```
##############################################################
# UE (Universo Emergente) — PSEUDOCÓDIGO REVISADO (COMPLETO)
# Objetivo: construir dominios como (ℛ + candados + cuencas metastables)
#           y validarlos con candados anti-posdición.
##############################################################

##############################
# 0) TIPOS Y CONTRATOS
##############################

type Data:
   times: List[t]
   nodes: List[i]          # opcional (si no hay grafo, nodes=[0])
   graph: optional adjacency / distances
   channels: Dict[String, Array]  # señales base

   function slice(t, i) -> Dict:   # retorna canales/medidas de un nodo i en tiempo t
   function global_slice(t) -> Dict: # si no hay nodos

type MicroState ψ:
   x: Vector           # features en R^d (representación)
   t: t
   i: i
   label_R: Bool       # True si t está dentro de una ventana ℛ
   window_id: Int

type ERSCondition ℛ:
   name: String
   predicate: function(Data, t) -> Bool
   persistence_tau: Integer     # τ_p
   windows_R: List[TimeWindow]  # True-segments (>= τ_p)
   windows_notR: List[TimeWindow]  # ventanas de control emparejadas (same-length /
seasonality)

type Lock:
   name: String
   predicate_on_x: function(Vector) -> Bool
   support_R: Float        # P(lock|ℛ)
   support_notR: Float     # P(lock|¬ℛ)
   specificity: Float      # contraste (p.ej. log-odds o ratio)
   tau_persist_R: Float
   robustness: Float       # estabilidad a ruido / bootstrap
   score: Float            # combinación penalizada (MDL/parquedad)
   evidence: Dict          # stats + intervalos

type MacroState Φ:
   id: Int
   centroid: Vector
   members: List[ψ]

type MSM:  # Markov State Model
   states: List[Φ]
```

```
    P: Matrix             # transición entre macrostates
    implied_timescales: List[Float]
    spectral_gap: Float
    tau_relax: Float
    tau_exit: Float
    metastable_ok: Bool

type UEModel:
    R: ℛ
    epsilon: Float
    energy: function(Vector x) -> Float      # E(x) ~ -log p(x|ℛ) + const
    cost: function(Vector x) -> Float        # F(x) = epsilon * E(x)
    propose: function(current_x) -> List[Vector]  # candidatos (q)
    constraints: function(current_x, next_x) -> Bool
    sc_select: function(current_x, candidates) -> Vector  # regla SC
    coarse_grain: function(List[ψ]) -> List[Φ]
    msm_fit: function(List[ψ], List[Φ]) -> MSM
    predictions: List[function(TestData)->MetricReport]

type Domain:
    name: String
    R: ℛ
    locks: List[Lock]
    macrostates: List[Φ]
    msm: MSM
    model: UEModel
    status: {"PASS","FAIL","INCONCLUSIVE"}
    report: Dict

type ValidationSpec:
    preregistered_R: List[ℛ]
    split: {train, val, test}         # por tiempo / por grupos
    min_samples_R: Int
    representation_spec
    energy_spec                        # classifier / density ratio / flow
    lock_search_spec
    cg_spec                          # coarse-graining
    msm_spec
    ablations: List[Ablation]
    falsification_tests: List[Test]
    robustness_checks: List[Check]
    success_metrics: List[Metric]
    stop_rules: List[Rule]
    alpha: Float                      # nivel de significancia (si aplica)

##############################
# 1) ERS (ℛ): EVENTO–RARO–SOSTENIDO
##############################

function build_ers(data, name, predicate, persistence_tau, control_match_spec) -> ℛ:
    mask[t] = predicate(data, t)
    windows_R = merge_true_segments(mask)
```

```
    windows_R = [w for w in windows_R if len(w) >= persistence_tau]

    # Ventanas ¬ℛ emparejadas: mismo tamaño, mismo contexto (hora/día/estación), sin solapar ℛ
    windows_notR = match_control_windows(data.times, windows_R, control_match_spec)

    return ERSCondition(name, predicate, persistence_tau, windows_R, windows_notR)

##############################
# 2) REPRESENTACIÓN: EXTRAER MICROESTADOS (ℛ y ¬ℛ)
##############################

function represent(data, t, i, representation_spec) -> Vector:
    raw = data.slice(t,i) or data.global_slice(t)
    x = feature_engineering(raw, representation_spec)
    return x

function extract_microstates(data, R, representation_spec) -> List[ψ]:
    states = []
    window_id = 0
    for w in R.windows_R:
        for t in w:
            for i in data.nodes:
                x = represent(data,t,i,representation_spec)
                states.append(MicroState(x=x,t=t,i=i,label_R=True,window_id=window_id))
        window_id += 1

    for w in R.windows_notR:
        for t in w:
            for i in data.nodes:
                x = represent(data,t,i,representation_spec)
                states.append(MicroState(x=x,t=t,i=i,label_R=False,window_id=window_id))
        window_id += 1

    return states

##############################
# 3) ENERGÍA / COSTE: ESTIMAR E(x) ~ -log p(x|ℛ) (VÍA CLASIFICADOR)
# Nota: esto evita normalizar densidades difíciles y da contraste directo ℛ vs ¬ℛ
##############################

function fit_energy_classifier(states, energy_spec) -> function energy(x):
    X = [s.x for s in states]
    y = [1 if s.label_R else 0 for s in states]
    clf = train_probabilistic_classifier(X, y, energy_spec)

    # E(x) = -log odds = -log( p(R|x) / p(notR|x) )
    return function energy(x):
        pR = clamp(clf.predict_proba(x), min=1e-9, max=1-1e-9)
        odds = pR / (1 - pR)
        return -log(odds)

function build_cost(energy, epsilon) -> function F(x):
```

```
    return function F(x):
        return epsilon * energy(x)

##############################
# 4) DINÁMICA SC: PROPUESTAS + CONSTRAINTS + SELECCIÓN
##############################

function build_proposer(proposal_spec, data) -> function propose(current_x):
    # ejemplo: vecinos en grafo, perturbación gaussiana, generador aprendido, etc.
    return function propose(current_x):
        return generate_candidates(current_x, proposal_spec, data)

function build_constraints(constraints_spec) -> function constraints(x, x2):
    return function constraints(x, x2):
        return check_constraints(x, x2, constraints_spec)

function sc_select(current_x, candidates, F, mode, temperature):
    feasible = [x2 for x2 in candidates if constraints(current_x, x2)]
    if len(feasible) == 0: return current_x

    if mode == "argmin":
        return argmin_{x2 in feasible} F(x2)

    if mode == "softmin_sample":
        w[x2] = exp( -F(x2) / temperature )
        return sample(feasible, weights=w)

##############################
# 5) CANDADOS (LOCKS): PROPONER + SCORE CON CONTRASTE + PARQUEDAD +
MULTIHIPÓTESIS
##############################

function propose_lock_hypotheses(states_R, states_notR, lock_search_spec) -> List[(name,
predicate_on_x)]:
    # genera reglas candidatas (árboles pequeños, reglas lineales sparse, invariantes aproximados...)
    # Debe devolver reglas interpretables + límites de complejidad.
    return search_rules(states_R, states_notR, lock_search_spec)

function score_lock(rule, states_R, states_notR, lock_score_spec) -> Lock:
    pR = mean([rule(s.x) for s in states_R])
    pN = mean([rule(s.x) for s in states_notR])

    specificity = log( (pR+1e-9) / (pN+1e-9) )   # simple
    tau_persist = estimate_persistence(rule, states_R)  # en tiempo, no solo en conteo

    robustness = bootstrap_stability(rule, states_R, states_notR)

    complexity_penalty = mdl_penalty(rule)  # penaliza reglas largas
    score = combine(pR, specificity, tau_persist, robustness) - complexity_penalty

    return Lock(name=rule.name, predicate_on_x=rule.predicate,
                support_R=pR, support_notR=pN,
```

```
            specificity=specificity, tau_persist_R=tau_persist,
            robustness=robustness, score=score,
            evidence={...})

function detect_locks(states, R, lock_spec, alpha) -> List[Lock]:
   states_R = [s for s in states if s.label_R]
   states_N = [s for s in states if not s.label_R]

   rules = propose_lock_hypotheses(states_R, states_N, lock_spec.search)

   locks = []
   for rule in rules:
      L = score_lock(rule, states_R, states_N, lock_spec.score)
      if L.tau_persist_R >= R.persistence_tau and L.score >= lock_spec.min_score:
         locks.append(L)

   # Control de múltiples hipótesis (p.ej. BH/FDR) si usas p-values
   locks = multiple_hypothesis_control(locks, alpha, method=lock_spec.mh_method)

   # Selección final: conjunto mínimo que explica la mayor parte del contraste (parquedad)
   locks = select_minimal_cover(locks, lock_spec.cover_target)

   return sort_by_score_desc(locks)

##############################
# 6) COARSE-GRAINING Y METASTABILIDAD (MSM)
##############################

function build_macrostates(states_R_only, cg_spec) -> List[Φ]:
   X = [s.x for s in states_R_only]
   clusters = cluster(X, cg_spec)
   Φ_list = []
   for c in clusters:
      Φ_list.append(MacroState(id=c.id, centroid=c.centroid, members=c.members_states))
   return Φ_list

function assign_macrostate(x, Φ_list) -> Int:
   return nearest_centroid_id(x, Φ_list)

function fit_msm(states_R_only, Φ_list, msm_spec) -> MSM:
   # construye serie de ids Φ(t), estima matriz de transición con lag τ
   seq = []
   for s in sort_by_time(states_R_only):
      seq.append(assign_macrostate(s.x, Φ_list))

   P = estimate_transition_matrix(seq, lag=msm_spec.lag, regularize=msm_spec.reg)
   implied = compute_implied_timescales(P, msm_spec)
   gap = spectral_gap(P)
   tau_relax = relaxation_time(implied)        # a partir del 2º autovalor
   tau_exit = exit_time_estimate(P, msm_spec)   # dwell times / MFPT

   ok = (tau_relax * msm_spec.gap_factor) < tau_exit
```

```
    return MSM(states=Φ_list, P=P, implied_timescales=implied,
               spectral_gap=gap, tau_relax=tau_relax, tau_exit=tau_exit,
               metastable_ok=ok)

##############################
# 7) PREDICCIONES MÍNIMAS (TESTEABLES)
##############################

function default_predictions(domain) -> List[predictor]:
    preds = []

    # P1: distribución de dwell times por macroestado bajo ℛ
    preds.append( function(test_data):
        return test_dwell_times(domain.msm, test_data, domain.R) )

    # P2: tasas de salida (MFPT) cambian si rompes candado clave (ablación/intervención)
    preds.append( function(test_data):
        return test_lock_intervention_sensitivity(domain, test_data) )

    # P3: bajo ℛ, aumenta la masa en cuencas Φ* predichas vs ¬ℛ emparejado
    preds.append( function(test_data):
        return test_macro_mass_shift(domain, test_data) )

    return preds

##############################
# 8) CONSTRUIR DOMINIO
##############################

function build_domain(data_train, R, spec) -> Domain:
    states = extract_microstates(data_train, R, spec.representation_spec)

    # regla de parada por datos insuficientes
    if count([s for s in states if s.label_R]) < spec.min_samples_R:
        return Domain(name="D_"+R.name, R=R, status="INCONCLUSIVE",
report={"reason":"insufficient_R_samples"})

    energy = fit_energy_classifier(states, spec.energy_spec)
    F = build_cost(energy, spec.energy_spec.epsilon)

    propose = build_proposer(spec.energy_spec.proposal, data_train)
    constraints = build_constraints(spec.energy_spec.constraints)

    model = UEModel(R=R, epsilon=spec.energy_spec.epsilon,
                    energy=energy, cost=F,
                    propose=propose, constraints=constraints,
                    sc_select=function(x,cands):
sc_select(x,cands,F,spec.energy_spec.sc_mode,spec.energy_spec.temperature),
                    coarse_grain=None, msm_fit=None, predictions=[])

    locks = detect_locks(states, R, spec.lock_search_spec, spec.alpha)
```

```
    # sólo ℛ para cuencas/metastabilidad
    states_R_only = [s for s in states if s.label_R]
    Φ_list = build_macrostates(states_R_only, spec.cg_spec)
    msm = fit_msm(states_R_only, Φ_list, spec.msm_spec)

    # decide PASS/FAIL/INCONCLUSIVE
    if len(locks) == 0 or not msm.metastable_ok:
        status = "FAIL"
    else:
        status = "PASS"

    domain = Domain(name="D_"+R.name, R=R, locks=locks, macrostates=Φ_list, msm=msm, model=model, status=status, report={})
    domain.model.predictions = default_predictions(domain)
    return domain

##############################
# 9) VALIDACIÓN: PREREGISTRO + ABLACIONES + FALSIFICACIÓN + OOS
##############################

function preregister(validation_spec):
    # Congela: ℛ, representación, búsqueda de locks, cg, msm, métricas y tests.
    # Idealmente guarda hash del pipeline + fecha + versión de datos.
    save_frozen_spec(hash(validation_spec), validation_spec)

function rebuild_domain_same_R(data_train, R, frozen_spec) -> Domain:
    # MISMA ℛ, MISMA representación, MISMO pipeline (sin “tocar a mano”)
    return build_domain(data_train, R, frozen_spec)

function run_ablations(domain, data_train, frozen_spec) -> List[Dict]:
    out = []
    for abl in frozen_spec.ablations:
        data_abl = apply_ablation(data_train, abl)
        D_abl = rebuild_domain_same_R(data_abl, domain.R, frozen_spec)
        out.append({"abl":abl.name, "delta":compare_domain(domain, D_abl)})
    return out

function run_falsification(domain, data_train, frozen_spec) -> List[Dict]:
    out = []

    # Placebo 1: etiquetas ℛ barajadas (mismo número/longitud de ventanas)
    data_pl = shuffle_R_windows(data_train, domain.R)
    D_pl = rebuild_domain_same_R(data_pl, domain.R, frozen_spec)
    out.append({"test":"placebo_shuffle_R", "result":compare_domain(domain, D_pl)})

    # Placebo 2: ℛ aleatoria emparejada
    R_rand = random_ers_like(domain.R, data_train.times)
    D_rand = rebuild_domain_same_R(data_train, R_rand, frozen_spec)
    out.append({"test":"random_R_like", "result":summarize(D_rand)})

    return out
```

```
function out_of_sample_eval(domain, data_split, metrics) -> Dict:
    reports = []
    for pred in domain.model.predictions:
        reports.append(pred(data_split))
    return aggregate_metrics(reports, metrics)

function should_stop(outputs, stop_rules) -> Bool:
    for rule in stop_rules:
        if rule.triggers(outputs): return True
    return False

##############################
# 10) PIPELINE UE
##############################

function UE_PIPELINE(data, frozen_spec) -> List[Domain]:
    preregister(frozen_spec)

    domains = []
    for R in frozen_spec.preregistered_R:
        D = build_domain(data.train, R, frozen_spec)

        if D.status == "INCONCLUSIVE":
            domains.append(D)
            continue

        abl = run_ablations(D, data.train, frozen_spec)
        fals = run_falsification(D, data.train, frozen_spec)
        rob  = run_robustness_checks(D, data.val, frozen_spec.robustness_checks)
        val  = out_of_sample_eval(D, data.val, frozen_spec.success_metrics)

        outputs = {"ablations":abl, "falsification":fals, "robustness":rob, "val":val}

        if should_stop(outputs, frozen_spec.stop_rules):
            D.status = "INCONCLUSIVE"
            D.report["stop_reason"] = triggered_rule(outputs, frozen_spec.stop_rules)
        else:
            D.report.update(outputs)

        domains.append(D)

    # Test final
    for D in domains:
        if D.status == "PASS":
            test = out_of_sample_eval(D, data.test, frozen_spec.success_metrics)
            D.report["test"] = test

    return domains

##############################################################
# (OPCIONAL) 11) MÓDULOS: COLAS PESADAS / HIPERBOLICIDAD / MULTI-DOMINIO
##############################################################
```

```
function heavy_tail_module(series, tail_spec) -> Dict:
    # compara colas pesadas vs alternativas (p.ej. lognormal) con tests robustos
    return estimate_and_compare_tails(series, tail_spec)

function hyperbolicity_module(graph_or_states, hyp_spec) -> Dict:
    # estima crecimiento de bolas / accesibilidad / ramificación efectiva
    return estimate_hyperbolic_signatures(graph_or_states, hyp_spec)

function multi_domain_couplings(domains, data, coupling_spec) -> List[Dict]:
    # usa métricas como CMI/transfer-entropy con permutaciones para significancia
    return estimate_couplings_with_significance(domains, data, coupling_spec)
```