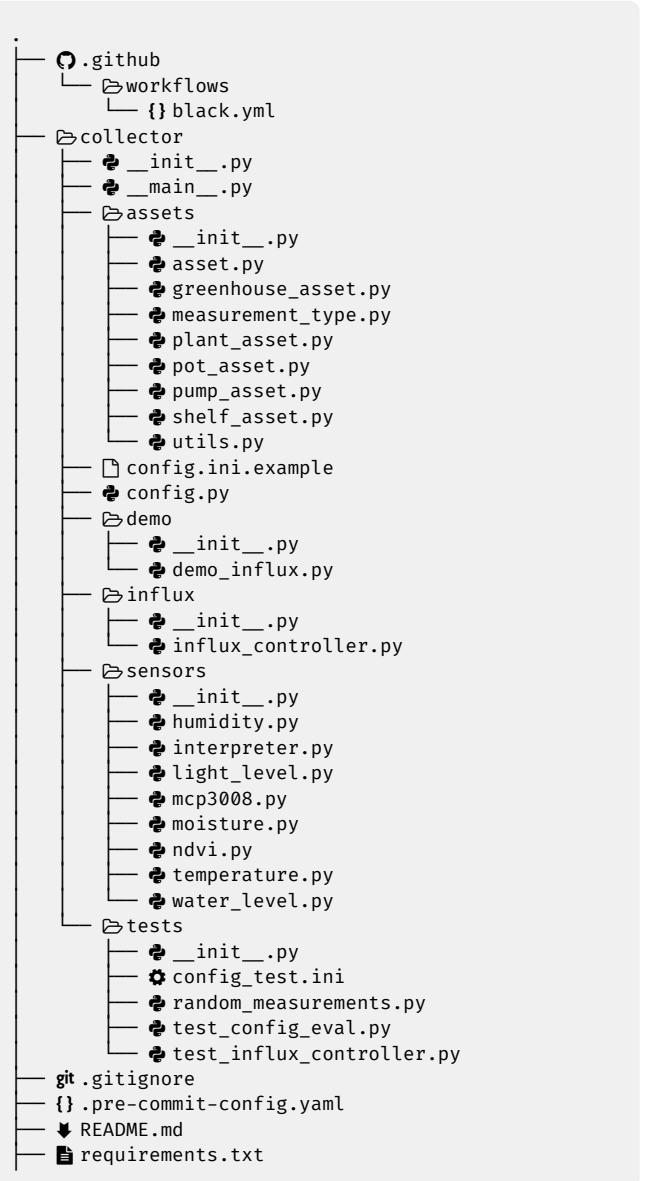
What is loaded into the program state is a digital twin of the physical greenhouse, with objects that mirror the physical objects and their properties.

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 can be found at <https://github.com/N-essuno/greenhouse-data-collector>.

The project is organized as follows:



```
└─ scripts
    └─ load_random_data.py
```

The project features a `requirements.txt` file that contains 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. The hooks are used to format the code, reorder imports, and 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 functionality of the program.

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 a `.ini`, 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 dictionary that represents 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.

The collector module is composed of the following submodules:

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

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.

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.

```

from typing import Iterable, Optional, Union ...


class InfluxController:
    """
    Singleton that 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()

```

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
        """
        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, for 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:
    # python2
    # Refer to the older SafeConfigParser as
    ConfigParser
        from configparser import SafeConfigParser as
    ConfigParser

class Interpreter:
    """
    Class that interprets raw values from sensors to

```

```

meaningful values. Uses linear interpolation,
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.

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.

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

fertilizing), can be electronic switches for the lights, can be fans for air circulation ecc...

Being separated into a separate module makes it so that the actuators can be run remotely from the server running the SMOL scheduler.

CONCLUSIONS

REFERENCES

- [1] *Semantic Objects*. [Online]. Available: <https://github.com/smolang/SemanticObjects/>
- [2] “Resource description framework (RDF).” <https://www.w3.org/RDF/>
- [3] Eduard Kamburjan, and Einar Broch Johnsen, “Knowledge structures over simulation units,” *2022 Annu. Model. Simul. Conf. (ANNSIM)*, pp. 78–89, 2022.
- [4] Rachel Lovinger, “RDF and OWL.” <https://www.slideshare.net/rlovinger/rdf-and-owl>
- [5] “OWL web ontology language.” <https://www.w3.org/OWL/>
- [6] “Webdam project.” <http://webdam.inria.fr/Jorge/html/wdmch8.html>
- [7] “SPARQL query language for RDF.” <https://www.w3.org/TR/rdf-sparql-query/>
- [8] “Protégé.” <https://protege.stanford.edu/>
- [9] Eduard Kamburjan, and Rudolf Schlatte, “The SMOL language.” <https://smolang.org/>
- [10] IBM, “What is a digital twin?.” <https://www.ibm.com/topics/what-is-a-digital-twin>
- [11] 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.
- [12] USGS, “What is the NDVI?.” <https://www.usgs.gov/landsat-missions/landsat-normalized-difference-vegetation-index>