**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**

**Rudolf Schlatte**

**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 collegues 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]

# 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.

```
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 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 read-out 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 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:

```
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: [3]

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

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()
```

# 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.

## Co-Simulation

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.

# 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 repo can be found at https://github.com/N-essuno/greenhouse-data-collector. What follows is an overview of the program

## Greenhouse Asset Model

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

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