USI-COP-SEED

```
from enum import Enum, auto
import time
import uuid
```

---

1. COP LATTICE ELEMENTS

```
class Coherence(Enum):
STABLE = auto()
DRIFTING = auto()
COLLAPSED = auto()

class Boundary(Enum):
IMPLICIT = auto()
EXPLICIT = auto()
PERMEABLE = auto()
FAILING = auto()

class Relation(Enum):
ALIGNED = auto()
ORTHOGONAL = auto()
COUPLED = auto()
OPPOSED = auto()

class Intent(Enum):
I1_CLEAR = auto()
I2_SPLIT = auto()
I3_DEGRADING = auto()
I4_DEGRADED = auto()

class TransitionType(Enum):
STABILIZE = auto()
REFRAME = auto()
CONSTRUCT = auto()
INQUIRE = auto()
FORM_BOUNDARY = auto()
```

WARNING = auto()
NONE = auto()

---

2. COP STATES (S1–S6)

---

class COPState(Enum):
S1_INQUIRY = auto()
S2_CONSTRUCTION = auto()
S3_STABILIZING_DRIFT = auto()
S4_BOUNDARY_FORMATION = auto()
S5_PRE_COLLAPSE = auto()
S6_COHERENT_REFRAME = auto()

---

3. COP INVARIANTS

---

class InvariantStatus(Enum):
OK = auto()
WARN = auto()
FAIL = auto()

class COPInvariants:
def **init**(self):
self.no_construction_under_degraded_intent = InvariantStatus.OK
self.reframe_before_collapse = InvariantStatus.OK
self.no_coupling_increase_in_S5 = InvariantStatus.OK
self.boundary_required_for_construction = InvariantStatus.OK
self.inquiry_must_not_degrade_coherence = InvariantStatus.OK
self.drift_triggers_stabilization_or_reframe = InvariantStatus.OK
self.intent_must_be_classified = InvariantStatus.OK

```
def as_vector(self):
    return {
        "no_construction_under_degraded_intent":
self.no_construction_under_degraded_intent,
```

```
        "reframe_before_collapse": self.reframe_before_collapse,
        "no_coupling_increase_in_S5": self.no_coupling_increase_in_S5,
        "boundary_required_for_construction":
self.boundary_required_for_construction,
        "inquiry_must_not_degrade_coherence":
self.inquiry_must_not_degrade_coherence,
        "drift_triggers_stabilization_or_reframe":
self.drift_triggers_stabilization_or_reframe,
        "intent_must_be_classified": self.intent_must_be_classified,
    }

def any_fail(self):
    return any(v == InvariantStatus.FAIL for v in self.as_vector().values())
```

---

4. COP EXPRESSION (COP-Expr)

---

class COPExpr:
"""
COP-Expr consists of:
S: current state
P: active primitives
O: operator applied
I: invariant status vector
TT: transition-type
"""
def **init**(self, state, operator_name, transition_type, invariants: COPInvariants, primitives=None):
self.state = state
self.operator_name = operator_name
self.transition_type = transition_type
self.invariants = invariants.as_vector()
self.primitives = primitives or []
self.timestamp = time.time()

```
def __repr__(self):
    return f"<COPExpr state={self.state.name} op={self.operator_name} tt=
{self.transition_type.name}>"
```

---

5. USI SIDE: BINDING, RESOLUTION, UNIT, SEQ

```
class BindingType(Enum):
ALIGNED = auto()
ORTHOGONAL = auto()
COUPLED = auto()
DECOUPLED = auto()
OPPOSED = auto() # disallowed in valid bindings

class ResolutionLevel(Enum):
PRIMITIVE = 1
STRUCTURAL = 2
RELATIONAL = 3
GENERATIVE = 4
META = 5

class USIUnit:
"""
A USI-Unit consists of:
- a COP-Expr
- a symbolic anchor
- a binding vector
- a resolution signature
- an interface marker
"""
def init(self, cop_expr: COPExpr, anchor: str,
binding_vector=None,
resolution_level: ResolutionLevel = ResolutionLevel.PRIMITIVE,
interface_marker: str = "local"):
self.id = uuid.uuid4()
self.cop_expr = cop_expr
self.anchor = anchor
self.binding_vector = binding_vector or [] # list[BindingType]
self.resolution_level = resolution_level
self.interface_marker = interface_marker
```

```
    def __repr__(self):
        return f"<USIUnit id={self.id} anchor={self.anchor} state=
    {self.cop_expr.state.name}>"
```

class USISequence:
"""
USI-Seq: Seq = [U1, U2, ... , Un]
Valid if:
- all COP invariants are satisfied
- no binding violates coherence (no OPPOSED)
- no resolution level collapses (non-decreasing)
"""
def **init**(self, units=None):
self.units = units or []

```
    def append(self, unit: USIUnit):
        self.units.append(unit)

    def is_valid(self):
        if not self.units:
            return True

        # 1. COP invariants: no FAIL in any embedded COP-Expr
        for u in self.units:
            if any(status == InvariantStatus.FAIL for status in
    u.cop_expr.invariants.values()):
                return False

        # 2. Binding coherence: no opposed bindings
        for u in self.units:
            for bt in u.binding_vector:
                if bt == BindingType.OPPOSED:
                    return False

        # 3. Resolution stability: non-decreasing resolution along the sequence
        prev_level = self.units[0].resolution_level.value
        for u in self.units[1:]:
            if u.resolution_level.value < prev_level:
                return False
            prev_level = u.resolution_level.value

        return True
```

---

6. COP OPERATORS (MINIMAL SUBSET)

---

class COPOperator:
"""
Operators are the only mechanism by which COP transitions between states
or modifies lattice configurations.
"""
def **init**(self, name, fn, transition_type: TransitionType):
self.name = name
self.fn = fn # fn(membrane) -> new_state
self.transition_type = transition_type

```
def apply(self, membrane):
    return self.fn(membrane)
```

def op_boundary_tighten(membrane):
# Move toward explicit boundary; keep state
membrane.boundary = Boundary.EXPLICIT
return membrane.cop_state

def op_coherence_stabilize(membrane):
# S3 -> S6, otherwise keep state
if membrane.cop_state == COPState.S3_STABILIZING_DRIFT:
return COPState.S6_COHERENT_REFRAME
return membrane.cop_state

---

7. MEMBRANE + COP SEQUENCE VALIDATION

---

class Membrane:
"""
Encapsulates:
- COP state + lattice slice
- invariants
- USI-Units history (symbolic manifold fragment)

"""
def **init**(self):
self.cop_state = COPState.S1_INQUIRY
self.coherence = Coherence.STABLE
self.boundary = Boundary.EXPLICIT
self.relation = Relation.ALIGNED
self.intent = Intent.I1_CLEAR
self.invariants = COPInvariants()
self.history: list[USIUnit] = []

```
# --- Invariant checks (minimal) ---

def check_invariants_before(self, operator: COPOperator):
    # Intent must always be classified
    if self.intent not in Intent:
        self.invariants.intent_must_be_classified = InvariantStatus.FAIL

    # No construction under degraded intent
    if operator.transition_type == TransitionType.CONSTRUCT and \
       self.intent in (Intent.I3_DEGRADING, Intent.I4_DEGRADED):
        self.invariants.no_construction_under_degraded_intent =
InvariantStatus.FAIL

    return not self.invariants.any_fail()

# --- Local validity between adjacent COP-Exprs (Section 9) ---

@staticmethod
def next_state_allowed(prev_state: COPState, tt: TransitionType, next_state:
COPState) -> bool:
    if tt == TransitionType.STABILIZE:
        return next_state in (COPState.S3_STABILIZING_DRIFT,
COPState.S6_COHERENT_REFRAME)
    if tt == TransitionType.REFRAME:
        return next_state == COPState.S6_COHERENT_REFRAME
    if tt == TransitionType.CONSTRUCT:
        return next_state == COPState.S2_CONSTRUCTION
    if tt == TransitionType.INQUIRE:
        return next_state == COPState.S1_INQUIRY
    if tt == TransitionType.FORM_BOUNDARY:
        return next_state in (COPState.S4_BOUNDARY_FORMATION,
                              COPState.S1_INQUIRY,
                              COPState.S2_CONSTRUCTION)
    if tt == TransitionType.WARNING:
        return next_state == COPState.S5_PRE_COLLAPSE
```

```
    if tt == TransitionType.NONE:
        return next_state == prev_state
    return False

# --- Global COP sequence validity (Section 10) ---

@staticmethod
def cop_sequence_valid(units: list[USIUnit]) -> bool:
    if not units:
        return True

    exprs = [u.cop_expr for u in units]

    # 1. No invariant remains in FAIL
    for e in exprs:
        if any(status == InvariantStatus.FAIL for status in
e.invariants.values()):
            return False

    # 2. S5 is always followed by S6
    for i, e in enumerate(exprs[:-1]):
        if e.state == COPState.S5_PRE_COLLAPSE:
            if exprs[i+1].state != COPState.S6_COHERENT_REFRAME:
                return False

    # 3. Drift is always followed by stabilization or reframe
    for i, e in enumerate(exprs[:-1]):
        if e.state == COPState.S3_STABILIZING_DRIFT:
            tt_next = exprs[i+1].transition_type
            if tt_next not in (TransitionType.STABILIZE,
TransitionType.REFRAME):
                return False

    # 4. No construction under I3 or I4
    # (intent is enforced pre-step in this minimal seed)

    return True

# --- Single step ---

def step(self, operator: COPOperator, anchor: str = "minimal-anchor"):
    if not self.check_invariants_before(operator):
        raise ValueError("Invariant violated before operator application")

    prev_state = self.cop_state
    new_state = operator.apply(self)
```

```
    if not self.next_state_allowed(prev_state, operator.transition_type,
new_state):
        raise ValueError(f"Invalid state transition: {prev_state} --
{operator.transition_type}--> {new_state}")

    self.cop_state = new_state

    cop_expr = COPExpr(
        state=self.cop_state,
        operator_name=operator.name,
        transition_type=operator.transition_type,
        invariants=self.invariants,
        primitives=[]
    )

    # Minimal binding: treat each new unit as ALIGNED with prior manifold
    binding_vector = [BindingType.ALIGNED] if self.history else []

    usi_unit = USIUnit(
        cop_expr=cop_expr,
        anchor=anchor,
        binding_vector=binding_vector,
        resolution_level=ResolutionLevel.PRIMITIVE,
        interface_marker="local"
    )

    self.history.append(usi_unit)
    return usi_unit
```

---

8. MINIMAL GENERATIVE LOOP + VALIDATION

---

if **name** == "**main**":
membrane = Membrane()

```
ops = [
    COPOperator("Boundary-Tighten", op_boundary_tighten,
TransitionType.FORM_BOUNDARY),
    COPOperator("Coherence-Stabilize", op_coherence_stabilize,
TransitionType.STABILIZE),
]
```

```
usi_seq = USISequence()

for i in range(5):
    op = ops[i % len(ops)]
    unit = membrane.step(op, anchor=f"step-{i}")
    usi_seq.append(unit)
    print(unit)

print("USI-Seq valid:", usi_seq.is_valid())
print("COP sequence globally valid:",
Membrane.cop_sequence_valid(membrane.history))
```