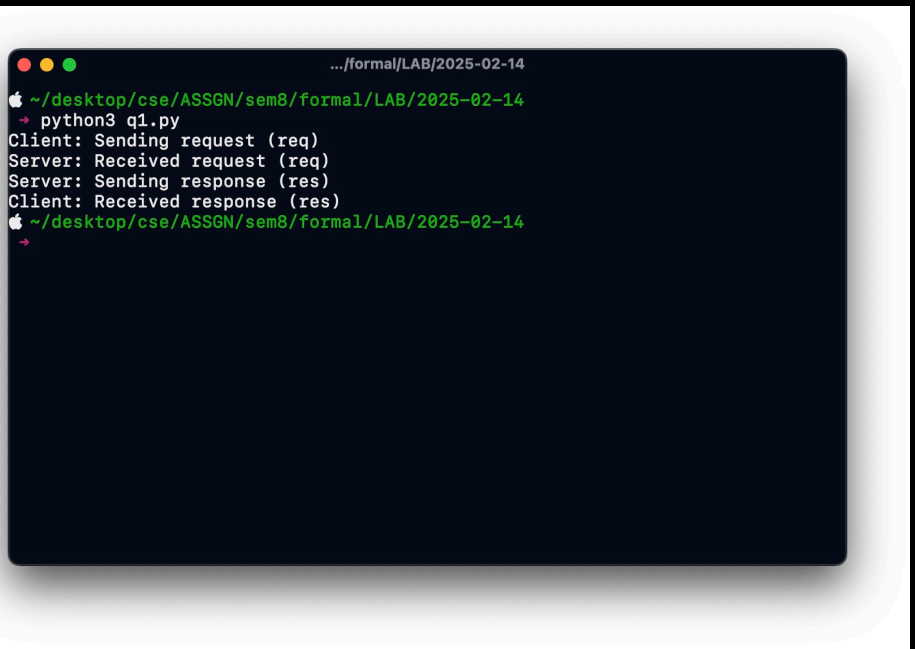
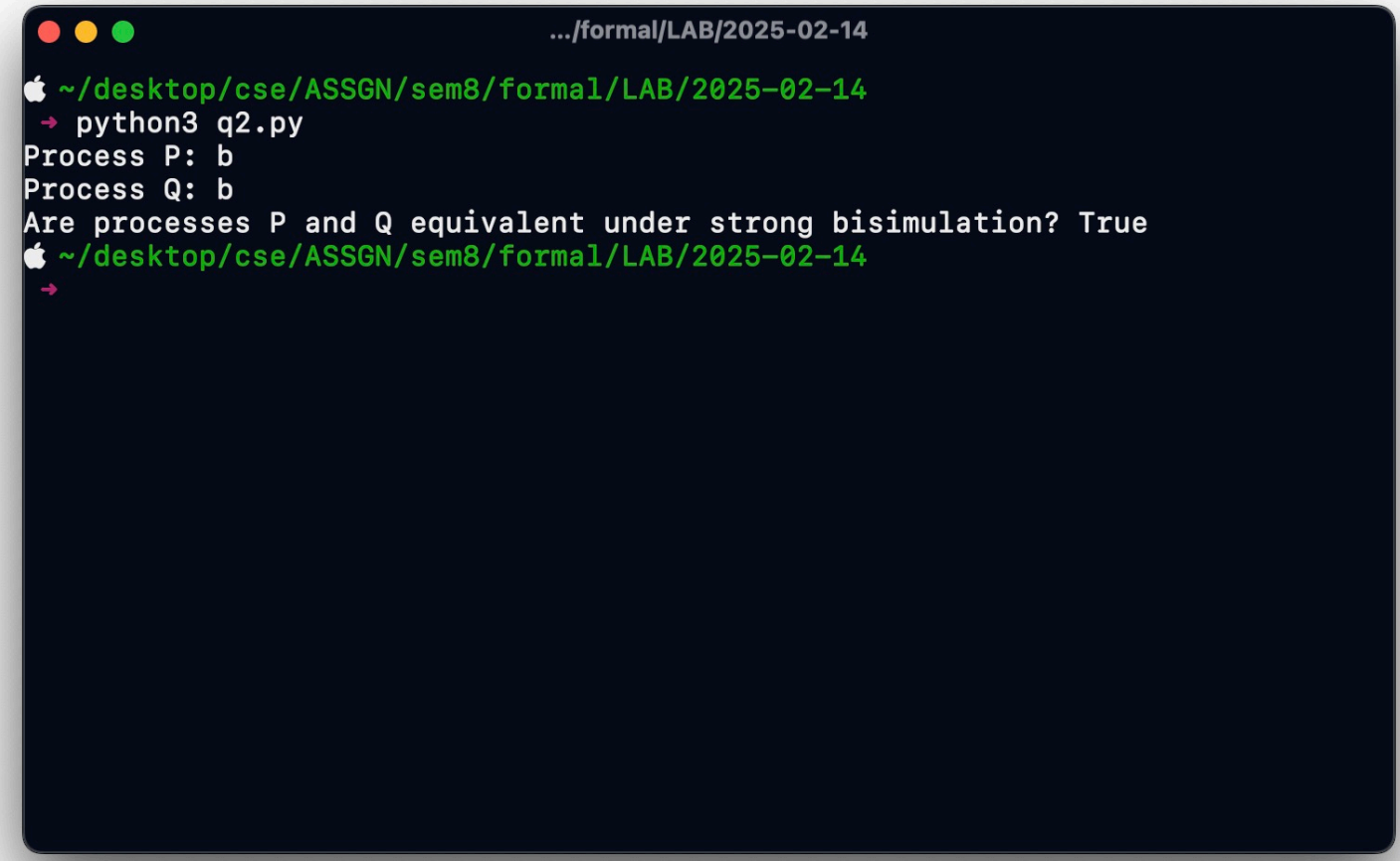
ASSIGNMENT 3
14/02/25

NAME : SHRESTH SONKAR
REGNO : 20214272
GROUP : CS8D
TOPIC : FORMAL METHODS
CODE : CS-18201

```python
# 1. Write a Python program to model a client-server
interaction using CCS process constructions. The
# client sends a request (req) and waits for a response
(res), while the server listens for req, processes it,
# and responds with res. Simulate the sequential
communication between both processes.

import asyncio

async def client(server_queue, client_queue):
    print("Client: Sending request (req)")
    await server_queue.put("req")

    response = await client_queue.get()
    print(f"Client: Received response ({response})")

async def server(server_queue, client_queue):
    request = await server_queue.get()
    print(f"Server: Received request ({request})")

    await asyncio.sleep(1)

    print("Server: Sending response (res)")
    await client_queue.put("res")


async def main():
    server_queue = asyncio.Queue()
    client_queue = asyncio.Queue()
    await asyncio.gather(client(server_queue,
client_queue), server(server_queue, client_queue))

asyncio.run(main())
```

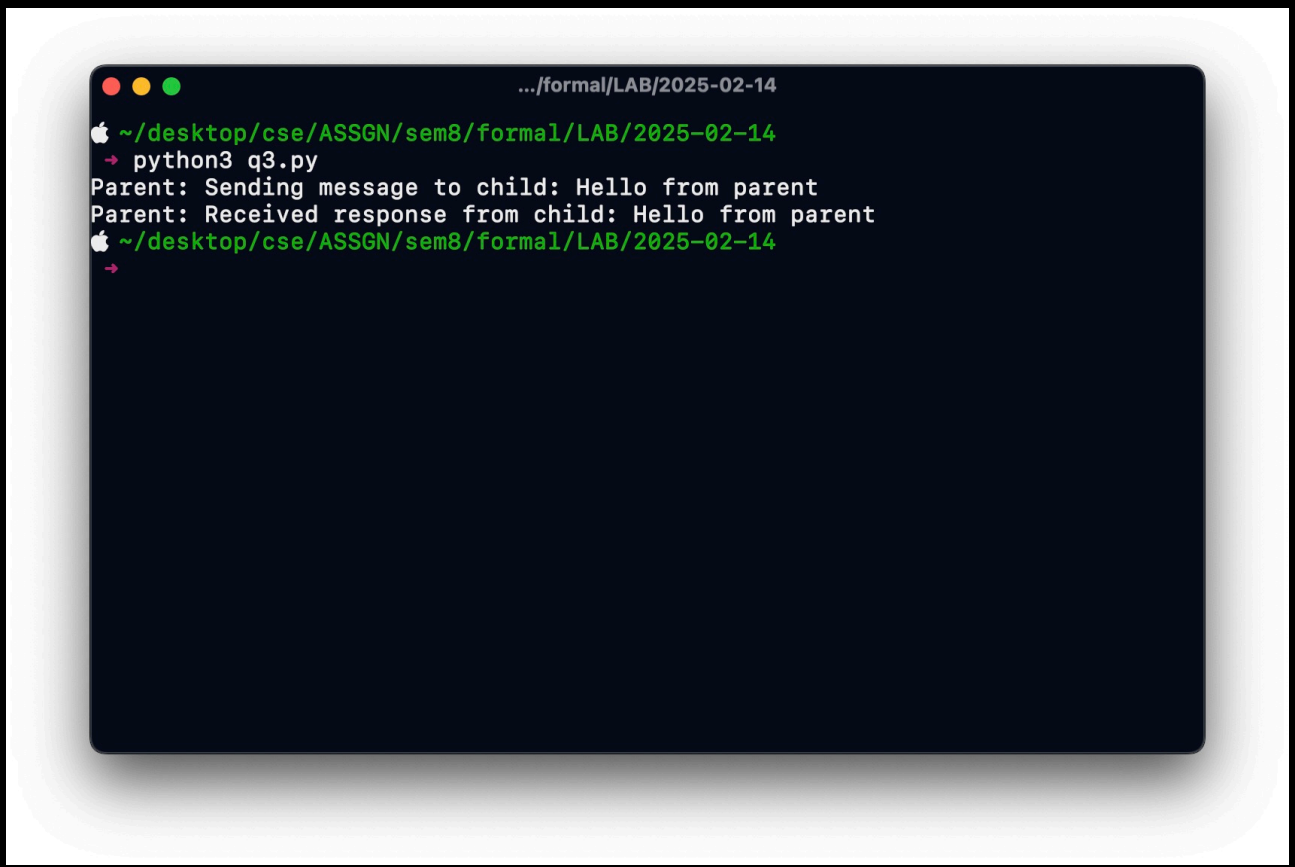


```python
# 2. Develop a Python program that defines two CCS
processes, P and Q, executing actions a and b.
# Apply relabeling (a → b) and restriction (\{a}) to
synchronize their execution. Verify whether they
# remain equivalent under strong bisimulation.

class Process:
    def __init__(self, name, action):
        self.name = name
        self.action = action

    def relabel(self, old_action, new_action):
        if self.action == old_action:
            self.action = new_action

    def restrict(self, restricted_actions):
        if self.action in restricted_actions:
            self.action = None

def strong_bisimulation(p1, p2):
    return p1.action == p2.action

P = Process("P", "a")
Q = Process("Q", "b")

P.relabel("a", "b")

P.restrict({"a"})
Q.restrict({"a"})

is_equivalent = strong_bisimulation(P, Q)

print(f"Process P: {P.action}")
print(f"Process Q: {Q.action}")
print(f"Are processes P and Q equivalent under strong
bisimulation? {is_equivalent}")
```



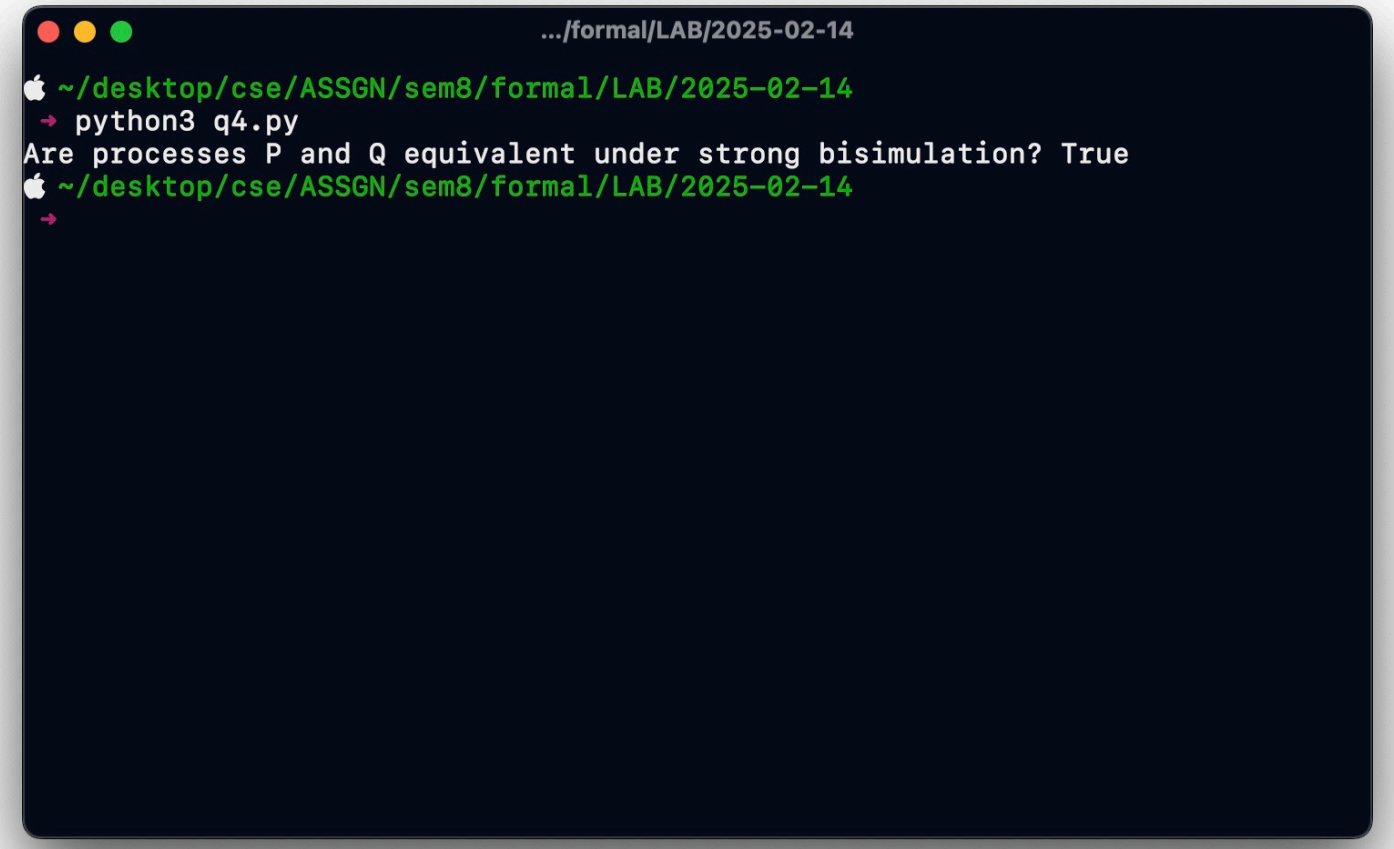

```python
# 3. Simulate a mobile communication system using Pi-Calculus in Python, where a parent process
# dynamically spawns a child process and exchanges messages over a dynamically created channel.
# Ensure the child process correctly receives and processes the messages.

import asyncio
import random

class Channel:
    def __init__(self, name):
        self.name = name
        self.queue = asyncio.Queue()

    async def send(self, message):
        await self.queue.put(message)

    async def receive(self):
        return await self.queue.get()

async def parent_process():
```

```python
    channel = Channel("dynamic_channel")

    asyncio.create_task(child_process(channel))

    message = "Hello from parent"
    print(f"Parent: Sending message to child:
{message}")
    await channel.send(message)

    response = await channel.receive()
    print(f"Parent: Received response from child:
{response}")

async def child_process(channel):
    message = await channel.receive()
    print(f"Child: Received message from parent:
{message}")

    response = f"Processed: {message}"

    await channel.send(response)
    print(f"Child: Sent response to parent:
{response}")

async def main():
    await parent_process()

asyncio.run(main())
```



```python
# 4. Write a Python program to define two finite-state
processes in CCS and implement a bisimulation
# equivalence check between them. The program should
determine whether both processes exhibit the
# same behavior using strong bisimulation principles
from CWB.

class Process:
    def __init__(self, name):
        self.name = name
        self.states = {}
        self.initial_state = None

    def add_state(self, state, transitions):
        self.states[state] = transitions

    def set_initial_state(self, state):
        self.initial_state = state

def strong_bisimulation(p1, p2):
    visited = set()
    return check_bisimulation(p1.initial_state,
p2.initial_state, p1, p2, visited)

def check_bisimulation(state1, state2, p1, p2,
visited):
    if (state1, state2) in visited:
        return True
    visited.add((state1, state2))

    transitions1 = p1.states.get(state1, {})
    transitions2 = p2.states.get(state2, {})

    if transitions1.keys() != transitions2.keys():
        return False

    for action in transitions1:
        next_states1 = transitions1[action]
        next_states2 = transitions2[action]

        if len(next_states1) != len(next_states2):
            return False

        for next_state1, next_state2 in
zip(next_states1, next_states2):
```

```
            if not check_bisimulation(next_state1,
next_state2, p1, p2, visited):
                return False

    return True

P = Process("P")
Q = Process("Q")

P.add_state("s0", {"a": ["s1"], "b": ["s2"]})
P.add_state("s1", {"c": ["s0"]})
P.add_state("s2", {})

Q.add_state("t0", {"a": ["t1"], "b": ["t2"]})
Q.add_state("t1", {"c": ["t0"]})
Q.add_state("t2", {})

P.set_initial_state("s0")
Q.set_initial_state("t0")

is_equivalent = strong_bisimulation(P, Q)

print(f"Are processes P and Q equivalent under strong
bisimulation? {is_equivalent}")
```



```python
# 5. Design a Python program to simulate a fair
resource scheduler for two processes (P and Q). Ensure
#     that both processes get access to a shared
resource in a round-robin manner, preventing livelock
or
# starvation. Verify fairness using CCS-style modeling.

import threading
import time

class Resource:
    def __init__(self):
        self.lock = threading.Lock()
        self.turn = 'P'

    def access(self, process_name):
        while True:
            with self.lock:
                if self.turn == process_name:
                    print(f"Process {process_name} is
accessing the shared resource")
                    time.sleep(1)
                    self.turn = 'P' if process_name ==
'Q' else 'Q'

def process(resource, process_name):
    while True:
        resource.access(process_name)

resource = Resource()
p_thread = threading.Thread(target=process,
args=(resource, 'P'))
q_thread = threading.Thread(target=process,
args=(resource, 'Q'))

p_thread.start()
q_thread.start()

p_thread.join()
q_thread.join()
```


python3 q5.py
➜ python3 q5.py
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource
Process Q is accessing the shared resource
Process P is accessing the shared resource