**Formal Methods Lab**

**Assignment -4**

Task 1: π-Calculus Communication System

In π-Calculus, processes communicate by sending and receiving messages over dynamically created channels. Below is a Python implementation of two processes communicating using π-Calculus principles:

```python

import threading

# Channel for communication

channel = []

# Process 1: Sends a message

def process1():

global channel

message = "Hello from Process 1"

print(f"Process 1 sending: {message}")

channel.append(message) # Send message to channel

# Process 2: Receives a message

def process2():

global channel

while not channel: # Wait for message

pass

message = channel.pop(0)

print(f"Process 2 received: {message}")

# Create threads for processes

thread1 = threading.Thread(target=process1)

thread2 = threading.Thread(target=process2)

# Start threads

thread1.start()

thread2.start()

# Wait for threads to finish

thread1.join()

thread2.join()

```

Explanation:

- Two processes (`process1` and `process2`) communicate via a shared channel (`channel`).

- `process1` sends a message, and `process2` receives it.

- Threading ensures concurrent execution.

---

Task 2: CCS Processes with Relabeling and Restriction

CCS (Calculus of Communicating Systems) models concurrent processes that synchronize on actions. Below is a Python implementation of three CCS processes with relabeling and restriction:

```python

import threading

# Actions and synchronization

actions = {"a", "b", "c"}

sync\_actions = {"a"}

# Process A

def process\_a():

print("Process A executing action 'a'")

# Relabeling: Action 'a' becomes 'x'

print("Relabeled action 'a' to 'x'")

# Process B

def process\_b():

print("Process B executing action 'b'")

# Process C

def process\_c():

print("Process C executing action 'c'")

# Synchronize processes

def synchronize():

print("Synchronizing on action 'a'")

process\_a()

# Create threads

thread\_a = threading.Thread(target=process\_a)

thread\_b = threading.Thread(target=process\_b)

thread\_c = threading.Thread(target=process\_c)

thread\_sync = threading.Thread(target=synchronize)

# Start threads

thread\_a.start()

thread\_b.start()

thread\_c.start()

thread\_sync.start()

# Wait for threads to finish

thread\_a.join()

thread\_b.join()

thread\_c.join()

thread\_sync.join()

```

Explanation:

- Three processes (`process\_a`, `process\_b`, `process\_c`) execute actions in parallel.

- Relabeling is demonstrated by renaming action `'a'` to `'x'`.

- Synchronization is enforced on action `'a'`.

---

Task 3: Process Algebra-Based Load Balancer

A load balancer distributes tasks among workers. Below is a Python simulation:

```python

import threading

import queue

import time

# Task queue and worker pool

task\_queue = queue.Queue()

workers = ["Worker1", "Worker2", "Worker3"]

# Client: Sends requests

def client(name):

for i in range(5):

task = f"Task {i} from {name}"

print(f"{name} sending {task}")

task\_queue.put(task)

time.sleep(1)

# Worker: Processes tasks

def worker(name):

while True:

task = task\_queue.get()

if task is None:

break

print(f"{name} processing {task}")

time.sleep(2) # Simulate task processing

task\_queue.task\_done()

# Create client and worker threads

clients = [threading.Thread(target=client, args=(f"Client{i}",)) for i in range(2)]

workers = [threading.Thread(target=worker, args=(worker,)) for worker in workers]

# Start threads

for c in clients:

c.start()

for w in workers:

w.start()

# Wait for clients to finish

for c in clients:

c.join()

# Wait for all tasks to be processed

task\_queue.join()

# Stop workers

for w in workers:

task\_queue.put(None)

for w in workers:

w.join()

```

Explanation:

- Clients send tasks to a shared queue.

- Workers process tasks from the queue.

- The system ensures fair task distribution and prevents starvation.

---

Task 4: Strong Bisimulation Verification

Strong bisimulation checks if two processes exhibit the same external behavior. Below is a Python implementation:

```python

# Define processes as finite-state machines

process1 = {

"q0": {"a": "q1"},

"q1": {"b": "q0"}

}

process2 = {

"p0": {"a": "p1"},

"p1": {"b": "p0"}

}

# Strong bisimulation check

def is\_bisimilar(process1, process2):

# Check if transitions match

for state1, transitions1 in process1.items():

for action, next\_state1 in transitions1.items():

if action not in process2.get(state1, {}):

return False

if process2[state1][action] != next\_state1:

return False

return True

# Test bisimulation

print("Are the processes bisimilar?", is\_bisimilar(process1, process2))

```

Explanation:

- Two processes are represented as finite-state machines.

- The `is\_bisimilar` function checks if their transitions match.

---

Task 5: Producer-Consumer System in CCS

A producer-consumer system ensures proper synchronization and prevents deadlocks. Below is a Python implementation:

```python

import threading

import queue

import time

# Shared buffer

buffer = queue.Queue(maxsize=5)

# Producer

def producer():

for i in range(10):

item = f"Item {i}"

buffer.put(item)

print(f"Produced {item}")

time.sleep(1)

# Consumer

def consumer():

while True:

item = buffer.get()

if item is None:

break

print(f"Consumed {item}")

time.sleep(2)

buffer.task\_done()

# Create threads

producer\_thread = threading.Thread(target=producer)

consumer\_thread = threading.Thread(target=consumer)

# Start threads

producer\_thread.start()

consumer\_thread.start()

# Wait for producer to finish

producer\_thread.join()

# Wait for buffer to empty

buffer.join()

# Stop consumer

buffer.put(None)

consumer\_thread.join()

```

Explanation:

- The producer adds items to a shared buffer.

- The consumer removes items from the buffer.

- Synchronization ensures no deadlocks occur.