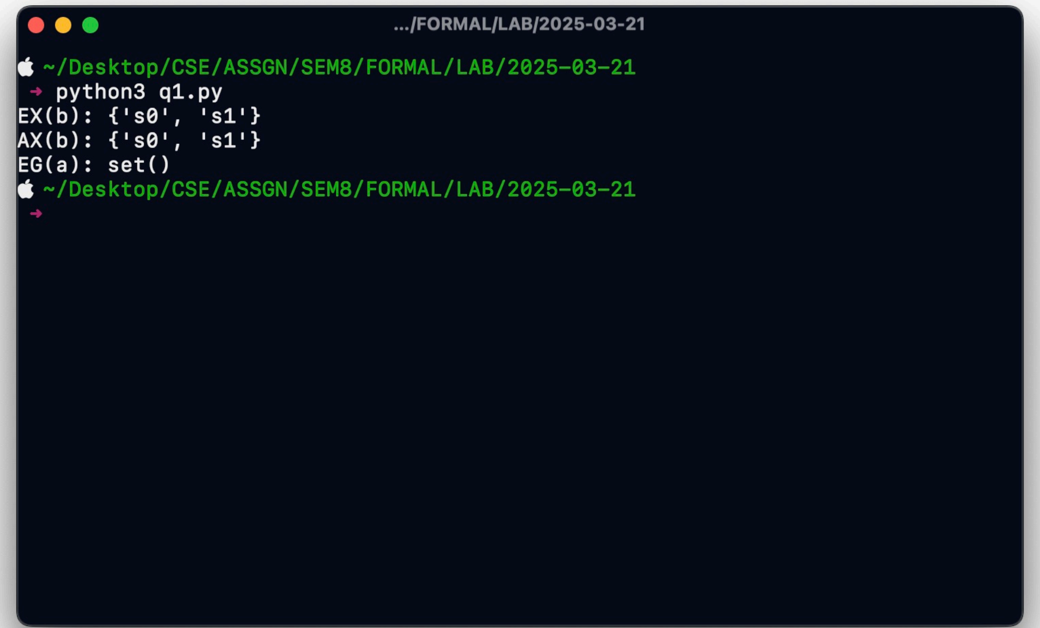
ASSIGNMENT 5
21/03/25

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

```
# Q1 Implement a Kripke Structure in Python and verify
Computation Tree Logic (CTL) properties.

class KripkeStructure:
    def __init__(self):
        self.states = set()
        self.transitions = {}
        self.labeling = {}

    def add_state(self, state, labels=set()):
        self.states.add(state)
        self.transitions[state] = set()
        self.labeling[state] = labels

    def add_transition(self, state_from, state_to):
        if state_from in self.states and state_to in
self.states:
            self.transitions[state_from].add(state_to)

    def satisfies(self, state, prop):
        return prop in self.labeling.get(state, set())


def EX(kripke, prop):
    result = set()
    for state in kripke.states:
        if any(kripke.satisfies(next_state, prop) for
next_state in kripke.transitions[state]):
            result.add(state)
    return result


def AX(kripke, prop):
    result = set()
    for state in kripke.states:
        if kripke.transitions[state] and all(
                kripke.satisfies(next_state, prop) for
next_state in kripke.transitions[state]):
            result.add(state)
    return result


def EG(kripke, prop):
    result = set()
    for state in kripke.states:
```

```python
        visited, stack = set(), [state]
        while stack:
            s = stack.pop()
            if s in visited:
                continue
            visited.add(s)
            if not kripke.satisfies(s, prop):
                break
            stack.extend(kripke.transitions[s])
        else:
            result.add(state)
    return result


def test_kripke_structure():
    ks = KripkeStructure()
    ks.add_state("s0", {"a"})
    ks.add_state("s1", {"b"})
    ks.add_state("s2", {"a", "b"})

    ks.add_transition("s0", "s1")
    ks.add_transition("s1", "s2")
    ks.add_transition("s2", "s0")

    print("EX(b):", EX(ks, "b"))
    print("AX(b):", AX(ks, "b"))
    print("EG(a):", EG(ks, "a"))


test_kripke_structure()
```



```
# Q2 Develop a Python-based Linear Temporal Logic (LTL)
model checker for verifying safety and liveness
properties.

import networkx as nx

class LTLModelChecker:
    def __init__(self, transition_system):
        self.transition_system = transition_system

    def check_safety(self, atomic_proposition):
        for state in self.transition_system.nodes():
            if atomic_proposition not in
self.transition_system.nodes[state]['labels']:
                return False, state
        return True, None

    def check_liveness(self, atomic_proposition):
        satisfying_states = [state for state in
self.transition_system.nodes() if
                             atomic_proposition in
self.transition_system.nodes[state]['labels']]
        return satisfying_states if satisfying_states
else None

ts = nx.DiGraph()
ts.add_nodes_from([
    ("s0", {"labels": {"p"}}),
    ("s1", {"labels": {"q"}}),
])
ts.add_edges_from([("s0", "s1"), ("s1", "s0")])

checker = LTLModelChecker(ts)

safety_result, safety_state = checker.check_safety("p")
liveness_result = checker.check_liveness("q")
print("Safety holds:" if safety_result else f"Safety
fails at {safety_state}")
print(f"Liveness holds in states: {liveness_result}" if
liveness_result else "Liveness fails")
```


```python
# Q3 Model a state transition system and check for deadlock freedom using model checking.

class StateTransitionSystem:
    def __init__(self):
        self.transitions = {}

    def add_state(self, state):
        if state not in self.transitions:
            self.transitions[state] = []

    def add_transition(self, from_state, to_state):
        if from_state not in self.transitions:
            self.add_state(from_state)
        if to_state not in self.transitions:
            self.add_state(to_state)
        self.transitions[from_state].append(to_state)

    def check_deadlock_freedom(self):
        deadlocks = [state for state in self.transitions if not self.transitions[state]]
        if deadlocks:
            print("Deadlock detected in states:", deadlocks)
            return False
        else:
            print("System is deadlock-free.")
```

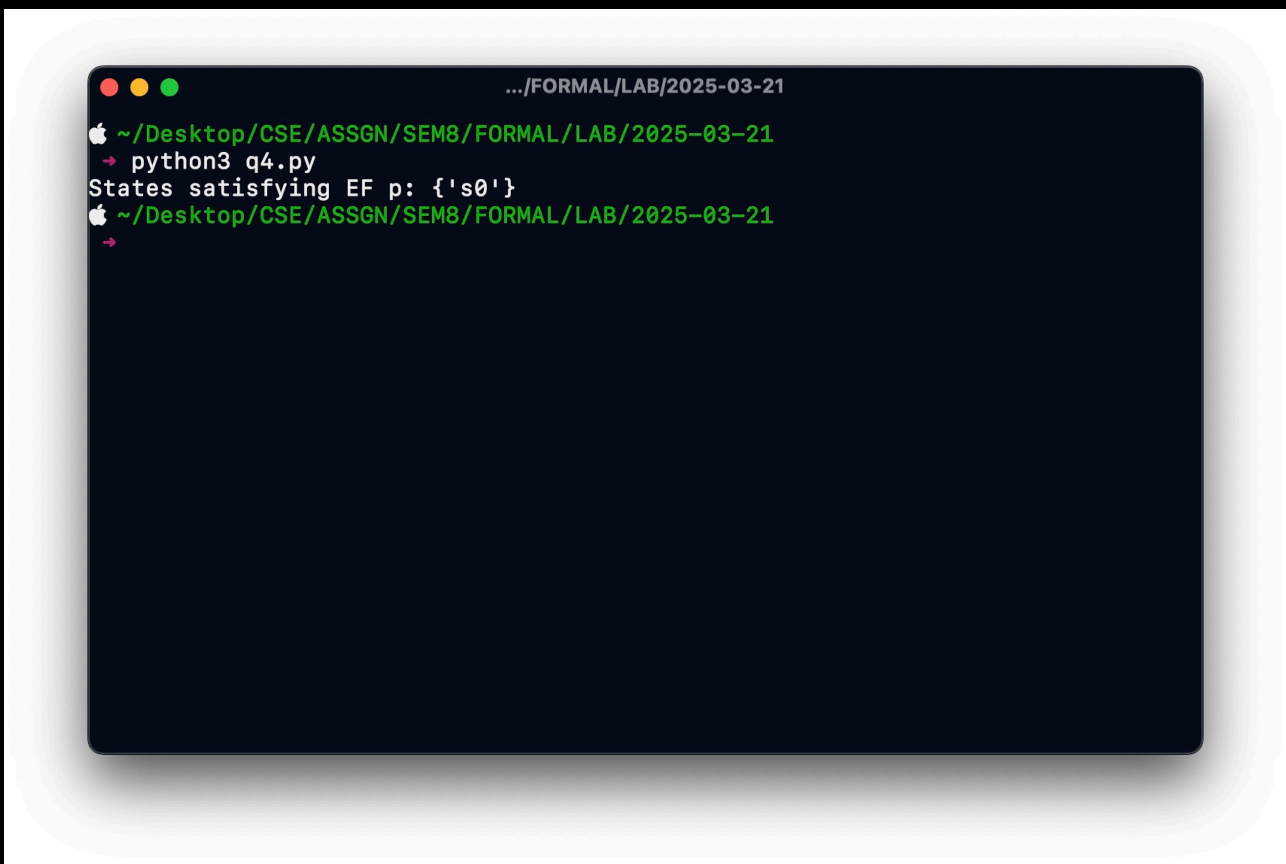
```python
                return True

sts = StateTransitionSystem()
sts.add_transition("S1", "S2")
sts.add_transition("S2", "S3")
sts.add_transition("S3", "S4")
sts.add_transition("S4", "S2")

sts.check_deadlock_freedom()
```


```python
# Q4 Implement a property verification tool using CTL
for a given transition system.

from itertools import product


class TransitionSystem:
    def __init__(self, states, transitions,
initial_states, atomic_props, labeling):
        self.states = states
        self.transitions = transitions
        self.initial_states = initial_states
        self.atomic_props = atomic_props
```

```
        self.labeling = labeling

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


class CTLModelChecker:
    def __init__(self, transition_system):
        self.ts = transition_system

    def check_property(self, formula):
        return self.evaluate(formula,
self.ts.initial_states)

    def evaluate(self, formula, states):
        if formula.startswith("EX"):
            subformula = formula[2:].strip()
            return self.ex(subformula, states)
        elif formula.startswith("EF"):
            subformula = formula[2:].strip()
            return self.ef(subformula, states)
        elif formula.startswith("EG"):
            subformula = formula[2:].strip()
            return self.eg(subformula, states)
        elif formula.startswith("AX"):
            subformula = formula[2:].strip()
            return self.ax(subformula, states)
        elif formula.startswith("AF"):
            subformula = formula[2:].strip()
            return self.af(subformula, states)
        elif formula.startswith("AG"):
            subformula = formula[2:].strip()
            return self.ag(subformula, states)
        else:
            return {s for s in states if formula in
self.ts.labeling.get(s, [])}

    def ex(self, formula, states):
        sat_states = self.evaluate(formula,
self.ts.states)
        return {s for s in states if any(succ in
sat_states for succ in self.ts.get_successors(s))}

    def ax(self, formula, states):
```

```
        sat_states = self.evaluate(formula,
self.ts.states)
        return {s for s in states if all(succ in
sat_states for succ in self.ts.get_successors(s))}

    def ef(self, formula, states):
        sat_states = set()
        stack = list(self.evaluate(formula,
self.ts.states))
        while stack:
            s = stack.pop()
            if s not in sat_states:
                sat_states.add(s)
                stack.extend([pred for pred in
self.ts.states if s in self.ts.get_successors(pred)])
        return sat_states.intersection(states)

    def af(self, formula, states):
        sat_states = self.evaluate(formula,
self.ts.states)
        unsat_states = set(self.ts.states) - sat_states
        result = set(states)
        while True:
            new_result = {s for s in result if all(succ
in result for succ in self.ts.get_successors(s))}
            if new_result == result:
                break
            result = new_result
        return result

    def eg(self, formula, states):
        sat_states = self.evaluate(formula,
self.ts.states)
        result = set()
        stack = [s for s in sat_states if all(succ in
sat_states for succ in self.ts.get_successors(s))]
        while stack:
            s = stack.pop()
            if s not in result:
                result.add(s)
                stack.extend(
                    [pred for pred in self.ts.states if
pred in sat_states and s in
self.ts.get_successors(pred)])
        return result.intersection(states)
```

```
    def ag(self, formula, states):
        return states.intersection(self.ts.states -
self.ef(f"!({formula})", self.ts.states))


if __name__ == "__main__":
    states = {"s0", "s1", "s2", "s3"}
    transitions = {"s0": ["s1", "s2"], "s1": ["s3"],
"s2": ["s3"], "s3": []}
    initial_states = {"s0"}
    atomic_props = {"p"}
    labeling = {"s0": set(), "s1": {"p"}, "s2": set(),
"s3": {"p"}}

    ts = TransitionSystem(states, transitions,
initial_states, atomic_props, labeling)
    checker = CTLModelChecker(ts)

    formula = "EF p"
    result = checker.check_property(formula)
    print(f"States satisfying {formula}: {result}")
```



```
# Q5 Verify fairness conditions in a concurrent system
using temporal logic.

class SystemModel:
    def __init__(self):
        self.states = []
        self.transitions = {}
        self.current_state = None

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

    def add_transition(self, from_state, to_state):
        if from_state in self.states and to_state in
self.states:

self.transitions[from_state].append(to_state)

    def set_initial_state(self, state):
        if state in self.states:
            self.current_state = state

    def get_reachable_states(self, state,
visited=None):
        if visited is None:
            visited = set()
        if state in visited:
            return visited
        visited.add(state)
        for next_state in self.transitions.get(state,
[]):
            self.get_reachable_states(next_state,
visited)
        return visited


class TemporalLogicChecker:
    @staticmethod
    def globally(system, condition):
        visited =
system.get_reachable_states(system.current_state)
        return all(condition(state) for state in
visited)
```

```
    @staticmethod
    def eventually(system, condition):
        visited =
system.get_reachable_states(system.current_state)
        return any(condition(state) for state in
visited)

    @staticmethod
    def fairness_condition(system, request_condition,
grant_condition):
        visited =
system.get_reachable_states(system.current_state)
        for state in visited:
            if request_condition(state):
                if not any(grant_condition(next_state)
for next_state in visited):
                    return False
        return True


system = SystemModel()

system.add_state("idle")
system.add_state("request")
system.add_state("critical")

system.add_transition("idle", "request")
system.add_transition("request", "critical")
system.add_transition("critical", "idle")

system.set_initial_state("idle")

request_condition = lambda state: state == "request"
grant_condition = lambda state: state == "critical"
fairness_result =
TemporalLogicChecker.fairness_condition(system,
request_condition, grant_condition)

print("Fairness Condition Holds?" , fairness_result)
```

```
.../FORMAL/LAB/2025-03-21

 ~/Desktop/CSE/ASSGN/SEM8/FORMAL/LAB/2025-03-21
 ➜ python3 q5.py
Fairness Condition Holds? True
 ~/Desktop/CSE/ASSGN/SEM8/FORMAL/LAB/2025-03-21
 ➜
```