Embedded System Design and Modeling

ECE382N.23, Fall 2022

Homework #1

Models of Computation

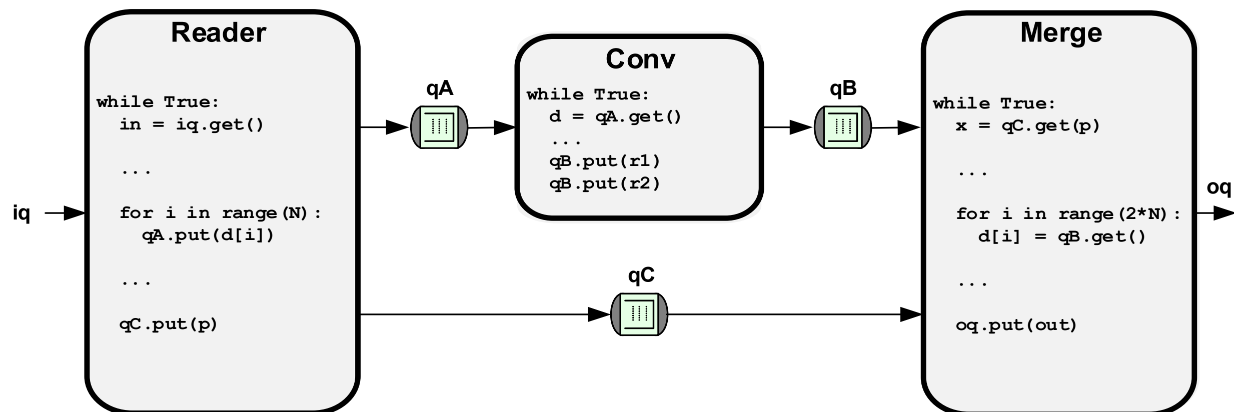
Assigned: September 9, 2022
Due: September 23, 2022

Instructions:

- Please submit your solutions via Canvas. Submissions should include a single PDF with the writeup and a single Zip or Tar archive for any supplementary files (e.g., source files, which has to be compilable by simply running 'make' and should include a README with instructions for running each model).
- You may discuss the problems with your classmates but make sure to submit your own independent and individual solutions.
- Some questions might not have a clearly correct or wrong answer. In general, grading is based on your arguments and reasoning for arriving at a solution.

Problem 1.1: KPN

Given the following KPN model:



(a) Show one possible schedule when executing this KPN using Park's algorithm. Show the schedule in terms of queue reads and writes made by each process. Is Park's algorithm able to run this model in a complete, bounded, and non-terminating manner? What is the buffer size needed for each queue?

(b) Can this KPN be executed in smaller memory (queue sizes) than what Park's requires? If not, why not? If yes, show the schedule and associated memory requirements.

(c) Write a Python program that realizes and executes this KPN model with the bounded queue sizes that you determined in the previous question(s) when N = 6. Your program must meet the following requirements:

   i. Expect one integer argument as input representing the number of elements that will be sent on ***iq*** (note that the size of ***iq*** can be 1 regardless of this input value)

ii. Simulate each KPN process as a different thread. Hint: Use Python's Queue and Thread classes.

iii. Queue sizes remain constant throughout the execution of the program

iv. Log every read and write operation in a different line (using print) with the format: *<task_name> <get/put> <queue_name>* (e.g., *Conv get qA*)

(d) This KPN can be converted into an equivalent synchronous dataflow (SDF) model. Show the converted model, its repetition vector and further conversion into an equivalent HSDF model. What impact does the conversion have on memory requirements (buffer sizes)?

(e) Using your code from question (c) as a basis, write a Python program that realizes and executes the SDF model that you proposed in question (d). Your program must meet the following requirements:

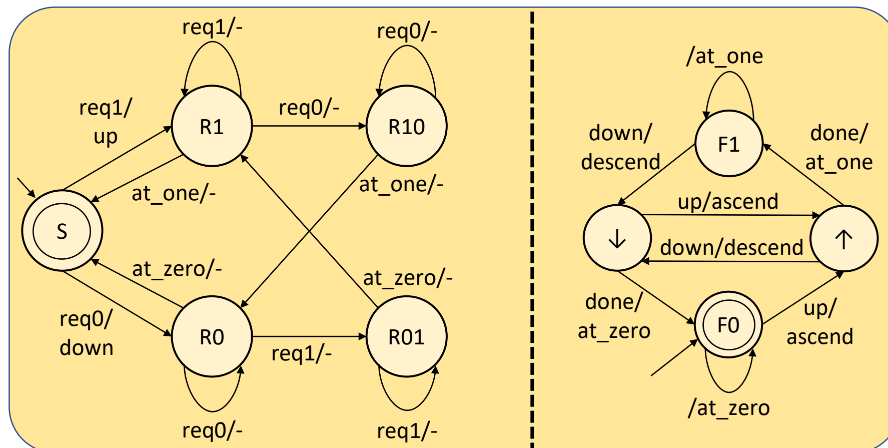
i. Expect one integer argument as input representing the number of elements that will be sent as inputs to the SDF

ii. Expose the full actor parallelism, i.e. do not implement a static schedule but provide code that executes each actor as one thread converted from your original KPN

iii. Log an actor name every time it is fired (using print)

Problem 1.2: HCFSM

Given in the following is the HCFSM model of an elevator control system for a two-story building. The machine has 3 external inputs: *req0* (for requesting elevator from floor #0), *req1* (for requesting elevator from floor #1), *done* (notification of mechanical system), 2 external outputs: *ascend* and *descend* (commands for mechanical system), and 4 internal signals for communication between state machines (*up*, *down*, *at_one*, *at_zero*). Unless specified otherwise, signals are by default absent and states have implicit self-transitions.



(a) Demonstrate the operation of the elevator for answering a request from floor #1 and going from floor #0 to floor #1. You can assume that the elevator starts at floor #0 (corresponding to start states *S* and *F0*). Show the sequence of events and state transitions.

(b) Statecharts are a well-known and widely-used formalism for specifying state machines following HCFSM semantics. Sismic is a Statechart library for Python. Use Sismic to model, execute and validate your elevator HCFSM above. Define the state machine with YAML and include it in your submission as *hcfsm.yaml*. Use the same signal names as shown in the figure above. Follow these tips:

- Define input signals as events
- Validate your state machine with Python.
  - Use *interpreter.queue(<input_name>)* to control the inputs
  - Print the output of *interpreter.execute()* to observe state transitions
  - Print *interpreter.context* to observe the value of signals and outputs

(c) Convert the HCFSM into an equivalent single, flat FSM. Define the flat FSM with Sismic and show the new diagram. Hint: you can export the YAML to PlantUML format and create a diagram with the online tool. Execute the flattened Sismic model and validate that it behaves the same.

(d) The state machine has a bug that can make the elevator get stuck at a floor. Write a Python script that uses your *hcfsm.yaml* model to show a trace of input events and transitions that demonstrates the bug. Include the script as *bug.py* in your submission.

(e) Show how the original HCFSM and the flattened FSM need to be modified to remember at least the second request. You can indicate changes in the graphs above or directly in the YAML files. Execute the modified Sismic models to validate that the bug is fixed.