FINITE STATE MACHINE

Max Visscher 1812998

Hogeschool Utrecht
HBO-ICT AI

Table of content

INHOUD

INTRODUCTION

In this document I will write about how the process of designing and implementing a final state machine(FSM). My implementation is based on how I understood the theory behind FSM.

The main point of this exercise is to build a FMS with the use of OOP. We were allowed to pick our own implementation of the FSM. For this exercise I choose to implement it with the functions of a coffee machine.

My main source of the knowledge is the lectures and the canvas page. As I find consuming knowledge the easiest through discussion of the theory, most of my theory will be based on the discussion we had during the lectures.

The document will have three main parts, one will be focused on the design and development of the FSM, and one part will be focused on the implementation. The last part of the document will focus on reflection and discussion.

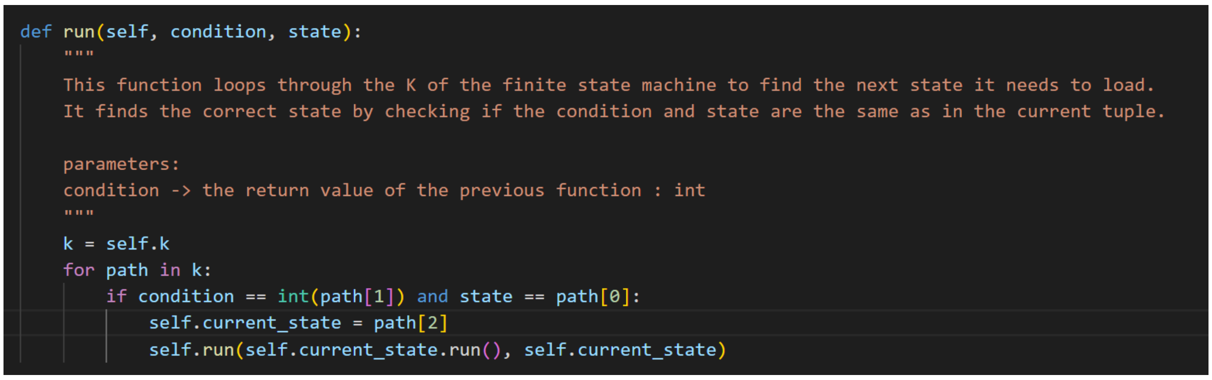
LIBRARY

The library point of the library is to be able to run a program based on a set of possible paths. To do this we create states. You could see these states as a note in the diagram. I will elaborate on this in the chapter of implementation.

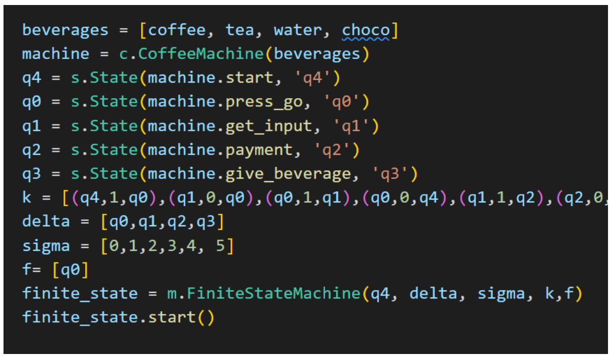
Each state is given an function of the program. After this function is run, the state and the condition are returned. Based on this information the program decides on which state to move to.

To finish of the library I added a few functions to add information to each category of the FSM.

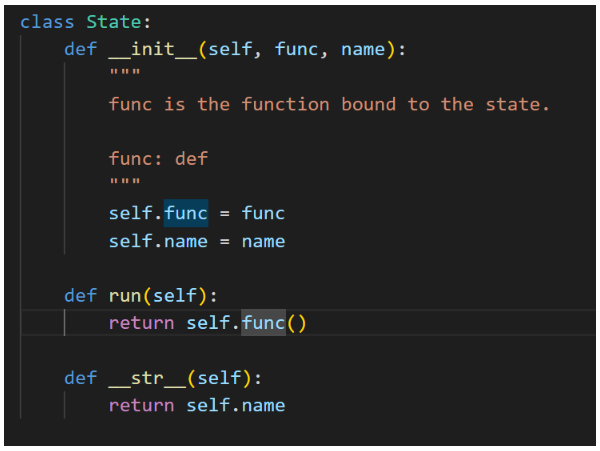
I decided to give the FSM a parameter in which you can give it all possible paths. This is a list of tuples. Each tuple consists of a state, condition and another state. The first state is the state the FSM is currently on. The condition is an integer that allows the program select the next path. The second State is the next state. The concept of understanding the FSM was harder. The code ended up being a single for loop with an if condition.



```python
def run(self, condition, state):
    """
    This function loops through the K of the finite state machine to find the next state it needs to load.
    It finds the correct state by checking if the condition and state are the same as in the current tuple.

    parameters:
    condition -> the return value of the previous function : int
    """
    k = self.k
    for path in k:
        if condition == int(path[1]) and state == path[0]:
            self.current_state = path[2]
            self.run(self.current_state.run(), self.current_state)
```



```python
beverages = [coffee, tea, water, choco]
machine = c.CoffeeMachine(beverages)
q4 = s.State(machine.start, 'q4')
q0 = s.State(machine.press_go, 'q0')
q1 = s.State(machine.get_input, 'q1')
q2 = s.State(machine.payment, 'q2')
q3 = s.State(machine.give_beverage, 'q3')
k = [(q4,1,q0),(q1,0,q0),(q0,1,q1),(q0,0,q4),(q1,1,q2),(q2,0,
delta = [q0,q1,q2,q3]
sigma = [0,1,2,3,4, 5]
f= [q0]
finite_state = m.FiniteStateMachine(q4, delta, sigma, k,f)
finite_state.start()
```



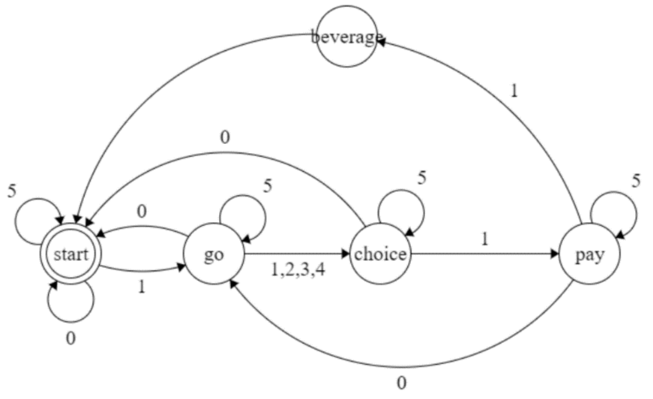
```python
class State:
    def __init__(self, func, name):
        """
        func is the function bound to the state.

        func: def
        """
        self.func = func
        self.name = name

    def run(self):
        return self.func()

    def __str__(self):
        return self.name
```

In the for loop I check all possible paths. For each path I check if the current node and condition are the same. If this is the case I go to the selected next node.

It was difficult to find a way to write the FSM so the implementation wasn't to hard. I came up with giving each state a parameter which requires a function as input.

IMPLEMENTATION

For implementation I choose to create a coffee machine. To do this I had to recreate al the functionalities of this machine. Because a coffee machine only had buttons, so I designed the machine while keeping in mind that misinputs are impossible. I account for pressing buttons which are non functional during a state. But inserting a string in the console will throw an error.



To show the flow the program I created this diagram. The start note is the start state of the program. If the user gives a 1 as input the program will go to its second stage. Here the user can pick between 4 beverages. If the selected beverage is out of stock the user will return to the start state.

Each beverage is a dictionary which keeps track of the stock. By updating this I am able to keep track of this. To make this a true finite machine I need to add a note for each of these steps. But I found this to be overcomplicated for the given problem.

The functions of the coffee machine is to return either a 0,1 and 5, with the exception of choice, which is 0,1,2,3,4,5.

0 means failure, 1 through 4 in the given scenario means success. And 5 means invalid input.

DISCUSSION

I think this is a great implementation to find out the boundaries of FSM. It allowed me to think about the possibilities of FSM, and how complicated and in depth they can be. As I mentioned before there were possibilities to increase the states of the program. Theoretically each action, such as removing one item from the stock or selecting a specific beverage should be a singe note. But I found this too complicated.

It was fun trying multiple things and experimenting with different ways of implementing FSM. The hardest part for me was finding the sweet spot on combining the coffee machine with the final state machine.

This is because the final state machine is less complicated as I thought. In the end the FSM only took a few rows. It was also challenging to write a coffee machine which is able to communicate with a FSM, since I am unable to return the values I am used to return. By returning only a 1 or 0, based on if the function succeeded or not allowed me to learn more about OOP.

It was fun to wrap my head about the content and find the correct way of thinking.