**Assignment - 5**

**Formal Methods Lab**

1. Kripke Structure and CTL Verification

```python

class KripkeStructure:

def \_\_init\_\_(self, states, transitions, labeling, initial\_states):

"""

states: set of states

transitions: dict mapping states to sets of successor states

labeling: dict mapping states to sets of atomic propositions

initial\_states: set of initial states

"""

self.states = states

self.transitions = transitions

self.labeling = labeling

self.initial\_states = initial\_states

def satisfies\_CTL(self, formula, state):

"""Recursive CTL model checking"""

if formula == 'True':

return True

elif formula.startswith('~'):

return not self.satisfies\_CTL(formula[1:], state)

elif formula.startswith('EX'):

phi = formula[2:]

return any(self.satisfies\_CTL(phi, s) for s in self.transitions.get(state, set()))

elif formula.startswith('EF'):

phi = formula[2:]

visited = set()

stack = [state]

while stack:

current = stack.pop()

if current in visited:

continue

visited.add(current)

if self.satisfies\_CTL(phi, current):

return True

stack.extend(self.transitions.get(current, set()))

return False

elif formula.startswith('EG'):

phi = formula[2:]

# Find maximal states satisfying phi where EG phi holds

sccs = self.find\_SCCs(lambda s: self.satisfies\_CTL(phi, s))

reachable\_sccs = set()

stack = [state]

visited = set()

while stack:

current = stack.pop()

if current in visited:

continue

visited.add(current)

for scc in sccs:

if current in scc and len(scc) > 1:

return True

stack.extend(self.transitions.get(current, set()))

return False

elif formula.startswith('E['):

# EU operator: E[phi U psi]

parts = formula[2:-1].split(' U ')

phi, psi = parts[0], parts[1]

visited = set()

stack = [(state, False)]

while stack:

current, psi\_holds = stack.pop()

if current in visited:

continue

visited.add(current)

if self.satisfies\_CTL(psi, current):

return True

if not self.satisfies\_CTL(phi, current):

continue

stack.extend((s, False) for s in self.transitions.get(current, set()))

return False

else:

# Atomic proposition

return formula in self.labeling.get(state, set())

def find\_SCCs(self, condition):

"""Find strongly connected components where condition holds"""

# Kosaraju's algorithm for SCCs

visited = set()

order = []

def visit(node):

if node not in visited and condition(node):

visited.add(node)

for neighbor in self.transitions.get(node, set()):

if condition(neighbor):

visit(neighbor)

order.append(node)

for node in self.states:

visit(node)

visited = set()

sccs = []

reverse\_transitions = {s: set() for s in self.states}

for s in self.transitions:

for t in self.transitions[s]:

reverse\_transitions[t].add(s)

for node in reversed(order):

if node not in visited and condition(node):

scc = set()

stack = [node]

while stack:

current = stack.pop()

if current in visited:

continue

visited.add(current)

scc.add(current)

for neighbor in reverse\_transitions.get(current, set()):

if condition(neighbor):

stack.append(neighbor)

sccs.append(scc)

return sccs

# Example usage

states = {'s0', 's1', 's2'}

transitions = {'s0': {'s1'}, 's1': {'s2'}, 's2': {'s0', 's1'}}

labeling = {'s0': {'p'}, 's1': {'q'}, 's2': {'p', 'q'}}

initial\_states = {'s0'}

ks = KripkeStructure(states, transitions, labeling, initial\_states)

# Check some CTL formulas

print(ks.satisfies\_CTL('EX q', 's0')) # True (s0->s1 and s1 has q)

print(ks.satisfies\_CTL('EG p', 's0')) # False (no infinite path where p always holds)

print(ks.satisfies\_CTL('E[p U q]', 's0')) # True (s0->s1 and s1 has q)

```

## 2. LTL Model Checker

```python

class LTLChecker:

def \_\_init\_\_(self, transitions, labeling, initial\_states):

"""

transitions: dict mapping states to sets of successor states

labeling: dict mapping states to sets of atomic propositions

initial\_states: set of initial states

"""

self.transitions = transitions

self.labeling = labeling

self.initial\_states = initial\_states

def check\_LTL(self, formula):

"""Check if the formula holds for all paths from initial states"""

# Convert LTL formula to Buchi automaton (simplified)

buchi = self.ltl\_to\_buchi(formula)

# Check emptiness of product automaton

return not self.has\_accepting\_cycle(buchi)

def ltl\_to\_buchi(self, formula):

"""Simplified LTL to Buchi automaton conversion"""

# In a real implementation, use an algorithm like LTL2BA

# Here we just handle some simple cases for demonstration

if formula == 'G p': # Always p

return {

'states': {'q0'},

'initial': {'q0'},

'transitions': {

('q0', 'p'): {'q0'},

},

'accepting': {'q0'}

}

elif formula == 'F p': # Eventually p

return {

'states': {'q0', 'q1'},

'initial': {'q0'},

'transitions': {

('q0', 'p'): {'q1'},

('q0', '~p'): {'q0'},

('q1', 'p'): {'q1'},

('q1', '~p'): {'q1'},

},

'accepting': {'q1'}

}

elif formula == 'p U q': # p until q

return {

'states': {'q0', 'q1'},

'initial': {'q0'},

'transitions': {

('q0', 'p & ~q'): {'q0'},

('q0', 'q'): {'q1'},

('q1', 'True'): {'q1'},

},

'accepting': {'q1'}

}

else:

raise ValueError(f"Unsupported LTL formula: {formula}")

def has\_accepting\_cycle(self, buchi):

"""Check if the product automaton has an accepting cycle"""

# Simplified implementation - in practice use nested DFS

# Here we just check if any accepting state is reachable from itself

for state in buchi['accepting']:

visited = set()

stack = [state]

while stack:

current = stack.pop()

if current == state and len(visited) > 0:

return True

if current in visited:

continue

visited.add(current)

# Get all transitions from current state

for (s, cond), dests in buchi['transitions'].items():

if s == current:

# Check if condition is satisfied by any system state

for sys\_state in self.transitions:

if self.eval\_condition(cond, sys\_state):

stack.extend(dests)

return False

def eval\_condition(self, cond, state):

"""Evaluate a Buchi transition condition on a system state"""

if cond == 'True':

return True

elif cond == 'p':

return 'p' in self.labeling.get(state, set())

elif cond == '~p':

return 'p' not in self.labeling.get(state, set())

elif cond == 'p & ~q':

return ('p' in self.labeling.get(state, set())) and ('q' not in self.labeling.get(state, set()))

else:

raise ValueError(f"Unsupported condition: {cond}")

# Example usage

transitions = {'s0': {'s1'}, 's1': {'s2'}, 's2': {'s0', 's1'}}

labeling = {'s0': {'p'}, 's1': {'q'}, 's2': {'p', 'q'}}

initial\_states = {'s0'}

checker = LTLChecker(transitions, labeling, initial\_states)

print(checker.check\_LTL('G p')) # False (p doesn't hold in s1)

print(checker.check\_LTL('F q')) # True (q holds in s1)

```

## 3. Deadlock Freedom Checker

```python

class DeadlockChecker:

def \_\_init\_\_(self, transitions, initial\_states):

"""

transitions: dict mapping states to sets of successor states

initial\_states: set of initial states

"""

self.transitions = transitions

self.initial\_states = initial\_states

def is\_deadlock\_free(self):

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

visited = set()

stack = list(self.initial\_states)

while stack:

current = stack.pop()

if current in visited:

continue

visited.add(current)

# Check if this state has no transitions

if not self.transitions.get(current, set()):

return False

stack.extend(self.transitions.get(current, set()))

return True

# Example usage

transitions = {

's0': {'s1'},

's1': {'s2'},

's2': {'s0', 's1'}

}

initial\_states = {'s0'}

checker = DeadlockChecker(transitions, initial\_states)

print(checker.is\_deadlock\_free()) # True

# System with deadlock

transitions\_deadlock = {

's0': {'s1'},

's1': {'s2'},

's2': set() # Deadlock state

}

checker\_deadlock = DeadlockChecker(transitions\_deadlock, {'s0'})

print(checker\_deadlock.is\_deadlock\_free()) # False

```

## 4. CTL Property Verification Tool

```python

class CTLVerifier:

def \_\_init\_\_(self, transitions, labeling, initial\_states):

self.transitions = transitions

self.labeling = labeling

self.initial\_states = initial\_states

def verify(self, formula):

"""Verify if the CTL formula holds in all initial states"""

results = {}

for state in self.initial\_states:

results[state] = self.check\_state(formula, state)

return results

def check\_state(self, formula, state):

"""Recursive CTL checking for a single state"""

if formula == 'True':

return True

elif formula.startswith('~'):

return not self.check\_state(formula[1:], state)

elif formula.startswith('EX'):

phi = formula[2:]

return any(self.check\_state(phi, s) for s in self.transitions.get(state, set()))

elif formula.startswith('AX'):

phi = formula[2:]

successors = self.transitions.get(state, set())

if not successors:

return True # Vacuous truth

return all(self.check\_state(phi, s) for s in successors)

elif formula.startswith('EF'):

phi = formula[2:]

visited = set()

stack = [state]

while stack:

current = stack.pop()

if current in visited:

continue

visited.add(current)

if self.check\_state(phi, current):

return True

stack.extend(self.transitions.get(current, set()))

return False

elif formula.startswith('AG'):

phi = formula[2:]

visited = set()

stack = [state]

result = True

while stack:

current = stack.pop()

if current in visited:

continue

visited.add(current)

if not self.check\_state(phi, current):

result = False

break

stack.extend(self.transitions.get(current, set()))

return result

elif formula.startswith('E['):

# EU operator

parts = formula[2:-1].split(' U ')

phi, psi = parts[0], parts[1]

visited = set()

stack = [(state, False)]

while stack:

current, psi\_holds = stack.pop()

if current in visited:

continue

visited.add(current)

if self.check\_state(psi, current):

return True

if not self.check\_state(phi, current):

continue

stack.extend((s, False) for s in self.transitions.get(current, set()))

return False

else:

# Atomic proposition

return formula in self.labeling.get(state, set())

# Example usage

transitions = {

's0': {'s1'},

's1': {'s2'},

's2': {'s0', 's1'}

}

labeling = {

's0': {'start'},

's1': {'middle'},

's2': {'end'}

}

initial\_states = {'s0'}

verifier = CTLVerifier(transitions, labeling, initial\_states)

print(verifier.verify('EX middle')) # {'s0': True}

print(verifier.verify('AG EF start')) # {'s0': True} (always possible to return to start)

print(verifier.verify('E[start U end]')) # {'s0': True} (path s0->s1->s2)

```

## 5. Fairness Condition Verification

```python

class FairnessChecker:

def \_\_init\_\_(self, transitions, labeling, initial\_states, fairness\_conditions):

"""

transitions: dict mapping states to sets of successor states

labeling: dict mapping states to sets of atomic propositions

initial\_states: set of initial states

fairness\_conditions: list of LTL formulas representing fairness conditions

"""

self.transitions = transitions

self.labeling = labeling

self.initial\_states = initial\_states

self.fairness = fairness\_conditions

def verify\_fairness(self):

"""Verify if the system satisfies all fairness conditions"""

results = {}

for condition in self.fairness:

results[condition] = self.check\_fairness\_condition(condition)

return results

def check\_fairness\_condition(self, condition):

"""Check a single fairness condition"""

# Convert to Buchi automaton

buchi = self.ltl\_to\_buchi(condition)

# Check if there exists a fair path (one that satisfies the condition infinitely often)

# Here we use a simplified approach - check if any state satisfying the condition

# is reachable from itself (cycle)

accepting\_states = buchi['accepting']

for state in accepting\_states:

visited = set()

stack = [state]

while stack:

current = stack.pop()

if current == state and len(visited) > 0:

return True

if current in visited:

continue

visited.add(current)

for (s, cond), dests in buchi['transitions'].items():

if s == current:

for sys\_state in self.transitions:

if self.eval\_condition(cond, sys\_state):

stack.extend(dests)

return False

def ltl\_to\_buchi(self, formula):

"""Simplified LTL to Buchi conversion for fairness conditions"""

if formula == 'GF p': # Infinitely often p

return {

'states': {'q0', 'q1'},

'initial': {'q0'},

'transitions': {

('q0', 'p'): {'q1'},

('q0', '~p'): {'q0'},

('q1', 'p'): {'q1'},

('q1', '~p'): {'q0'},

},

'accepting': {'q1'}

}

elif formula == 'FG p': # Eventually always p

return {

'states': {'q0', 'q1'},

'initial': {'q0'},

'transitions': {

('q0', 'p'): {'q1'},

('q0', '~p'): {'q0'},

('q1', 'p'): {'q1'},

('q1', '~p'): {'q0'},

},

'accepting': {'q1'}

}

else:

raise ValueError(f"Unsupported fairness condition: {formula}")

def eval\_condition(self, cond, state):

"""Evaluate a condition on a system state"""

if cond == 'p':

return 'p' in self.labeling.get(state, set())

elif cond == '~p':

return 'p' not in self.labeling.get(state, set())

else:

raise ValueError(f"Unsupported condition: {cond}")

# Example usage

transitions = {

's0': {'s1'},

's1': {'s2'},

's2': {'s0', 's1'}

}

labeling = {

's0': {'p'},

's1': {'q'},

's2': {'p'}

}

initial\_states = {'s0'}

fairness\_conditions = ['GF p', 'GF q']

checker = FairnessChecker(transitions, labeling, initial\_states, fairness\_conditions)

results = checker.verify\_fairness()

print(results) # {'GF p': True, 'GF q': False} (p occurs infinitely often, q doesn't)

```