**Assignment 6**

**Formal Methods Lab**

1. Petri Net for Synchronization Problem

```python

class PetriNet:

def \_\_init\_\_(self):

self.places = {}

self.transitions = {}

self.arcs = []

def add\_place(self, name, tokens=0):

self.places[name] = tokens

def add\_transition(self, name):

self.transitions[name] = {'input': [], 'output': []}

def add\_arc(self, source, target, weight=1):

self.arcs.append((source, target, weight))

if source in self.places:

self.transitions[target]['input'].append((source, weight))

else:

self.transitions[source]['output'].append((target, weight))

def is\_enabled(self, transition):

for place, weight in self.transitions[transition]['input']:

if self.places[place] < weight:

return False

return True

def fire(self, transition):

if not self.is\_enabled(transition):

return False

# Consume input tokens

for place, weight in self.transitions[transition]['input']:

self.places[place] -= weight

# Produce output tokens

for place, weight in self.transitions[transition]['output']:

self.places[place] += weight

return True

def check\_deadlock(self):

"""Check if the net is deadlock-free"""

# A deadlock occurs when no transitions are enabled

# but some tokens remain in the net

total\_tokens = sum(self.places.values())

if total\_tokens == 0:

return True # No deadlock if no tokens

for transition in self.transitions:

if self.is\_enabled(transition):

return True # At least one transition can fire

return False # Deadlock detected

# Model a synchronization problem (mutual exclusion)

def model\_mutual\_exclusion():

net = PetriNet()

# Places

net.add\_place('p1', 1) # Process 1 ready

net.add\_place('p2', 1) # Process 2 ready

net.add\_place('mutex', 1) # Mutex token

# Transitions

net.add\_transition('t1') # Process 1 enter CS

net.add\_transition('t2') # Process 1 exit CS

net.add\_transition('t3') # Process 2 enter CS

net.add\_transition('t4') # Process 2 exit CS

# Arcs

net.add\_arc('p1', 't1')

net.add\_arc('mutex', 't1', 1)

net.add\_arc('t1', 'cs1', 1) # Process 1 in CS

net.add\_arc('cs1', 't2')

net.add\_arc('t2', 'p1', 1)

net.add\_arc('t2', 'mutex', 1)

net.add\_arc('p2', 't3')

net.add\_arc('mutex', 't3', 1)

net.add\_arc('t3', 'cs2', 1) # Process 2 in CS

net.add\_arc('cs2', 't4')

net.add\_arc('t4', 'p2', 1)

net.add\_arc('t4', 'mutex', 1)

return net

# Verify deadlock freedom

mutex\_net = model\_mutual\_exclusion()

print("Mutual exclusion model is deadlock-free:", mutex\_net.check\_deadlock())

# Introduce a deadlock by removing the mutex token

mutex\_net.places['mutex'] = 0

print("Modified model is deadlock-free:", mutex\_net.check\_deadlock())

```

## 2. Workflow System with Petri Nets

```python

class WorkflowSystem:

def \_\_init\_\_(self):

self.net = PetriNet()

self.build\_order\_processing()

def build\_order\_processing(self):

# Places

self.net.add\_place('order\_received', 0)

self.net.add\_place('payment\_processed', 0)

self.net.add\_place('inventory\_checked', 0)

self.net.add\_place('order\_shipped', 0)

self.net.add\_place('order\_completed', 0)

self.net.add\_place('payment\_failed', 0)

self.net.add\_place('out\_of\_stock', 0)

# Transitions

self.net.add\_transition('receive\_order')

self.net.add\_transition('process\_payment')

self.net.add\_transition('check\_inventory')

self.net.add\_transition('ship\_order')

self.net.add\_transition('complete\_order')

self.net.add\_transition('fail\_payment')

self.net.add\_transition('reject\_order')

# Arcs

self.net.add\_arc('receive\_order', 'order\_received')

self.net.add\_arc('order\_received', 'process\_payment')

self.net.add\_arc('order\_received', 'check\_inventory')

self.net.add\_arc('process\_payment', 'payment\_processed')

self.net.add\_arc('process\_payment', 'payment\_failed')

self.net.add\_arc('check\_inventory', 'inventory\_checked')

self.net.add\_arc('check\_inventory', 'out\_of\_stock')

self.net.add\_arc('payment\_processed', 'ship\_order')

self.net.add\_arc('inventory\_checked', 'ship\_order')

self.net.add\_arc('ship\_order', 'order\_shipped')

self.net.add\_arc('order\_shipped', 'complete\_order')

self.net.add\_arc('complete\_order', 'order\_completed')

self.net.add\_arc('payment\_failed', 'reject\_order')

self.net.add\_arc('out\_of\_stock', 'reject\_order')

def simulate(self):

# Start with an order

self.net.fire('receive\_order')

# Process the order

while True:

enabled = [t for t in self.net.transitions if self.net.is\_enabled(t)]

if not enabled:

break

# Choose a random enabled transition

import random

transition = random.choice(enabled)

print(f"Firing: {transition}")

self.net.fire(transition)

# Check termination

if self.net.places['order\_completed'] > 0:

print("Order completed successfully!")

return True

elif (self.net.places['payment\_failed'] > 0 or

self.net.places['out\_of\_stock'] > 0):

print("Order rejected!")

return False

print("Workflow terminated abnormally")

return False

# Analyze the workflow

workflow = WorkflowSystem()

print("Initial marking:", workflow.net.places)

workflow.simulate()

print("Final marking:", workflow.net.places)

# Verify correctness properties

def verify\_workflow\_properties(net):

# Property 1: Order completion requires both payment and inventory check

net.places = {'order\_received': 1, 'payment\_processed': 1, 'inventory\_checked': 1}

assert net.is\_enabled('ship\_order')

# Property 2: Payment failure leads to rejection

net.places = {'order\_received': 1, 'payment\_failed': 1}

assert net.is\_enabled('reject\_order')

print("All properties verified!")

verify\_workflow\_properties(workflow.net)

```

## 3. Client-Server Communication with CCS

```python

class CCSProcess:

def \_\_init\_\_(self, name):

self.name = name

self.actions = []

self.current = 0

def prefix(self, action, continuation):

self.actions.append(('prefix', action, continuation))

return self

def choice(self, \*processes):

self.actions.append(('choice', list(processes)))

return self

def parallel(self, \*processes):

self.actions.append(('parallel', list(processes)))

return self

def restrict(self, actions):

self.actions.append(('restrict', actions))

return self

def simulate(self, steps=10):

for \_ in range(steps):

if not self.step():

break

def step(self):

if self.current >= len(self.actions):

return False

action\_type = self.actions[self.current][0]

if action\_type == 'prefix':

\_, action, continuation = self.actions[self.current]

print(f"{self.name} performs {action}")

self.current = 0

continuation.step()

return True

elif action\_type == 'choice':

\_, processes = self.actions[self.current]

# Non-deterministic choice

import random

chosen = random.choice(processes)

print(f"{self.name} makes a choice")

self.current = 0

chosen.step()

return True

elif action\_type == 'parallel':

\_, processes = self.actions[self.current]

# Simulate parallel composition with interleaving

print(f"{self.name} running in parallel")

for p in processes:

p.step()

return True

elif action\_type == 'restrict':

\_, actions = self.actions[self.current]

print(f"{self.name} restricts {actions}")

self.current += 1

return self.step()

self.current += 1

return self.step()

# Model client-server communication

def model\_client\_server():

# Client process

client = CCSProcess("Client")

client.prefix("request", CCSProcess("ClientWait")) \

.prefix("receive", CCSProcess("Client"))

# Server process

server = CCSProcess("Server")

server.prefix("receive", CCSProcess("ServerProcess")) \

.prefix("respond", CCSProcess("Server"))

# Server processing

server\_process = CCSProcess("ServerProcess")

server\_process.prefix("process", CCSProcess("ServerRespond"))

# System composition

system = CCSProcess("System")

system.parallel(client, server) \

.restrict({'request', 'receive', 'process', 'respond'})

return system

# Simulate the system

print("Client-Server Communication Simulation")

system = model\_client\_server()

system.simulate(10)

```

## 4. Online Transaction System with Process Algebra

```python

class ProcessAlgebra:

def \_\_init\_\_(self, name):

self.name = name

self.processes = {}

def define(self, name, actions):

self.processes[name] = actions

def execute(self, process\_name):

if process\_name not in self.processes:

raise ValueError(f"Process {process\_name} not defined")

actions = self.processes[process\_name]

for action in actions:

if isinstance(action, str):

print(f"{self.name}.{process\_name} performs {action}")

elif isinstance(action, tuple) and action[0] == 'choice':

# Non-deterministic choice

import random

chosen = random.choice(action[1])

print(f"{self.name}.{process\_name} chooses path")

self.execute(chosen)

elif isinstance(action, tuple) and action[0] == 'parallel':

# Parallel composition

print(f"{self.name}.{process\_name} runs in parallel")

for p in action[1]:

self.execute(p)

elif isinstance(action, tuple) and action[0] == 'sync':

# Synchronization

print(f"{self.name}.{process\_name} synchronizes on {action[1]}")

self.execute(action[2])

# Model an online transaction system

def model\_transaction\_system():

pa = ProcessAlgebra("TransactionSystem")

# Define processes

pa.define("User", [

"login",

("choice", ["browse", "checkout"]),

"logout"

])

pa.define("browse", [

"view\_items",

("choice", ["continue\_browsing", "add\_to\_cart"])

])

pa.define("checkout", [

"view\_cart",

("sync", "payment", "process\_payment"),

"confirm\_order"

])

pa.define("Database", [

("parallel", ["handle\_query", "handle\_update"]),

"commit"

])

pa.define("handle\_query", [

"receive\_query",

"execute\_query",

"return\_results"

])

pa.define("handle\_update", [

"receive\_update",

"validate",

"apply\_update"

])

# Main system process

pa.define("System", [

("parallel", ["User", "Database"]),

"log\_metrics"

])

return pa

# Execute the system

print("Online Transaction System Specification")

system = model\_transaction\_system()

system.execute("System")

```

## 5. Distributed Computation with Pi-Calculus

```python

class PiCalculus:

def \_\_init\_\_(self):

self.processes = {}

self.channels = set()

def new\_channel(self, name):

self.channels.add(name)

return name

def define(self, name, definition):

self.processes[name] = definition

def send(self, channel, message):

return ('send', channel, message)

def receive(self, channel, callback):

return ('receive', channel, callback)

def parallel(self, \*processes):

return ('parallel', list(processes))

def restrict(self, channel, process):

return ('restrict', channel, process)

def simulate(self, process\_name):

if process\_name not in self.processes:

raise ValueError(f"Process {process\_name} not defined")

process = self.processes[process\_name]

self.\_execute(process)

def \_execute(self, process):

if isinstance(process, tuple):

if process[0] == 'send':

\_, channel, message = process

print(f"Sending '{message}' on channel '{channel}'")

elif process[0] == 'receive':

\_, channel, callback = process

print(f"Waiting to receive on channel '{channel}'")

# Simulate receiving a message

received\_msg = f"message\_{channel}"

print(f"Received '{received\_msg}' on channel '{channel}'")

self.\_execute(callback(received\_msg))

elif process[0] == 'parallel':

\_, processes = process

print("Running processes in parallel")

for p in processes:

self.\_execute(p)

elif process[0] == 'restrict':

\_, channel, subprocess = process

print(f"New private channel '{channel}' created")

self.\_execute(subprocess)

elif isinstance(process, str):

if process in self.processes:

self.\_execute(self.processes[process])

else:

print(f"Process {process} completes")

# Model mobile processes

def model\_mobile\_processes():

pi = PiCalculus()

# Create channels

a = pi.new\_channel('a')

b = pi.new\_channel('b')

# Define processes

pi.define("Client", [

pi.send(a, "request"),

pi.receive(a, lambda x: [

print(f"Client received {x}"),

"Client"

])

])

pi.define("Server", [

pi.receive(a, lambda x: [

print(f"Server received {x}"),

pi.send(a, "response"),

"Server"

])

])

pi.define("MobileAgent", [

pi.restrict('c', [

pi.send(c, "mobile\_data"),

pi.receive(c, lambda x: [

print(f"Agent received {x} at new location"),

"MobileAgent"

])

])

])

# System composition

pi.define("System", [

pi.parallel("Client", "Server", "MobileAgent")

])

return pi

# Simulate the system

print("Distributed Computation with Pi-Calculus")

system = model\_mobile\_processes()

system.simulate("System")

```

These implementations demonstrate:

1. Petri Net for synchronization with deadlock detection

2. Workflow system modeling order processing with correctness verification

3. CCS client-server communication model

4. Process algebra specification for transaction systems

5. Pi-calculus model for mobile processes with channel restriction