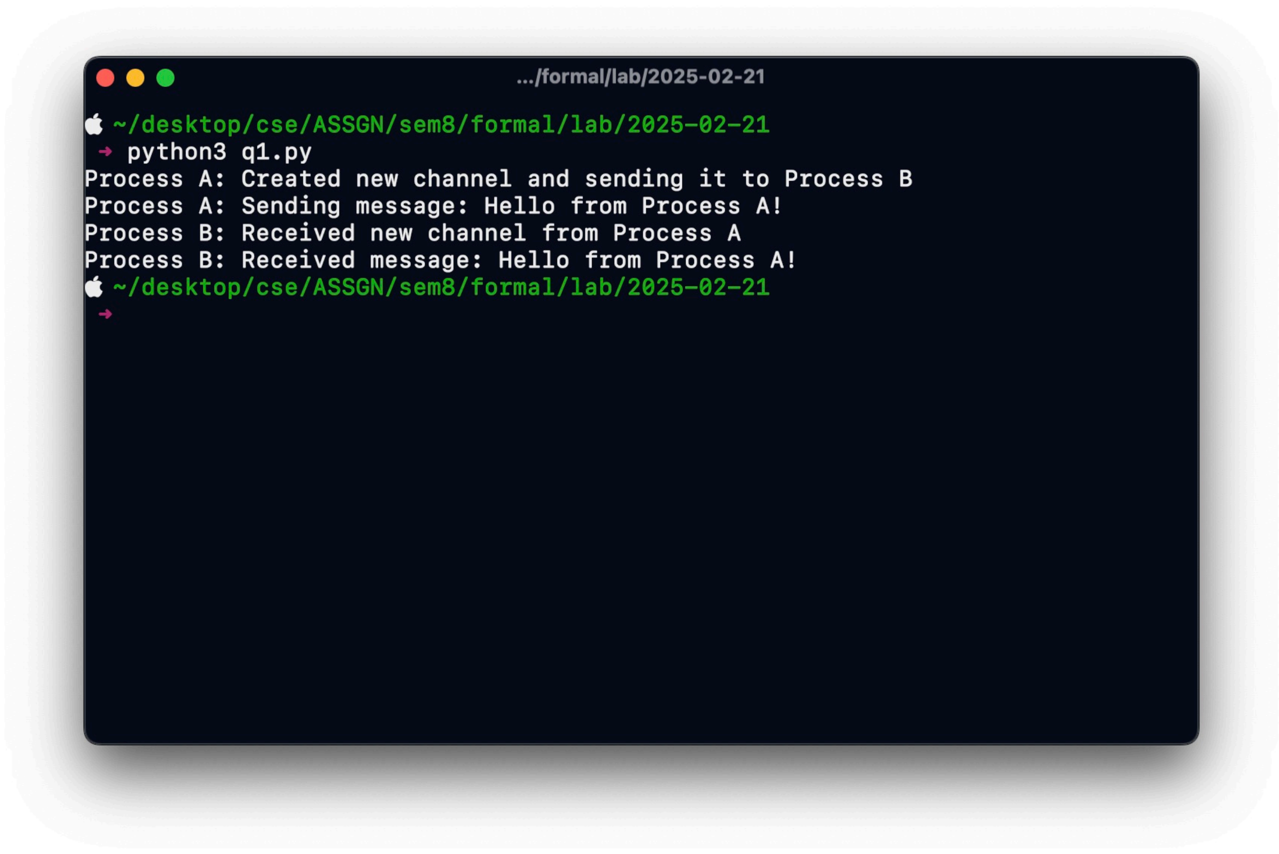
ASSIGNMENT 5
21/02/25

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

```python
# 1. Implement a system where two processes communicate
using the π-Calculus framework,
# dynamically creating channels and exchanging
messages. Ensure that the processes interact
# correctly and handle concurrent execution.

import multiprocessing

def process_a(channel_queue, name):
    new_channel = multiprocessing.Manager().Queue()
    print(f"{name}: Created new channel and sending it
to Process B")
    channel_queue.put(new_channel)

    message = f"Hello from {name}!"
    print(f"{name}: Sending message: {message}")
    new_channel.put(message)

def process_b(channel_queue, name):
    new_channel = channel_queue.get()
    print(f"{name}: Received new channel from Process
A")

    message = new_channel.get()
    print(f"{name}: Received message: {message}")

def main():
    with multiprocessing.Manager() as manager:
        channel_queue = manager.Queue()

        process1 =
multiprocessing.Process(target=process_a,
args=(channel_queue, "Process A"))
        process2 =
multiprocessing.Process(target=process_b,
args=(channel_queue, "Process B"))

        process1.start()
        process2.start()

        process1.join()
        process2.join()

if __name__ == "__main__":
    main()
```



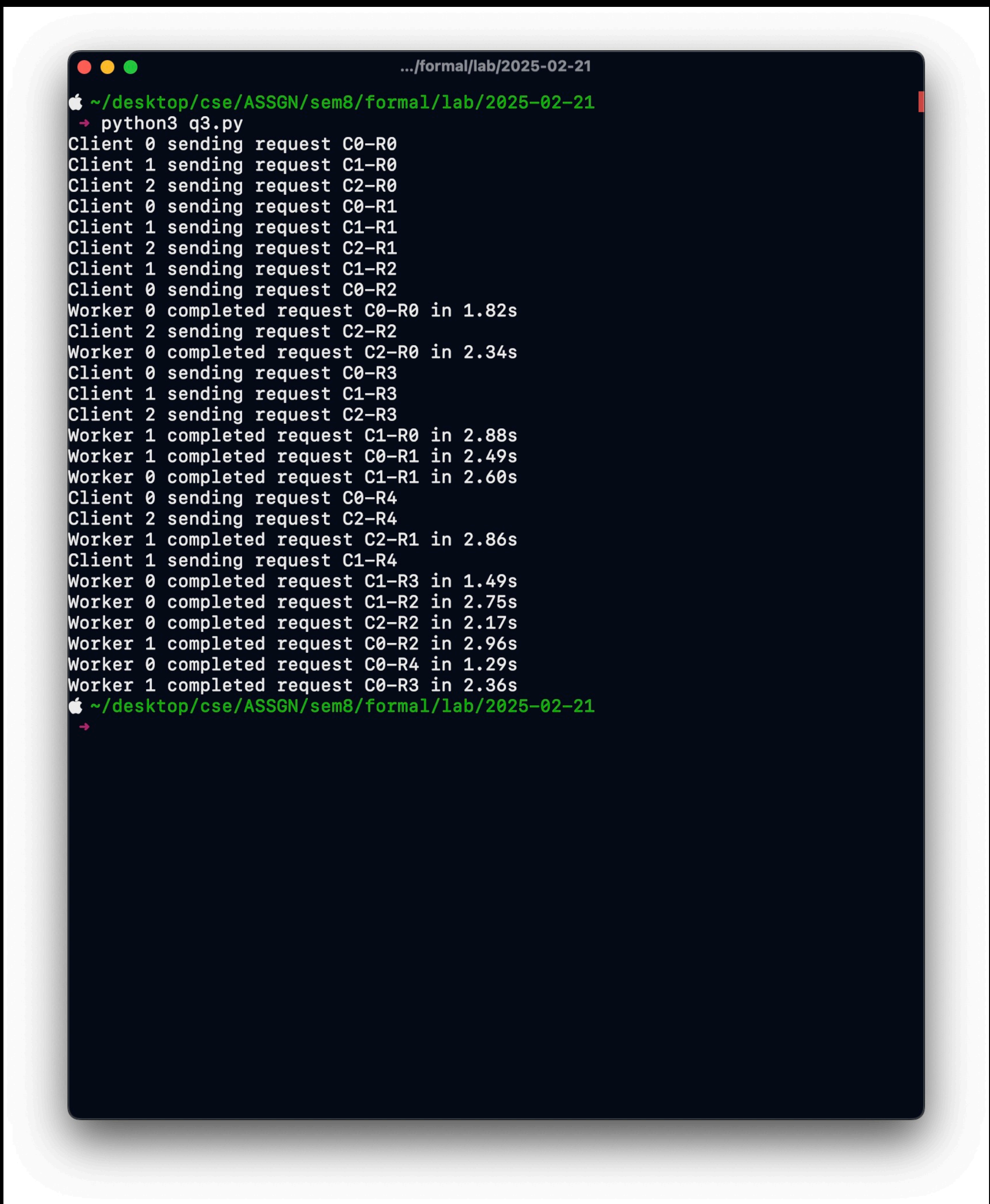

```python
# 2. Develop a Python program that models a system of three CCS processes executing actions in
# parallel, ensuring synchronization where required. Introduce relabeling and restriction to study
# their impact on process behavior.

class CCSProcess:
    def __init__(self, name, actions):
        self.name = name
        self.actions = actions

    def relabel(self, relabel_map):
        self.actions = {relabel_map.get(action, action) for action in self.actions}

    def restrict(self, restricted_actions):
        self.actions -= restricted_actions

    def synchronize(self, other):
        common_actions = self.actions & other.actions
        return CCSProcess(f"{self.name}|{other.name}", common_actions)

    def __repr__(self):
        return f"{self.name}: {self.actions}"

P1 = CCSProcess("P1", {"a", "b", "c"})
```

```python
P2 = CCSProcess("P2", {"b", "c", "d"})
P3 = CCSProcess("P3", {"c", "d", "e"})

relabel_map = {"a": "x", "d": "y"}
P1.relabel(relabel_map)
P2.relabel(relabel_map)
P3.relabel(relabel_map)

restricted_actions = {"c"}
P1.restrict(restricted_actions)
P2.restrict(restricted_actions)
P3.restrict(restricted_actions)

P1_P2 = P1.synchronize(P2)
P1_P2_P3 = P1_P2.synchronize(P3)

print("After Relabeling and Restriction:")
print(P1)
print(P2)
print(P3)
print("\nSynchronization Result:")
print(P1_P2_P3)
```

```
.../formal/lab/2025-02-21
~/desktop/cse/ASSGN/sem8/formal/lab/2025-02-21
→ python3 q2.py
After Relabeling and Restriction:
P1: {'b', 'x'}
P2: {'b', 'y'}
P3: {'y', 'e'}

Synchronization Result:
P1|P2|P3: set()
~/desktop/cse/ASSGN/sem8/formal/lab/2025-02-21
→
```

```
# 3. Simulate a process algebra-based load balancer
where multiple clients send requests to a central
# dispatcher that distributes tasks among available
workers. Verify that requests are handled
# fairly without starvation.

import asyncio
import random

class Worker:
    def __init__(self, worker_id):
        self.worker_id = worker_id

    async def process_request(self, request_id):
        process_time = random.uniform(1, 3)
        await asyncio.sleep(process_time)
        print(f"Worker {self.worker_id} completed
request {request_id} in {process_time:.2f}s")

class Dispatcher:
    def __init__(self, num_workers):
        self.workers = [Worker(i) for i in
range(num_workers)]
        self.queue = asyncio.Queue()
        self.round_robin_index = 0

    async def add_request(self, request_id):
        await self.queue.put(request_id)

    async def dispatch_requests(self):
        while True:
            request_id = await self.queue.get()
            worker =
self.workers[self.round_robin_index]
            self.round_robin_index =
(self.round_robin_index + 1) % len(self.workers)

asyncio.create_task(worker.process_request(request_id))
            self.queue.task_done()

async def client(dispatcher, client_id, num_requests):
    for i in range(num_requests):
        request_id = f"C{client_id}-R{i}"
        print(f"Client {client_id} sending request
{request_id}")
```

```python
        await dispatcher.add_request(request_id)
        await asyncio.sleep(random.uniform(0.5, 1.5))

async def main():
    num_clients = 3
    num_requests_per_client = 5
    num_workers = 2

    dispatcher = Dispatcher(num_workers)
    asyncio.create_task(dispatcher.dispatch_requests())
    client_tasks = 
[asyncio.create_task(client(dispatcher, i, 
num_requests_per_client)) for i in range(num_clients)]

    await asyncio.gather(*client_tasks)
    await dispatcher.queue.join()

if __name__ == "__main__":
    asyncio.run(main())
```



```
.../formal/lab/2025-02-21
 ~/desktop/cse/ASSGN/sem8/formal/lab/2025-02-21
➜ python3 q3.py
Client 0 sending request C0-R0
Client 1 sending request C1-R0
Client 2 sending request C2-R0
Client 0 sending request C0-R1
Client 1 sending request C1-R1
Client 2 sending request C2-R1
Client 1 sending request C1-R2
Client 0 sending request C0-R2
Worker 0 completed request C0-R0 in 1.82s
Client 2 sending request C2-R2
Worker 0 completed request C2-R0 in 2.34s
Client 0 sending request C0-R3
Client 1 sending request C1-R3
Client 2 sending request C2-R3
Worker 1 completed request C1-R0 in 2.88s
Worker 1 completed request C0-R1 in 2.49s
Worker 0 completed request C1-R1 in 2.60s
Client 0 sending request C0-R4
Client 2 sending request C2-R4
Worker 1 completed request C2-R1 in 2.86s
Client 1 sending request C1-R4
Worker 0 completed request C1-R3 in 1.49s
Worker 0 completed request C1-R2 in 2.75s
Worker 0 completed request C2-R2 in 2.17s
Worker 1 completed request C0-R2 in 2.96s
Worker 0 completed request C0-R4 in 1.29s
Worker 1 completed request C0-R3 in 2.36s
 ~/desktop/cse/ASSGN/sem8/formal/lab/2025-02-21
➜
```

```
# 4. Implement a Python-based verification system that
checks whether two given finite-state
# processes are equivalent using strong bisimulation.
The program should take two process
# descriptions as input and determine whether they
exhibit the same external behavior.

from collections import defaultdict, deque

class LTS:
    def __init__(self, transitions, initial_state):
        self.transitions = defaultdict(set,
transitions)
        self.initial_state = initial_state
        self.states = set(transitions.keys()) | {s for
targets in transitions.values() for (_, s) in targets}

    def get_transitions(self, state):
        return self.transitions[state]

def bisimulation_check(lts1, lts2):
    queue = deque([(lts1.initial_state,
lts2.initial_state)])
    visited = set()

    while queue:
        s1, s2 = queue.popleft()
        if (s1, s2) in visited:
            continue
        visited.add((s1, s2))

        trans1 = defaultdict(set)
        trans2 = defaultdict(set)

        for label, target in lts1.get_transitions(s1):
            trans1[label].add(target)
        for label, target in lts2.get_transitions(s2):
            trans2[label].add(target)

        if set(trans1.keys()) != set(trans2.keys()):
            return False

        for label in trans1:
            if trans1[label] != trans2[label]:
                return False
```

```
            for t1, t2 in zip(sorted(trans1[label]),
sorted(trans2[label])):
                queue.append((t1, t2))
    return True

transitions1 = {
    'q0': {('a', 'q1'), ('b', 'q2')},
    'q1': {('c', 'q3')},
    'q2': {('c', 'q3')},
    'q3': set()
}

transitions2 = {
    'p0': {('a', 'p1'), ('b', 'p2')},
    'p1': {('c', 'p3')},
    'p2': {('c', 'p3')},
    'p3': set()
}

lts1 = LTS(transitions1, 'q0')
lts2 = LTS(transitions2, 'p0')

print("Equivalent under strong bisimulation?",
bisimulation_check(lts1, lts2))
```

```python
# 5. Design a producer-consumer system using CCS principles, ensuring correct message passing
# and proper synchronization between the producer and the consumer while preventing
# deadlocks.

import threading
import queue
import time

class Producer(threading.Thread):
    def __init__(self, buffer, event):
        super().__init__()
        self.buffer = buffer
        self.event = event

    def run(self):
        for i in range(5):
            time.sleep(1)
            item = f"Item-{i}"
            self.buffer.put(item)
            print(f"Produced: {item}")
            self.event.set()
        self.buffer.put(None)
        self.event.set()

class Consumer(threading.Thread):
    def __init__(self, buffer, event):
        super().__init__()
        self.buffer = buffer
        self.event = event

    def run(self):
        while True:
            self.event.wait()
            self.event.clear()
            item = self.buffer.get()
            if item is None:
                break
            print(f"Consumed: {item}")
            time.sleep(2)

if __name__ == "__main__":
    buffer = queue.Queue()
    event = threading.Event()
```

```python
producer = Producer(buffer, event)
consumer = Consumer(buffer, event)

producer.start()
consumer.start()

producer.join()
consumer.join()

print("Processing complete.")
```

```
python3 q5.py
~/desktop/cse/ASSGN/sem8/formal/lab/2025-02-21
➜ python3 q5.py
Produced: Item-0
Consumed: Item-0
Produced: Item-1
Consumed: Item-1
Produced: Item-2
Produced: Item-3
Consumed: Item-2
Produced: Item-4
Consumed: Item-3
```