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

Aknowlegements

Special thanks to Chinmayi Bp for always being available to help us figure out the electronics and Gianluca Barmina and Marco Amato with whom I worked together on the project

Declaration of Originality

"Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata."

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

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 in the creation of a digital twin of a greenhouse and the optimization of the environmental conditions for maximum growth. Using SMOL we can predict when to water the plants and how much water to use.

Learning Outcome

- Understanding the concept of digital twins
- Learning the syntax and semantics of SMOL
- Learning the basics of the FMI standard
- Learning basic electronics to connect sensors and actuators to a Raspberry Pi

By modelling 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 competences listed above, it was also important being able to work in 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).

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.

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 exentable to add new components and functionalities such as actuators for the control of the temperature and the 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 indipendently from each other.

Tools and Technologies

The following is a general oveview of the tools and technologies used in 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 of name `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

Developing a library to interface with the sensors

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.

OWL

OWL is a knowledge representation language that is used to describe the `asset model` of the greenhouse. It is used to create a formal description of the greenhouse's physical structure and the relationships between the different components.

The following code example models a class 'Basilicum' and links information about the ideal moisture in the asset model:

```
###  http://www.semanticweb.org/gianl/ontologies/
2023/1/sirius-greenhouse#Basilicum
ast:Basilicum rdf:type owl:Class ;
  rdfs:subClassOf ast:Plant ,
    [ rdf:type owl:Restriction ;
        owl:onProperty ast:hasIdealMoisture ;
          owl:hasValue "70.0"
    ] .
```

SMOL Language

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

- Interact with the `InfluxDB` and read data from the database, directly without the need of a 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 by means of semantic lifting, the program state can be then queried to extract information about the state of the system
- Represent and run simulation and interact with `modelica`

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

Digital Twins

NASA's definition of digital twin
"An integrated multiphysics, multiscale, probabilistic simulation of a veichle 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 interdependant veichle systems"

A digital twin is a live replica of a Pysical 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. [1]

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 these kind of problems.

Applications

Digital Twins are already exensively used in a wide range of fields, ranging from power generation equipment - like large engines, power generation turbines - to establish timeframes for regularly scheduled maintenance, to the health industry where they can be used profile patients and help tracking a variety of health indicators. [2]

Twinning by Construction

Digital twins that mirror a structure that does not change over time, also reffered to as static digital twins, are said to be *twinned-by-construction* if the initialisation of the digital system ensures the twinning property [3].

Digital Shadows

Overview of the Greenhouse

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

- It is divided in two shelves
- Each shelf is composed by two groups of plants
- Each group of plants is watered by a single water pump
- Each group of plants is composed by 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`

Raspberry Responsabilities 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 responsability is to collect data using sensors attached to the board and send the measurements to the time series database.

In case one wants to repllicate our setup, one can use whichever computer they want as the host, these pharagraphs 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

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 a simpler data analysis.

We used and recommend using the Raspberry Pi Imager to mount the OS image on the micro-sd card.

Host Setup

The only thing that is necessary to install and configure on the host computer is an instance of InfluxDB. One also needs to clone the repository containing the SMOL scheduler.

Router Setup

We used a separate Raspberry Pi 4 as a router but that's not striclty necessary, one could simply use an off-the-shelf router for this purpose or give to the host also the responsability 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

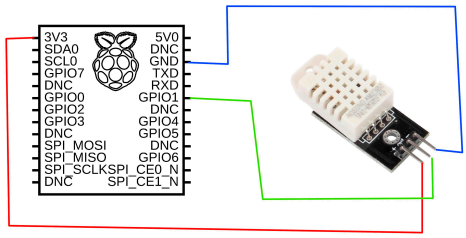
The datapoints from the sensors that it needs to manage are the following:

- Temperature
- Humidity
- Moisture
- Light Level
- NDVI

For the temperature and moisture we used a `DHT22` sensor, being very common results in a very good software support.

DHT22

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



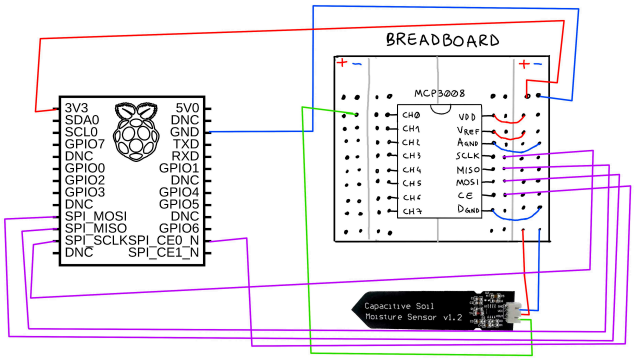
We used the circuitpython libraries, which provide a more modern implementation compared to the standard adafruit libraries.

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

```python
import time
import board
import adafruit_dht

# Initialize the dht device
# Pass the GPIO pin it's connected to as arugument
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 analogue 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 exend 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:

```python
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 micro seconds
      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:

```python
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 beacuse it is better supported compared to picamera2.

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

```
import cv2

from time import sleep

cap = cv2.VideoCapture(0)

sleep(2) #lets 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?

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: [4]

$$\frac{\text{NIR} - R}{\text{NIR} + R} \tag{1}$$

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 straight forward given that we used the `Raspberry Pi NoIR` module.

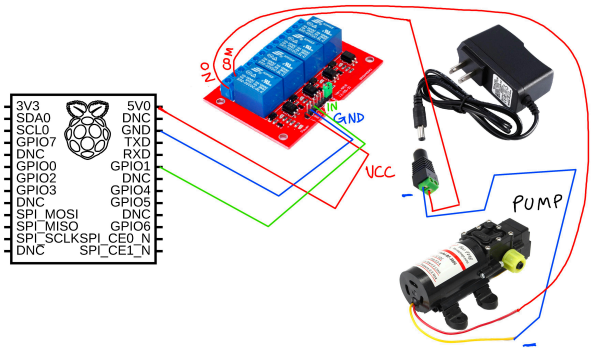
The following is an extract of the class we use to calculate the index:

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


class NDVI:
  def __init__(
    self, 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].
      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 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 relay on a relay. To connect the relay we can refer to the following schematics:



In our project we just conneted 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()
```

Putting it all Together

Figure 2: the bottom shelf of our greenhouse, not in the picture the router, host and lights

SMOL

SMOL (Semantic Micro Object Language) is an imperative, object-oriented language with integrated semantic state access. It can be used served 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 [1].

The project is in its early stages of developement, during our internship one of our objectives was to demonstrate the capabilities of the language and help with its developement 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 [5]. SMOL goes beyond what would be achievable by extending any tradional language by incorporating FMUs as Functional Mock-Up Objects in its runtime.

Co-Simulation

Semantical Lifting

Semantical Lifting maps a program state to a knowledge graph, to see the program state through the lense of domain knowledge [5].

Semantic Lifting

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 in 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 runs a SMOL program, extracts the data from a knoledge graph generated by SMOL and interacts with other layers to create the digital twin.

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:

```
.
├── .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 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 also 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 that 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 actually used for our sensors after calibrating them, `pots`, `shelves` and `plants` are dictionary that represent the asset model of the greenhouse, `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 on how each of them work 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()
```

Where we can see the way that the `to_point` and `stop_sensor` methods are implemented by looking, as an example, to 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 are 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 in 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 noticable 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


class Interpreter:
  """
  Class that interprets raw values from sensors to
  meaningful values. Uses a 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 reding 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 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 seaprated in a separate module makes it so that the actuators can be run remotely from the server running the SMOL scheduler.

Greenhouse Asset Model

The asset model is a representation of the greenhouse as a knowledge graph.

Working with the Ontology

To work with the ontology we created a `SMOL` class that reflects the structure of the asset model.

```
/**
* 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 [1]. 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 literarls or lifted objects using a query mode `SPARQL` to access the semantically lifted state or `INFLUXDB` to access an external InfluxDB database [1]. In our case we use the `INFLUXDB` query mode to retrieve the values of the sensors from the database.

SMOL Twinning program

The `SMOL` program is run periodically by the server and is responsible for creating the digital twin and running the FMI simulators.

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
4. After retrieving the data it performs the semantic lifting of the program state, creating a knowledge grappph that represents the state of the assets in the greenhouse

The Digital Twin

Conclusions

References

[1] Eduard Kamburjan, and Rudolf Schlatte, "The SMOL language." https://smolang.org/

[2] IBM, "What is a digital twin?." https://www.ibm.com/topics/what-is-a-digital-twin

[3] 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.

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

[5] Eduard Kamburjan, and Einar Broch Johnsen, "Knowledge structures over simulation units," *2022 Annu. Model. Simul. Conf. (ANNSIM)*, pp. 78–89, 2022.