ASSIGNMENT 1
24/01/25

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

```
# 1. Write a Python program to implement a simple state
transition system.

class State:
    def __init__(self, name):
        self.name = name
        self.transitions = {}

    def add_transition(self, input_symbol, next_state):
        self.transitions[input_symbol] = next_state

    def get_next_state(self, input_symbol):
        return self.transitions.get(input_symbol, None)


class StateMachine:
    def __init__(self, initial_state):
        self.current_state = initial_state

    def transition(self, input_symbol):
        next_state =
self.current_state.get_next_state(input_symbol)
        if next_state:
            print(f"Transitioning from
{self.current_state.name} to {next_state.name} on input
'{input_symbol}'")
            self.current_state = next_state
        else:
            print(f"No transition from
{self.current_state.name} on input '{input_symbol}'")

state_a = State("A")
state_b = State("B")
state_c = State("C")

state_a.add_transition('0', state_b)
state_a.add_transition('1', state_c)
state_b.add_transition('0', state_a)
state_b.add_transition('1', state_c)
state_c.add_transition('0', state_a)
state_c.add_transition('1', state_b)

fsm = StateMachine(state_a)

inputs = ['0', '1', '1', '0', '1']
```

```python
for input_symbol in inputs:
    fsm.transition(input_symbol)
```

```
.../formal/lab/2025-01-24

(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
➜ python3 q1.py
Transitioning from A to B on input '0'
Transitioning from B to C on input '1'
Transitioning from C to B on input '1'
Transitioning from B to A on input '0'
Transitioning from A to C on input '1'
(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
➜ |
```

```
# 2. Design a Python program to verify simple Boolean
expressions using truth tables.
# ❷ Input a Boolean expression (e.g., (A and B) or (not
A)), and generate the truth table for all
# possible values of the variables.
# ❷ Compare the result against a user-provided expected
truth table to verify its correctness.

import itertools

def generate_truth_table(expression, variables):
    truth_table = []
    for values in itertools.product([False, True],
repeat=len(variables)):
        env = dict(zip(variables, values))
        result = eval(expression, {}, env)
        truth_table.append((values, result))
    return truth_table

def print_truth_table(truth_table, variables):
    header = variables + ["Result"]
    print("\t".join(header))
    for row in truth_table:
        values, result = row
        print("\t".join(map(str, values)) + "\t" +
str(result))

def verify_truth_table(expression, variables,
expected_truth_table):
    generated_truth_table =
generate_truth_table(expression, variables)
    return generated_truth_table ==
expected_truth_table

expression = input("Enter a Boolean expression (e.g.,
(A and B) or (not A)): ")
variables = input("Enter the variables in the
expression separated by spaces (e.g., A B): ").split()

truth_table = generate_truth_table(expression,
variables)
print("Generated Truth Table:")
print_truth_table(truth_table, variables)

expected_truth_table = []
```

```python
print("Enter the expected truth table (e.g., for A B
Result):")
for _ in range(2 ** len(variables)):
    row = input().split()
    values = tuple(map(lambda x: x == 'True',
row[:-1]))
    result = row[-1] == 'True'
    expected_truth_table.append((values, result))

is_correct = verify_truth_table(expression, variables,
expected_truth_table)
if is_correct:
    print("The generated truth table matches the
expected truth table.")
else:
    print("The generated truth table does not match the
expected truth table.")
```

```
.../formal/lab/2025-01-24

(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
 ➜ python3 q2.py
Enter a Boolean expression (e.g., (A and B) or (not A)): A or B
Enter the variables in the expression separated by spaces (e.g., A B): A B
Generated Truth Table:
A       B       Result
False   False   False
False   True    True
True    False   True
True    True    True
Enter the expected truth table (e.g., for A B Result):
False   False   False
False   True    True
True    False   True
True    True    True
The generated truth table matches the expected truth table.
(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
 ➜
```

```python
# 3. Implement a Python program to verify Linear Temporal Logic (LTL) formulas against a simple
# finite-state machine (FSM).

import networkx as nx

class Formula:
    def __init__(self, formula_str):
        self.formula_str = formula_str
        self.operands = []
        self.parse_formula(formula_str)

    def parse_formula(self, formula_str):
        if "&&" in formula_str:
            self.operands = formula_str.split("&&")
            self.type = "AND"
        elif "||" in formula_str:
            self.operands = formula_str.split("||")
            self.type = "OR"
        elif "!" in formula_str:
            self.operands = [formula_str[1:]]
            self.type = "NOT"
        elif "X" in formula_str:
            self.operands = [formula_str[1:]]
            self.type = "NEXT"
        elif "U" in formula_str:
            self.operands = formula_str.split("U")
            self.type = "UNTIL"
        else:
            self.type = "LITERAL"
            self.value = formula_str.strip()

    def is_literal(self):
        return self.type == "LITERAL"

    def is_and(self):
        return self.type == "AND"

    def is_or(self):
        return self.type == "OR"

    def is_not(self):
        return self.type == "NOT"

    def is_next(self):
```

```
        return self.type == "NEXT"

    def is_until(self):
        return self.type == "UNTIL"

class FSM:
    def __init__(self):
        self.graph = nx.DiGraph()
        self.initial_state = None

    def add_state(self, state, is_initial=False):
        self.graph.add_node(state)
        if is_initial:
            self.initial_state = state

    def add_transition(self, from_state, to_state,
label):
        self.graph.add_edge(from_state, to_state,
label=label)

    def get_transitions(self, state):
        return self.graph.out_edges(state, data=True)

def check_ltl_formula(fsm, formula):
    for state in fsm.graph.nodes:
        if not check_state(fsm, state, formula):
            return False
    return True

def check_state(fsm, state, formula):
    if formula.is_literal():
        return formula.value in fsm.graph.nodes[state]
    elif formula.is_and():
        return check_state(fsm, state,
Formula(formula.operands[0])) and check_state(fsm,
state,

Formula(formula.operands[1]))
    elif formula.is_or():
        return check_state(fsm, state,
Formula(formula.operands[0])) or check_state(fsm,
state,

Formula(formula.operands[1]))
    elif formula.is_not():
```

```
        return not check_state(fsm, state,
Formula(formula.operands[0]))
    elif formula.is_next():
        for _, next_state, data in
fsm.get_transitions(state):
            if check_state(fsm, next_state,
Formula(formula.operands[0])):
                return True
        return False
    elif formula.is_until():
        for _, next_state, data in
fsm.get_transitions(state):
            if check_state(fsm, next_state,
Formula(formula.operands[1])):
                return True
            if check_state(fsm, next_state,
Formula(formula.operands[0])) and check_state(fsm,
next_state, formula):
                return True
        return False
    return False

fsm = FSM()
fsm.add_state("S0", is_initial=True)
fsm.add_state("S1")
fsm.add_state("S2")
fsm.add_transition("S0", "S1", "a")
fsm.add_transition("S1", "S2", "b")
fsm.add_transition("S2", "S0", "c")

fsm.graph.nodes["S0"]["a"] = True
fsm.graph.nodes["S0"]["b"] = True

ltl_formula_nl = "a && b"
ltl_formula = Formula(ltl_formula_nl)

if check_ltl_formula(fsm, ltl_formula):
    print("The FSM satisfies the LTL formula.")
else:
    print("The FSM does not satisfy the LTL formula.")
```

```
.../formal/lab/2025-01-24
(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
 → fsm.add_state("S0", is_initial=True)
fsm.add_state("S1")
fsm.add_state("S2")
fsm.add_transition("S0", "S1", "a")
fsm.add_transition("S1", "S2", "b")
fsm.add_transition("S2", "S0", "c")
zsh: unknown file attribute: 0
(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
 → python3 q3.py
The FSM does not satisfy the LTL formula.
(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
 →
```

```python
# 4. Create a Python program to simulate a reactive
system for a traffic light controller with three
lights:
#     RED, YELLOW, and GREEN.

import time

class TrafficLight:
    def __init__(self):
        self.state = "RED"

    def transition(self):
        if self.state == "RED":
            self.state = "GREEN"
        elif self.state == "GREEN":
            self.state = "YELLOW"
        elif self.state == "YELLOW":
            self.state = "RED"

    def run(self, cycles=5):
        for _ in range(cycles):
            print(f"Light is {self.state}")
            if self.state == "RED":
```

```python
                time.sleep(5)
            elif self.state == "GREEN":
                time.sleep(5)
            elif self.state == "YELLOW":
                time.sleep(2)
            self.transition()

if __name__ == "__main__":
    traffic_light = TrafficLight()
    traffic_light.run()
```

```
.../formal/lab/2025-01-24
(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
 ➜ python3 q4.py
Light is RED
Light is GREEN
Light is YELLOW
Light is RED
Light is GREEN
(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
 ➜
```

```python
# 5. Write a Python program to simulate process
communication using the Communicating
# Sequential Processes (CSP) model.

import threading
import queue
import time

class CSPChannel:
    def __init__(self):
        self.channel = queue.Queue()

    def send(self, data):
        self.channel.put(data)

    def receive(self):
        return self.channel.get()

def process_a(channel_out, data):
    print("Process A: Sending data to Process B...")
    time.sleep(1)
    channel_out.send(data)
    print(f"Process A: Sent data '{data}' to Process
B.")


def process_b(channel_in, channel_out):
    print("Process B: Waiting to receive data from
Process A...")
    data = channel_in.receive()
    print(f"Process B: Received data '{data}' from
Process A.")
    time.sleep(1)
    response = f"{data} processed by B"
    print("Process B: Sending response to Process
C...")
    channel_out.send(response)

def process_c(channel_in):
    print("Process C: Waiting to receive response from
Process B...")
    response = channel_in.receive()
    print(f"Process C: Received response '{response}'
from Process B.")
```

```python
if __name__ == "__main__":
    channel_a_to_b = CSPChannel()
    channel_b_to_c = CSPChannel()

    thread_a = threading.Thread(target=process_a,
args=(channel_a_to_b, "Hello, B!"))
    thread_b = threading.Thread(target=process_b,
args=(channel_a_to_b, channel_b_to_c))
    thread_c = threading.Thread(target=process_c,
args=(channel_b_to_c,))

    thread_a.start()
    thread_b.start()
    thread_c.start()

    thread_a.join()
    thread_b.join()
    thread_c.join()

    print("All processes completed communication.")
```

```
.../formal/lab/2025-01-24
(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
➜ python3 q5.py
Process A: Sending data to Process B...
Process B: Waiting to receive data from Process A...
Process C: Waiting to receive response from Process B...
Process A: Sent data 'Hello, B!' to Process B.
Process B: Received data 'Hello, B!' from Process A.
Process B: Sending response to Process C...
Process C: Received response 'Hello, B! processed by B' from Process B.
All processes completed communication.
(base)  ~/desktop/cse/ASSGN/sem8/formal/lab/2025-01-24
➜
```