**Formal Methods Lab**

**Assignment 9**

1. Model Checking for Topological Sorting Correctness

Objective: Verify that a topological sort algorithm produces a valid linear ordering where every directed edge \( u \rightarrow v \) has \( u \) before \( v \) in the ordering.

Steps:

1. Model the Algorithm (e.g., Kahn's or DFS-based) as a finite-state transition system.

2. Formalize Correctness Property in Temporal Logic (LTL/CTL):

- Acyclicity: `AG(¬(edge ∧ visited[u] ∧ visited[v] ∧ pos[u] > pos[v]))`

(No edge violates the ordering.)

- Completeness: `AF(all\_nodes\_visited)`

(All nodes are eventually included.)

3. Model Checker Setup:

- Use NuSMV or SPIN to encode the graph and algorithm.

- Example (NuSMV):

```smv

MODULE main

VAR

visited: array 1..n of boolean;

pos: array 1..n of 0..n;

-- Transition: Select next node with zero in-degree

...

SPEC AG (FORALL (u, v): !(edge[u][v] & visited[u] & visited[v] & pos[u] > pos[v]))

```

4. Result: If the model checker finds no violations, the algorithm is correct for the input graph class.

---

2. Hoare Logic Proof for Binary Search

Objective: Prove partial/total correctness of binary search for a sorted array.

Precondition: `sorted(A) ∧ ∃i ∈ [0..n-1], A[i] = key`

Postcondition: `A[low] = key ∧ ∀k < low, A[k] ≠ key`

Loop Invariant (critical step):

```

sorted(A) ∧ (A[low] ≤ key ≤ A[high]) ∧ (key ∈ A ⇒ key ∈ A[low..high])

```

Proof Outline:

1. Initialization: `low = 0, high = n-1` satisfies the invariant.

2. Maintenance: For midpoint `mid`, adjust `low`/`high` preserving the invariant.

- `A[mid] < key ⇒ new\_low = mid+1` (still `A[new\_low] ≤ key`).

- `A[mid] ≥ key ⇒ new\_high = mid` (preserves `key ≤ A[new\_high]`).

3. Termination: When `low ≥ high`, the loop exits. By the invariant, `A[low] = key`.

Hoare Triple:

```

{sorted(A) ∧ ∃i, A[i] = key}

while low < high:

mid := (low + high) / 2

if A[mid] < key: low := mid + 1

else: high := mid

{A[low] = key ∧ ∀k < low, A[k] ≠ key}

```

---

3. Formal Verification of Loop Invariants for Fixed-Point Algorithms

Example: Verify convergence of PageRank’s power iteration.

Loop Invariant:

```

∀v, rank[v] ≥ 0 ∧ Σ rank[v] = 1 ∧ error = |rank\_new - rank\_old| < ε

```

Verification Steps:

1. Encode in Dafny/Coq:

```coq

Fixpoint pageRank (ranks: list R) (iter: nat): list R :=

if iter = 0 then ranks

else update\_ranks (pageRank ranks (iter-1)).

Lemma convergence: forall ε, exists n,

after\_n\_steps n → error < ε.

```

2. Prove:

- Monotonicity: Show `error` decreases with each iteration.

- Termination: Bound `error` using geometric series convergence.

---

4. Best Formal Method for Concurrent Access Control

Choice: Process Algebra (CSP/π-Calculus)

Justification:

- Expressive concurrency primitives (parallel composition, synchronization).

- Deadlock/livelock detection via equivalence checking (e.g., trace refinement).

- Tool Support: FDR4 for CSP (finite-state checks) or ProB for model checking.

Example (CSP):

- Model a reader-writer lock:

```csp

READER = read -> READER [] release -> STOP.

WRITER = write -> release -> STOP.

LOCK = (acquire -> release -> LOCK) [] (acquire -> WRITER).

SYSTEM = (READER ||| WRITER) [|{acquire,release}|] LOCK.

```

- Verification in FDR4:

- Check `SYSTEM` is deadlock-free.

- Assert `WRITER` requires exclusive access:

`assert SYSTEM :[deadlock free]`.

---

5. Formal Specification for Job Scheduling

System: Priority-based scheduler with preemption.

TLA+ Specification:

```tla

EXTENDS Integers, Sequences

VARIABLE queue, running, completed

Init == queue = <<>> ∧ running = {} ∧ completed = {}

Schedule(job) ==

LET highest = CHOOSE j ∈ queue: ∀k ∈ queue: j.prio ≥ k.prio

IN running' = running ∪ {highest} ∧ queue' = queue \ {highest}

Complete(job) ==

running' = running \ {job} ∧ completed' = completed ∪ {job}

Next == ∃j ∈ Jobs: Schedule(j) ∨ Complete(j)

Fairness == ∀j ∈ Jobs: WF\_⟨Schedule(j)⟩

Spec == Init ∧ □[Next]\_⟨queue, running, completed⟩ ∧ Fairness

```

Verification:

1. Invariant: `∀j ∈ running, ¬∃k ∈ queue: k.prio > j.prio` (no priority inversion).

2. Liveness: `∀j ∈ Jobs, ◇(j ∈ completed)` (all jobs eventually complete).

3. Model Check with TLC for small instances.

---

Each task demonstrates how formal methods can rigorously verify algorithmic and system properties. The choice of method depends on the problem domain (concurrency vs. sequential algorithms) and available tooling.