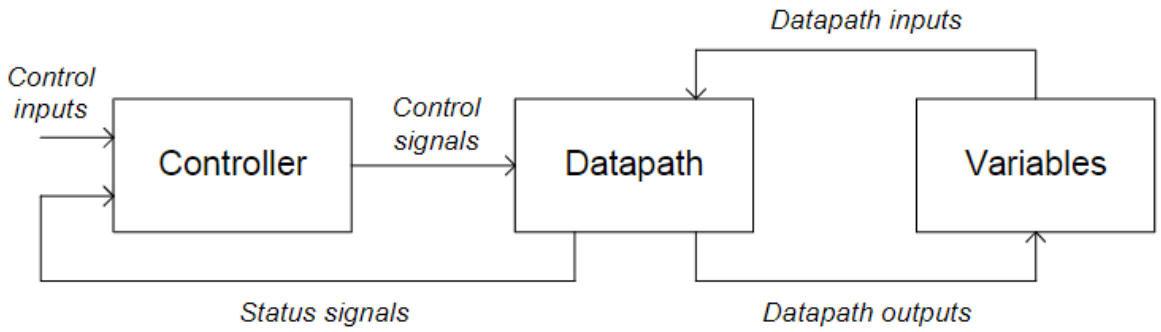
02135 Assignment 1:

Implementation of an FSMD simulator in Python



Report

By Adrian Maciejewski, Elias Haynie-Gay, and Balint Regoczi

Section One: Code Overview

1. Given code: Input Reading, Support Initialization, and Description Printing

The code we submitted starts with the code provided to us. This code takes program arguments to define the number of iterations that the state machine will preform. It also takes the program arguments for the location of the FSMD description and stimuli files, which are parsed using the xmltodict library. The provided code then prints the description, and initializes several dictionaries based on the descriptions from the previously parsed files.

2. Initialization

Outside of the provided code we needed to do a few things before carryingout cycles. We print the required start message including the initial state and the initial variable values. We then initialize several variables to be used in our cycle progression, including currentState, currentInstruction, previousInstruction, and cycle.

3. Cycle progression

Our program uses a while loop to simulate the cycles, with one iteration through the loop representing one cycle. The loop terminates when finished, as described in the “cycle termination” section. At the beginning of each cycle the ‘currentState’ and ‘currentInstruction’ are stored in ‘previousState’ and ‘previousInstruction’ to later check if there is an infinite loop. We then check for stimuli input in ‘executeInput()’, which uses the provided code. The transition is then handled in ‘calculateAndExecuteTransition()’ (which will be further explained in the Transition Handling section), which also returns the state, instruction, and transition to be used for printing in the ‘printCycle()’ method. Lastly in the loop, the cycle variable is increased by one.

4. Cycle Termination

The while loop which we use to carry out each cycle will terminate based on three conditions. First, the number of cycles carried out so far must be less than or equal to the number of requested iterations, so that the loop will stop at the requested number of iterations, and not exceed it. Next, it must not be the end state, as determined by isEndState() which uses the provided code to check the stimuli file for the end state and return if it equals the current state or not. Lastly, the cycles will only continue to progress if they’re not in an infinite loop, as determined by isInfiniteLoop() which makes sure the current state dosen’t equal the previous state, and that the current instruction and previous instructions weren’t ‘NOP’. If all of these conditions pass, the cycle progresses as explained in the above section.

5. Transition Handling

The next transition is found and executed by ‘calculateAndExecuteTransition()’ which works by first confirming the transitions of the given state are of type list, and putting them in one if not. The code then iterates through each avaliable transition, checking if the condition for each possible transition is true, using the given ‘evaluate_condition()’ method. If it is true the corresponding instruction is executed using ‘execute_instruction()’, which also returns that instruction, transition, and next state to the main loop to be printed.

6. Completion

Once the while loop has exited as specified in the above ‘Cycle termination’ section, the program must carry out the last few steps to complete. It double checks that its the end state, and the prints the last cycle similarly to what was described in ‘cycle progression’. It then prints the closing messages, and is complete.

Section two: Example Use - Test 2

In the case of Test 2, where we run the command, or one similar to:
‘python fsmd-sim.py 100 'fsmdSim\test_2\gcd_desc.xml' 'fsmdSim\test_2\gcd_stim.xml’’

Here, it begins by printing the description, and then starts the simulation:

```
---Start simulation---
At the beginning of the simulation the status is:
Variables:
  var_A = 0
  var_B = 0
Initial state: INITIALIZE
```

Cycle 0, Initalizes A and B to 100, and 12.
Cycle 1, Tests if A is greater than B, then on seeing that is sets next state to AminB
Cycle 2, Sets a to a minus b, then goes back to test
Cycle 3, tests if a is greater than B, then on seeing it is sets next state to AminB
This process repeats until cycle 22, where A equals B, so the program goes to state Finish
Cycle 22, A=B, Program is finished.

We see we get the same output as the given file.

Section three: Our Test - Test 3

Concept

The idea behind our test case was based on the algorithm calculating *in how many ways it is possible to climb a given number of stairs in an infinite staircase if it is possible to climb only oneor two stairs at a time and the starting point is step number zero*.

Initially we created a simple implementation of the algorithm in Python. Starting with a recursive algorithm, then converting it to an iterative algorithm. Finally we adjusted it to have the easiest possible form which we could use to create FSMD. Although the version we ended up with was not the simplest from the raw Python perspective, it was better suited for our needs. Despite the fact that In the meantime, our implementation of the FSMD slightly diverged from the initial algorithm which we tried to implement identically step by step, the main concept remained the same.

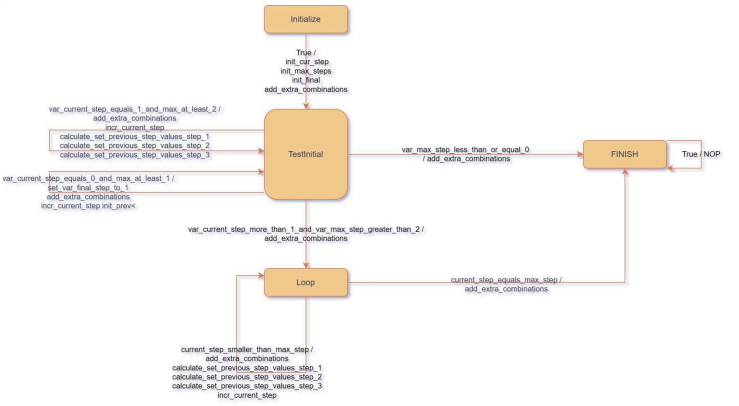
```python
def loopCountStairs(n):
    finalStep = 0

    if n <= 2:
        finalStep = n
        return finalStep

    prev2 = 1
    prev1 = 2
    currentStairNumber = 2
    while True:
        if currentStairNumber >= n:
            break

        next = prev1 + prev2
        prev2 = prev1
        prev1 = next
        finalStep = next

        currentStairNumber += 1

    return finalStep
```

While creating the above algorithm we took into account time and space complexity and adjusted it for maximum performance. Therefore, the space complexity is linear, because there is only one while loop. In addition, the space complexity is constant, because we only need two previous numbers to calculate the next one.

The above code was tested on a vast set of test cases, thus we are certain that it works and can be used as a reference to calculate what the outcome for a given parameter *n* should be.

State graph

Having the algorithm implemented, the next thing we star5ted doing was creating a state graph. It helped us conceptualise how we should approach the problem from the FSMD design perspective. Throughout the process of incremental improvements we settled on the following version of the diagram.



The most important parts of the diagram include:

- States
    - INITIALIZE - initial state
    - TESTINITIAL - state right after initialization, it is responsible for accounting for conditions when var_final_combinations is 0, 1 and 2 while also checking that those steps are only calculated when var_max_step is large enough.
    - LOOP - it is responsible for calculating var_final_combinations for values larger than 2.
    - FINISH - end state.

- Conditions - Conditions belonging to e.g. TESTINITIAL are exclusive and exactly one is always true.
- Operations - They determine what should happen when a given transition is true e.g. for condition current_step_smaller_than_max_step a bunch of operations is executed to complete an iteration of a loop.

Description file

When it comes to implementing the FSMD in the descrition file, we tried to follow the algorithm we created and the state graph. Initially we converted the previously createed algorithm line by line into the xml file with all the variables, input, conditions and operations. Nonetheless, with time this file slightly diverged from being one to one representation of out algorithm. One of the reasons for it was the need to utilise stimuli file in an engaging way.

Transition table

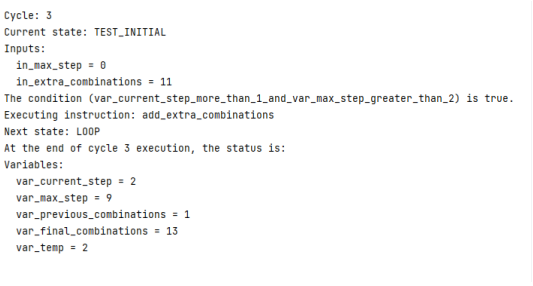
Present state	Next state	Input Condition	Output Signals
Initialize	TestInitial	‘True’	init_cur_step init_max_steps init_final add_extra_combinations
TestInitial	Finish	Max step <= 0	add_extra_combinations
	Loop	Current step > 1 And max step > 2	add_extra_combinations
	TestInitial	Current Step == 0 And Max Step >= 1	set_var_final_step_to_1 add_extra_combinations incr_current_step init_prev
	TestInitial	Current Step == 1 And Max Step >= 2	add_extra_combinations incr_current_step calculate_set_previous_step_values_step_1 calculate_set_previous_step_values_step_2 calculate_set_previous_step_values_step_3
Loop	Finish	Current Step == Max step	add_extra_combinations
	Loop	Current Step < Max Step	add_extra_combinations calculate_set_previous_step_values_step_1 calculate_set_previous_step_values_step_2 calculate_set_previous_step_values_step_3 incr_current_step

Stimuli file

There are several ways in which we utilise stimuli.

1. We use it to initialise *in_max_step*
2. We use it to set the end state
3. Finally, we also use it to add a specific number defined by *in_extra_combinations* to *var_final_combinations* in a specific cycle. It was done in order to demonstrate that it is possible to use stimuli in a more advanced way than simply specify initial values of the program and a final state. Although, from the use case perspective the validity of this feature could be questioned, it proves that stimuli has real-time impact on the state of the FSMD. This feature is also one of the reasons why final description file is not perfectly aligned with initial algorithm.

Cycle analysis



The below given screenshots represent data outputed in cycle 3 and 4.

The following observations can be made:

- var_current_step determines for which step var_final_combinations has just been calculated
- the purpose is to have var_final_combinations determine in how many ways it is possible to climb the stairs for var_current_step (at the end for var_max_step)
- For var_current_step=1, var_final_combinations was equal 1, because in this case: var_final_combinations = var_final_combinations + var_previous_combinations + in_extra_combinations.
- In cyscle 4 this operation happens for the following values: var_final_combinations = 1 + 13 + 0.
- in_max_step is initialised once at the beginning
- var_temp is a temporaty variable and can be ignored