COMP712: Classical Artificial Intelligence

Workshop: Finite State Machine (FSM)

Dr Daniel Zhang @ Falmouth University
2023-2024 Study Block 1

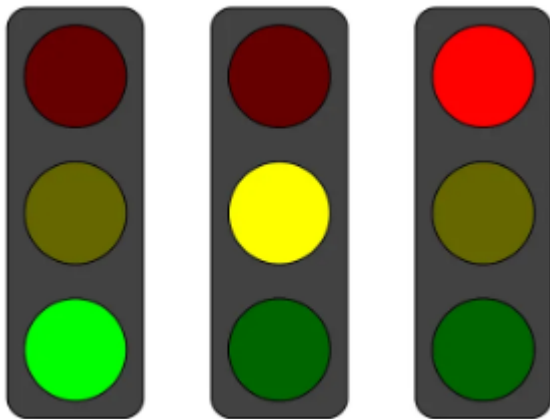Table of Contents

Introduction

Top

Finite State Machines (FSMs) are powerful tools in computer science and programming. They provide a structured way to model complex systems, making them an essential concept for any aspiring programmer or computer scientist. In this session, we will delve into the world of Finite State Machines, exploring their fundamental principles and applications. We'll also demonstrate how to implement FSMs using Python, a versatile and popular programming language.

Understanding Finite State Machines

Top

At its core, a Finite State Machine is a mathematical model that describes a system's behaviour by dividing it into a finite number of states, transitions between these states, and the conditions triggering these transitions. FSMs are excellent for representing systems with distinct modes or conditions. They are widely used in various domains, such as robotics, gaming, natural language processing, and more.

Components of a Finite State Machine

Top

Before we delve into Python programming, let's understand the essential components of an FSM:

- `States` : These represent the distinct conditions or modes of the system. For instance, in a traffic light system, the states could be 'Green,' 'Yellow,' and 'Red.'
- `Transitions` : These define the conditions under which the system switches from one state to another. Transitions are often triggered by specific `Events` (*It is also considered as `Conditions` or `Signals`*) or criteria.
- `Actions` : Actions are tasks or operations performed when a transition occurs. They can range from simple outputs to complex calculations.

- **Initial State** : This is the starting point of the FSM, where the system begins.
- **Final State** : The final state indicates the end of a process or scenario.

Note: In certain FSMs, `transitions` and `actions` may be very similar, as no additional actions are required beyond state transitions.

A Practical Example: Traffic Lights FSM
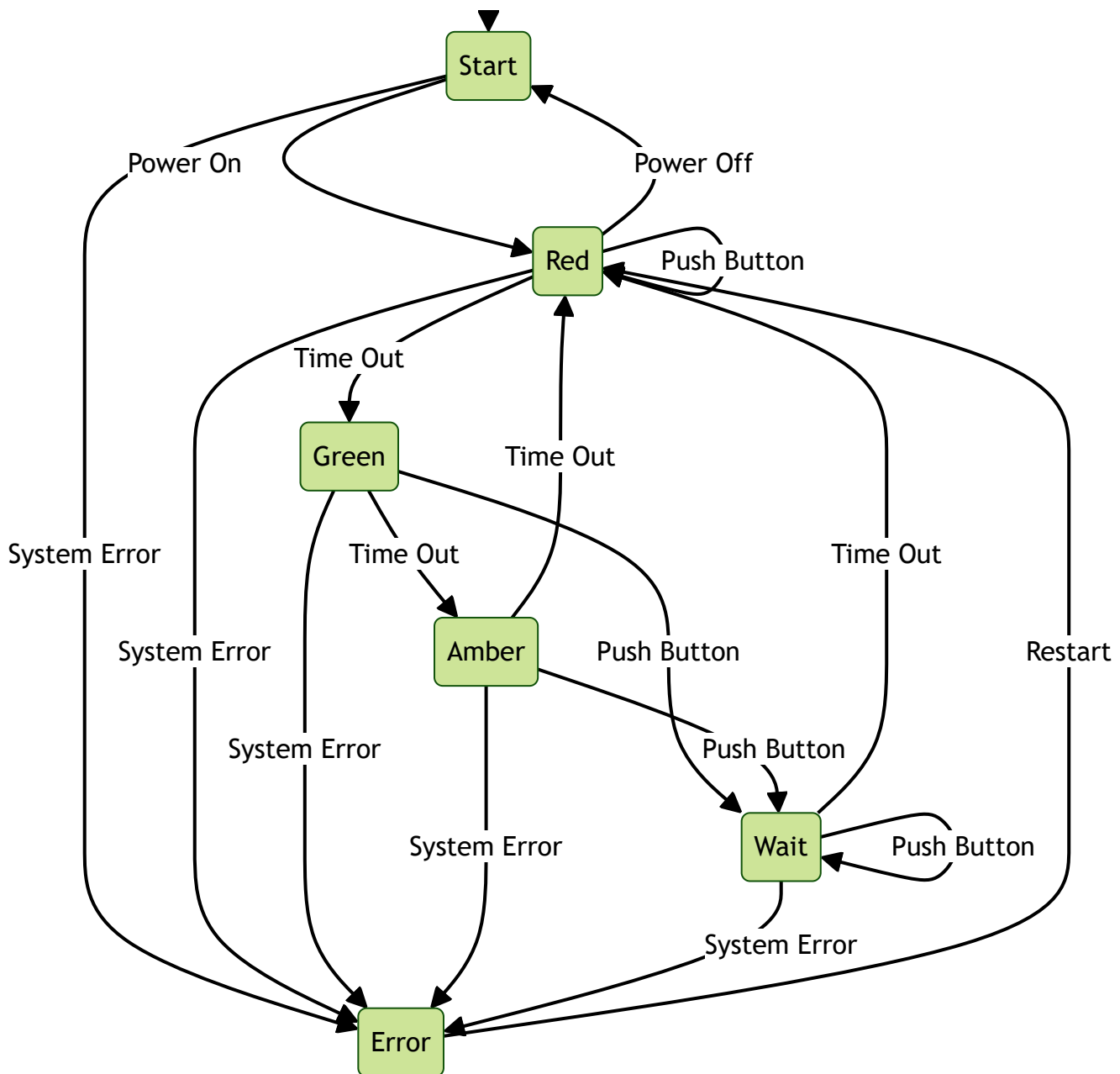
Let's consider a simple example of a traffic lights model. The FSM for the traffic lights could have states like ' `Red` ', ' `Green` ', ' `Amber` ', etc. Signals occur as the time elapses or pedestrians pushed the button for crossing. The transition operations need to change the lights' states.
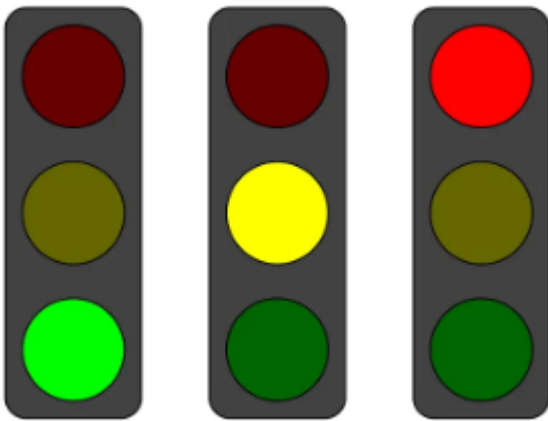
The state diagram of the Traffic Light FSM is illustrated below.

Traffic Lights FSM

Start

Power On Power Off

Red Push Button

Time Out

Green Time Out

System Error

System Error Time Out Time Out

System Error Amber Push Button Restart

Push Button

System Error Push Button

Wait Push Button

System Error

Error

When comparing the state transition diagram above to the definition in the previous section, we can describe the Traffic Light FSM as follows:

- **State** : The Traffic Light FSM encompasses the states of `Start` , `Red` , `Green` , `Amber` , `Wait` , and `Error` .
- **Signals** : The FSM responds to signals including `Power On` , `Power Off` , `Time Out` , `Push Button` , `Restart` , and `System Error` .
- **Actions** : State transitions between different states.
- **Initial State** : The initial state for this FSM is `Start` .

- **`Final State`** : The determination of the final state is context-dependent. For instance, `Error` might be considered the final state, although the `Restart` signal can trigger a transition to change the light to the `Red` state. Multiple states can be included in the final state set to terminate the FSM at various points.

Adjacency List

In practical applications, an adjacency list becomes particularly useful when the state diagram grows too complex to explore easily. It serves as a collection of unordered lists, primarily used to represent a finite graph. Each unordered list within an adjacency list describes the set of transitions originating from a specific vertex in the graph. This representation is one of several commonly employed methods for graph representation in computer programming.

For relatively simple state machines, explicitly writing out adjacency lists might seem unnecessary. However, for more intricate systems, you'll likely find yourself creating a form of it, making it a valuable practice.

In the context of this specific discussion, we won't adhere to the precise definition of an adjacency list, as we're not describing a "**set of neighbours**" but rather a "**set of transitions**", as shown in the table below.

State	Transition to
Start	Red or Error
Red	Start or Green or Error
Green	Amber or Wait or Error
Amber	Red or Wait or Error
Wait	Red or Error
Error	Red

Your Task

1. Traffic Light FSM

This task is to implement the FSM for this Traffic Lights model according to the state diagram above.

The Repository

The repository below contains the template code of an FSM implementation in Python.

Fork the repository (NOT clone!) and work on your fork. This will enable you to submit a pull request in the end.

https://github.falmouth.ac.uk/Daniel-Zhang/COMP712-Finite-State-Machine.git

The Code Structure

The repository contains 3 file.

- `fsm.py` : it is the implementation of a simple FSM with all the components introduced above except `Actions` , which is the task for you to implement in a separate script during this workshop. *This design will make sure the `FSM` model response to any model if the `transitions` and `actions` are defined properly somewhere*.
- `fsm_demo.py` : it is a template python script that you are going to working on.
- `fsm_example.pyc` : it is an example of completed implementation of Traffic Light FSM, which can be simulated by running `python fsm_example.pyc`
  - It was compiled using `python 3.12.0` , which should work with all python `3.x` version. Please let the tutor know if it doesn't work and a newer version will be updated.
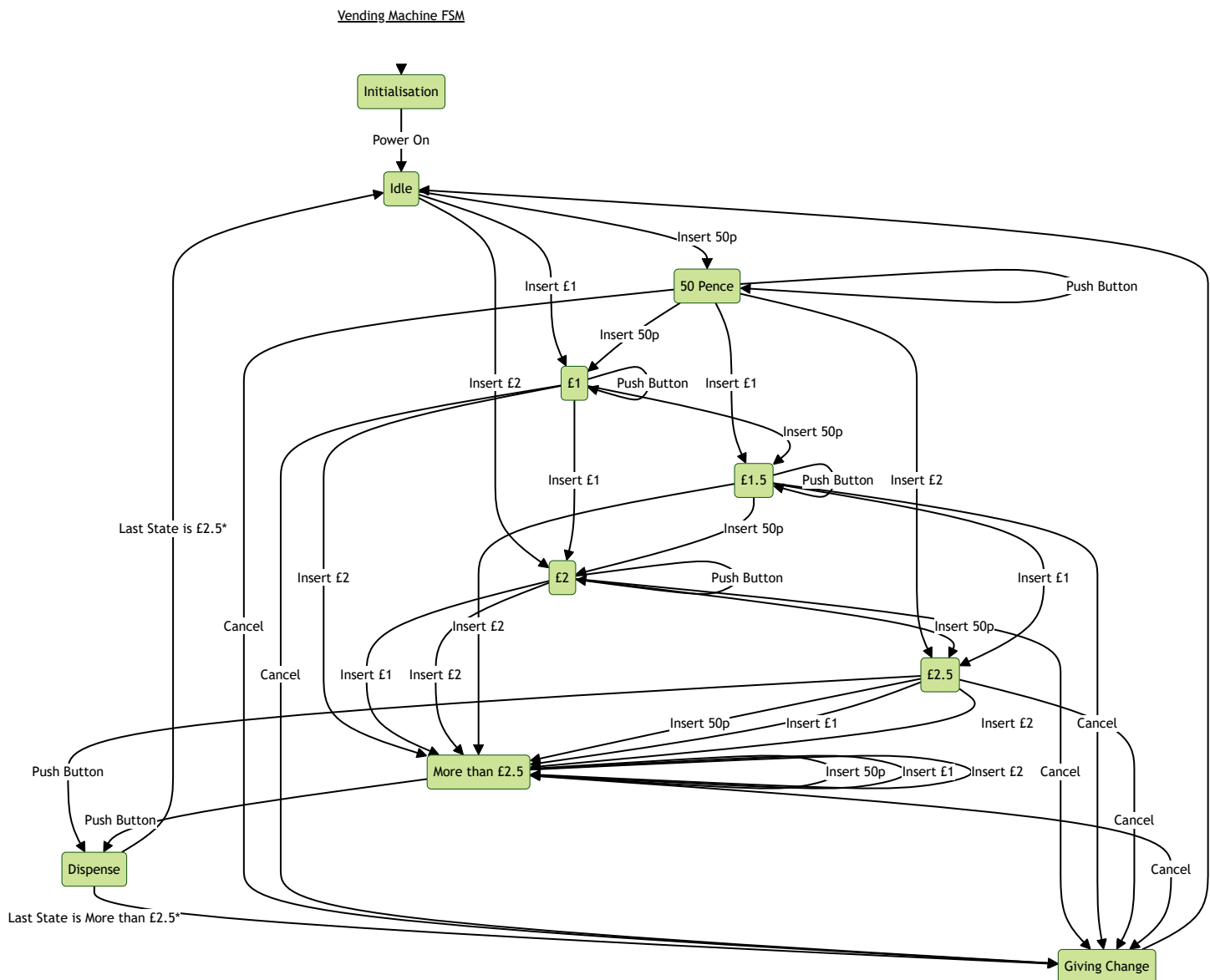
Exercise

- complete the implementation of the Traffic Lights FSM
  - make sure it works as expected
  - increase the number of actions to simulate the process longer
  - add one or more states to the model

2. Vending Machine FSM

This task is to implement a Vending Machine FSM that sells Coca to the user. This special machine only accepts 50p, £1, and £2 coins and each can of Coca costs £2.5 (a little bit expensive 😜). Additionally, we will assume the vending machine never runs out of stock for economic reasons. Similarly, it never runs out of change. This example is a perfect computational problem to model with a finite state machine!

The state diagram is illustrated below.

Vending Machine FSM

Exercise

- Work out the adjacency list to help you understand the diagram if necessary.
- Create a new Python file (`.py`), either inherit from the FSM model defined in `fsm.py` or start from scratch.
- Implement the Vending Machine FSM to meet the requirements outlined in the state diagram.

Submit Your Code

Feel free to submit a pull request to the original repository to showcase your work if you wish to share your solution with others.

Further Reading

- Wiki - Finite State Machine
- Python State Machine
- An implementation of FSM