Verification Code Dossier

Article 104: From Registry-Controlled Capture to Productive Molecular Assembly

Constructive Existence of a Minimal Abiotic Loom

Annals of the Chrysene Formalism, Vol. 1, No. 5
November 2026

Executed finite verification

Script	`verify_article104.py`
Recorded assertion executions	**4,936 passed; 0 failed**
Four-residue graph words	16
Independent rate-box corners	512
Program/terminal histories	112
Deliberately invalid cases rejected	14
Arithmetic	Integers and exact rational fractions
Dependencies	Python standard library only
Tested Python	3.12.14
Execution status	Exit code 0

Scope of this certificate

These are repeated finite assertion executions, not 4,936 independent theorems. The program checks the stated reduced model and selected exact certificates. It does not establish chemical synthesis, an atomistic reaction trajectory, a physical molecular writer, or all continuous transport collisions. It is separate from the accompanying partial Lean formalization.

Reproduction and file identity

Save the complete source below as `verify_article104.py` and run:

```
python3 verify_article104.py
```

Python 3.10 or newer is sufficient. No network access, random inputs, package installation, or third-party numerical libraries are required. The script writes `results_article104.json` next to itself. The JSON includes exact fractions, cases, and category counts; floating-point decimal displays are explanatory only.

Compiling this document in Overleaf produces the dossier PDF. It does *not* execute Python or Lean. The recorded status above comes from an actual separate execution of the embedded Python source.

Source SHA-256:
`a97778cec6c7b7b191532e77238f71b6e287e50af773583c59c85ad5f574e553`

Verification block index and manuscript correspondence

Block	Exact or finite object checked	Interpretation and limitation
V1: joining	All 16 explicit-hydrogen Gly/Ala tetramers and intermediate prefixes; neutral valence, origins, waters, connected product, and formulas	Supports the finite graph witness and ledger. It is not a chemical reaction mechanism.
V2: resources	Common eight-packet pool, held/unused/incorporated partitions, unused alternatives, loss-to-waste accounts	Missing or duplicated material and invalid records are rejected. No material is erased on loss.
V3: kinetics	Exact initialized yield, wrong/loss probabilities, collected-chain purity, seven-leaf tree, and conditional mean-time arithmetic	The paper supplies the CTMC interpretation and exponential-waiting proof. Wrong shorter chains remain in purity.
V4: perturbations	512 rate-box corners and rational endpoint data for the analytic whole-box bound; delivery diagnostics	Corner enumeration and a small delivery grid alone are not universal proofs. The monotonicity arguments remain in the paper.
V5: receiver	All 16 signature cases, stationary points, minimum energy identities, positive Hessian	The neutral interaction law is stipulated. These are not calibrated Gly/Ala binding energies.
V6: Taylor bounds	Degree-40 rational sums/tails and three occupancy intervals; contrast/readout margins	The analytic positive-series tail argument connects these rational expressions to real exponentials; no sampled approximation is used as proof.
V7: geometry	Full patch counts, center separations, basis data, exact segment-to-core projections, packet and collar margins	Not a complete tether-to-tether collision proof. Irrational boundary rays use the paper argument.
V8: protocol	112 histories across 16 programs and seven terminal branches; waiting, selection, transport, commitment, stopping	Selection alone does not debit joining resources or append a code symbol. No hidden transport failure is simulated.
V9: negative controls	14 invalid material, code, rate, purity, phase, and resource examples	The test passes only if the intended validators reject these examples.

Central exact results

The nominal initialized-attempt values are

$$Y = \frac{125000}{132651} > 0.94, \quad B = D = \frac{7651}{265302}, \quad \Pi = \frac{250000}{257651} > 0.97.$$

Purity is a number/probability fraction among all collected chains, not a mass fraction and not a tetramer-only fraction. Delivered purity has its own conditional-delivery formula.

The receiver intervals are

	strict lower	strict upper
no product	0.15428	0.15429
$GGGA$	0.89459	0.89461
$GAGA$	0.98659	0.98660

They imply contrasts exceeding 0.83 and 0.09. For rational $t > 0$, $N = 40$, and $t < N + 2$, the certificate uses

$$S_N(t) = \sum_{k=0}^{N} \frac{t^k}{k!}, \quad U_N(t) = S_N(t) + \frac{t^{N+1}}{(N+1)!} \frac{1}{1 - t/(N+2)}.$$

Positive Taylor terms and a geometric bound on the omitted term ratios give $S_N(t) < e^t \le U_N(t)$. The script then performs exact reciprocal/normalization comparisons. The relation to e^t is this analytic argument, not a floating-point library call.

Negative controls and scope boundaries

The rejected examples include a reused or doubly incorporated packet, a mismatched record, omitted or double-counted water, an inserted foreign atom, wrong-output omissions from purity, a negative event rate, commitment before transport, advancing an uncommitted selection, insufficient tickets or export slots, and selection after stopping.

Whole-segment rational projections cover the stated core/packet comparisons, not every possible tether deformation. The receiver body is assumed to remain inside its specified envelope; intentional collar contact is not ordinary positive clearance. Molecular activation, solvent chemistry, protected atomwise physical conformance, and an actual nucleic-acid writer require their separate certificates.

Complete executable Python source

```python
#!/usr/bin/env python3
"""Exact finite checks for Article 104; not an atomistic simulation or Lean proof.

Uses only Python's standard library. Run with Python 3.10 or newer.
All proof-bearing comparisons use integers/Fraction. Decimal displays are
non-certifying aids; rational endpoints in results_article104.json are certificates.
The code checks a finite model, not physical chemical realizability.
"""
from collections import Counter
from fractions import Fraction as F
from itertools import product
from pathlib import Path
import json

CHECKS = Counter()
def check(condition, category):
    if not condition:
        raise AssertionError(category)
    CHECKS[category] += 1

def edge(a, b):
    return tuple(sorted((a, b)))
```

```
class Graph:
    def __init__(self):
        self.elements, self.bonds, self.origins = {}, {}, {}
    def atom(self, packet, name, element):
        v = packet + ':' + name
        assert v not in self.elements
        self.elements[v], self.origins[v] = element, packet
        return v
    def bond(self, a, b, order=1):
        assert a != b and edge(a,b) not in self.bonds
        self.bonds[edge(a,b)] = order
    def components(self):
        adj = {v: set() for v in self.elements}
        for a,b in self.bonds:
            adj[a].add(b); adj[b].add(a)
        unseen = set(adj); result = []
        while unseen:
            seed = unseen.pop(); comp = {seed}; todo = [seed]
            while todo:
                v = todo.pop()
                for u in adj[v] & unseen:
                    unseen.remove(u); comp.add(u); todo.append(u)
            result.append(comp)
        return result
    def formula(self, vertices=None):
        return Counter(self.elements[v] for v in (self.elements if vertices is None else vertices))
    def valid(self):
        degree = Counter()
        for (a,b), n in self.bonds.items():
            if a == b or n not in (1,2,3) or a not in self.elements or b not in self.elements:
                return False
            degree[a] += n; degree[b] += n
        valence = {'C':4, 'H':1, 'N':3, 'O':2}
        return all(degree[v] == valence[e] for v,e in self.elements.items())

def residue(g, name, kind):
    atoms = {}
    def atom(label, element):
        atoms[label] = g.atom(name, label, element)
    for label, element in [('n','N'),('ca','C'),('c','C'),('o','O'),('w','O'),('hn1','H'),('hn2','H'),('ho'
    ,'H'),('ha1','H')]:
        atom(label,element)
    for a,b,n in [('n','ca',1),('ca','c',1),('c','o',2),('c','w',1),('n','hn1',1),('n','hn2',1),('w','ho'
    ,1),('ca','ha1',1)]:
        g.bond(atoms[a],atoms[b],n)
    if kind == 'G':
        atom('ha2','H'); g.bond(atoms['ca'],atoms['ha2'])
    else:
        atom('cb','C'); g.bond(atoms['ca'],atoms['cb'])
        for k in range(3):
            h='hb'+str(k); atom(h,'H'); g.bond(atoms['cb'],atoms[h])
    return atoms

def graph_checks():
    cases = []
    for letters in product('GA', repeat=4):
        word=''.join(letters); g=Graph(); packets={}
        for j in range(4):
            for kind in 'GA':
                packets[(j,kind)] = residue(g,f'{kind}{j+1}',kind)
        original_elements=dict(g.elements); original_origins=dict(g.origins)
        check(g.valid(), 'neutral_valence')
        check(g.formula()==Counter(C=20,H=48,N=8,O=16) and len(g.elements)==92,'common_pool')
        waters=[]; selected=set(); record=[]
        for j,kind in enumerate(letters):
            incoming=packets[(j,kind)]; selected.update(incoming.values()); record.append('C_'+kind)
            if j:
                prev=packets[(j-1,letters[j-1])]
```

```
                for a,b in [(prev['c'],prev['w']),(incoming['n'],incoming['hn1'])]:
                    check(g.bonds.pop(edge(a,b))==1,'join_precondition')
                g.bond(prev['c'],incoming['n']); g.bond(prev['w'],incoming['hn1'])
                waters.append({prev['w'],prev['ho'],incoming['hn1']})
            check(g.elements==original_elements and g.origins==original_origins,'persistent_atoms_and_origins')
            check(g.valid(),'neutral_valence')
            check(g.formula()==Counter(C=20,H=48,N=8,O=16),'conserved_ledger')
            components=g.components(); water_atoms=set().union(*waters) if waters else set()
            check(all(w in components and g.formula(w)==Counter(H=2,O=1) for w in waters),'explicit_waters')
            check(len(water_atoms)==3*j,'disjoint_waters')
            chain=selected-water_atoms; n=j+1; a=letters[:n].count('A')
            check(chain in components,'single_product_chain')
            check(g.formula(chain)==Counter(C=2*n+a,H=3*n+2*a+2,N=n,O=n+1),'prefix_formula')
            check(all(v in chain for v in selected if g.elements[v] in ('C','N')),'retained_carbon_nitrogen')
            check(''.join(token[2:] for token in record)==word[:n],'external_code_prefix')
        cases.append({'word':word,'product_atoms':len(chain),'product_formula':dict(sorted(g.formula(chain).items())),'waters':len(waters),'unused_atoms':92-len(selected)})
    return cases

def kinetic_values(rates):
    Y=F(1); B=D=F(0)
    for a,b,d in rates:
        total=a+b+d
        B += Y*b/total; D += Y*d/total; Y *= a/total
    return Y,B,D,Y/(Y+B)

def exact_string(x): return str(x.numerator)+'/'+str(x.denominator)
def interval_record(bounds):
    return {'lower':exact_string(bounds[0]), 'upper':exact_string(bounds[1]), 'display':[float(x) for x in bounds]}

def kinetic_checks():
    Y,B,D,P=kinetic_values([(F(100),F(1),F(1))]*3)
    for actual,expected in [(Y,F(125000,132651)),(B,F(7651,265302)),(D,F(7651,265302)),(P,F(250000,257651))]:
        check(actual==expected,'kinetic_exact_fraction')
    check(Y+B+D==1,'absorption_partition'); check(Y>F(94,100) and P>F(97,100),'kinetic_strict_threshold')
    plow=F(99900,101902)**3; rho=F(1001,2000); purity=plow/(plow+rho*(1-plow))
    check(plow>F(94,100) and purity>F(97,100),'robust_strict_threshold')
    corners=list(product((F(999,10),F(1001,10)),(F(999,1000),F(1001,1000)),(F(999,1000),F(1001,1000))))
    for rates in product(corners,repeat=3):
        y,b,d,p=kinetic_values(rates)
        check(y+b+d==1,'corner_absorption_partition')
        check(y>=plow and p>=purity,'robust_corner_bounds')
        check(all(bb/(bb+dd)<=rho for _,bb,dd in rates),'wrong_share_corner_bound')
    check(3*F(1,102)==F(1,34),'conditional_mean_time')
    return {'yield':exact_string(Y),'wrong':exact_string(B),'loss':exact_string(D),'purity':exact_string(P),'robust_yield_lower':exact_string(plow),'robust_purity_lower':exact_string(purity),'corner_cases':len(corners)**3}

def exp_positive_bounds(t,N=40):
    """S_N < exp(t) < S_N + first_tail/(1-t/(N+2)), t>0."""
    assert 0<t<N+2
    term=F(1); partial=term
    for k in range(1,N+1):
        term *= t/k; partial += term
    first_tail=term*t/(N+1)
    return partial,partial+first_tail/(1-t/(N+2))

def exp_negative_bounds(t):
    lo,hi=exp_positive_bounds(t)
    return 1/hi,1/lo

def occupancy_negative(t1,t2):
    a,b=exp_negative_bounds(t1),exp_negative_bounds(t2)
    return 1/(1+a[1]+b[1]),1/(1+a[0]+b[0])
```

```
def receiver_checks():
    certificates={}
    intervals={'GAGA':occupancy_negative(F(6),F(9,2)), 'GGGA':occupancy_negative(F(96,25),F(117,50))}
    lo,hi=exp_positive_bounds(F(3,2)); intervals['absent']=(1/(2+hi),1/(2+lo))
    desired={'GAGA':(F(98659,100000),F(98660,100000)), 'GGGA':(F(89459,100000),F(89461,100000)), 'absent'
     :(F(15428,100000),F(15429,100000))}
    for name,(lo,hi) in intervals.items():
        left,right=desired[name]
        check(left<lo<hi<right,'strict_exponential_enclosure')
        certificates[name]={'exact_taylor_interval':interval_record((lo,hi)),'certified_decimal_interval':
     interval_record((left,right))}
    check(intervals['GAGA'][0]-intervals['GGGA'][1]>F(9,100),'sequence_contrast')
    check(intervals['GAGA'][0]-intervals['absent'][1]>F(83,100),'absent_contrast')
    for word in map(''.join,product('GA',repeat=4)):
        signature=sum(2**j for j,s in enumerate(word) if s=='A'); lam=F(2,5)*signature
        check(-5<F(3,4)*lam<5,'polarization_minimum_interior')
        check(2<F(11,4)<6 and 2<F(3)<6,'opening_minimum_interior')
    check(16>0 and F(4,3)>0,'receiver_hessian_positive')
    check(F(3,8)*4**2==6 and F(3,8)*F(16,5)**2==F(96,25),'receiver_energy_shift')
    return {'taylor_order':40,'method':'Positive exponential Taylor polynomial with geometric tail bound;
     negative exponentials bounded by reciprocation; all arithmetic Fraction.','occupancies':certificates}

def geometry_checks():
    nodes=list(product(range(4),range(2),range(2))); points={v:(4*v[0]+v[1],4*v[1],4*v[2]) for v in nodes};
      edges=[]
    for i,u in enumerate(nodes):
        for v in nodes[i+1:]:
            check(sum((a-b)**2 for a,b in zip(points[u],points[v]))>=16,'core_center_separation')
            diff=[abs(a-b) for a,b in zip(u,v)]
            if sum(diff)==1: edges.append((u,v))
    lateral=sum(u[2]==v[2] for u,v in edges); stacking=len(edges)-lateral
    check(len(nodes)==16 and lateral==20 and stacking==8,'intact_patch_counts')
    check(4*4*4==64,'lattice_determinant')
    return {'nodes':len(nodes),'lateral_edges':lateral,'stacking_edges':stacking,'
     minimum_squared_center_separation':16,'lattice_determinant':64,'note':'Exterior transport inequalities
      are proved in manuscript; no atomistic trajectories simulated.'}

# Publication supplement: executable validators and independent cross-checks.
# These are finite certificates. Analytic identities used below are stated
# explicitly; a finite grid is never substituted for a universal theorem.
NEGATIVE_CONTROLS = []

def reject(call, label):
    try:
        call()
    except ValueError:
        check(True, 'negative_control_rejected')
        NEGATIVE_CONTROLS.append(label)
    else:
        raise AssertionError('Invalid model accepted: ' + label)

def require(condition, message):
    if not condition:
        raise ValueError(message)

class ResourceLedger:
    """Disjoint packet provenance and a truthful external symbolic record."""
    def __init__(self):
        self.status = {f'{kind}{j}': 'unused' for j in range(1,5) for kind in 'GA'}
        self.selected = []
        self.record = []
    def hold(self, packet):
        require(packet in self.status and self.status[packet]=='unused', 'Packet not available')
        self.status[packet]='held'
    def incorporate(self, packet, token):
        require(packet in self.status and self.status[packet]=='held', 'Packet not held')
        require(token == 'C_'+packet[0], 'Code does not describe selected species')
        self.status[packet]='incorporated'; self.selected.append(packet); self.record.append(token)
```

```
    def validate_partition(self, universe, bins):
        parts = [set(b) for b in bins]
        require(sum(map(len,parts)) == len(set().union(*parts)), 'Atom occurs in two bins')
        require(set().union(*parts)==set(universe), 'Missing or foreign atom')

def ledger_checks():
    cases=[]
    for letters in product('GA',repeat=4):
        g=Graph(); packets={}
        for j in range(1,5):
            for kind in 'GA': packets[kind+str(j)]=residue(g,kind+str(j),kind)
        ledger=ResourceLedger(); waters=[]; selected=set()
        for j,kind in enumerate(letters,1):
            name=kind+str(j); ledger.hold(name)
            held=set(packets[name].values())
            unused=set().union(*(set(packets[k].values()) for k,s in ledger.status.items() if s=='unused'))

            # Before a join, product, existing water, held, and unused partition the full pool.
            water=set().union(*waters) if waters else set()
            ledger.validate_partition(g.elements,[selected-water,water,held,unused])
            check(True,'eight_packet_prejoin_partition')
            if j>1:
                prev=packets[letters[j-2]+str(j-1)]; inc=packets[name]
                for a,b in [(prev['c'],prev['w']),(inc['n'],inc['hn1'])]: del g.bonds[edge(a,b)]
                g.bond(prev['c'],inc['n']); g.bond(prev['w'],inc['hn1'])
                waters.append({prev['w'],prev['ho'],inc['hn1']})
            selected |= held; ledger.incorporate(name,'C_'+kind)
            water=set().union(*waters) if waters else set()
            ledger.validate_partition(g.elements,[selected-water,water,unused])
            check(g.valid(),'eight_packet_postjoin_partition_valence')
            # Loss moves atoms to waste; it does not annihilate them.
            ledger.validate_partition(g.elements,[set(),selected-water,water,unused])
            check(True,'loss_waste_partition')
        check(sum(s=='incorporated' for s in ledger.status.values())==4 and sum(s=='unused' for s in ledger
    .status.values())==4,'eight_packet_final_status')
        check(''.join(t[2:] for t in ledger.record)==''.join(letters),'truthful_record_validation')
        cases.append(''.join(letters))
    L=ResourceLedger(); L.hold('G1'); L.incorporate('G1','C_G')
    reject(lambda:L.hold('G1'),'reuse of incorporated packet')
    reject(lambda:L.incorporate('G1','C_G'),'double incorporation')
    L.hold('A2')
    reject(lambda:L.incorporate('A2','C_G'),'mismatched code token')
    reject(lambda:L.validate_partition(g.elements,[selected-water,unused]),'omitted water atoms')
    reject(lambda:L.validate_partition(g.elements,[selected,water,unused]),'water counted twice')
    reject(lambda:L.validate_partition(g.elements,[selected-water,water,unused|{'foreign:atom'}]),'foreign
      atom inserted')
    return {'word_cases':cases,'negative_controls':list(NEGATIVE_CONTROLS)}

def valid_rates(rates):
    require(len(rates)==3,'Three stages required')
    require(all(a>0 and b>=0 and d>=0 for a,b,d in rates),'Invalid rates')

def check_purity_claim(rates, claim):
    valid_rates(rates)
    _,_,_,actual=kinetic_values(rates)
    require(claim==actual,'Purity must count every collected off-specification chain')

def kinetic_independent_checks():
    rates=[(F(100),F(1),F(1))]*3
    Y,B,D,P=kinetic_values(rates)
    # Independent enumeration of all terminating paths in this three-stage tree.
    outcomes=[]; survival=F(1)
    for j,(a,b,d) in enumerate(rates,1):
        lam=a+b+d
        outcomes += [('wrong',j+1,survival*b/lam),('loss',j,survival*d/lam)]
        survival*=a/lam
    outcomes.append(('correct',4,survival))
    check(sum(p for _,_,p in outcomes)==1,'independent_tree_normalization')
    check(sum(p for k,_,p in outcomes if k=='wrong')==B,'independent_tree_wrong_population')
```

```
    check({n for k,n,_ in outcomes if k=='wrong'}=={2,3,4},'collected_wrong_chain_lengths')
    reject(lambda:check_purity_claim(rates,F(1)),'discarding every wrong collected chain from purity')
    tetramer_wrong=next(p for k,n,p in outcomes if k=='wrong' and n==4)
    reject(lambda:check_purity_claim(rates,Y/(Y+tetramer_wrong)),'counting only wrong tetramers in purity')

    reject(lambda:valid_rates([(F(-1),F(1),F(1))]*3),'negative correct-event rate')
    # Exact algebraic whole-box certificate: a/lambda increases with a and
    # decreases with b,d. b/(b+d) increases with b and decreases with d.
    amin,amax=F(999,10),F(1001,10); bmin,bmax=F(999,1000),F(1001,1000)
    qmin=amin/(amin+2*bmax); rho=bmax/(bmax+bmin)
    check(qmin==F(99900,101902) and rho==F(1001,2000),'analytic_box_endpoint_certificate')
    check(amin>0 and bmin>0 and amax>=amin and bmax>=bmin,'analytic_box_sign_hypotheses')
    # Common delivery probability preserves purity, unequal delivery does not.
    for eta in (F(1,4),F(1,2),F(1)):
        check(Y*eta/(Y*eta+B*eta)==P,'equal_delivery_preserves_purity')
    low,high=F(4,5),F(1)
    bound=Y*low/(Y*low+B*high)
    for ec,ew in product((low,F(9,10),F(1)),(F(0),F(1,2),high)):
        check(Y*ec/(Y*ec+B*ew)>=bound,'delivery_bound_finite_diagnostic')
    # A concrete counterexample to calling chemical purity delivered purity.
    check(Y*F(1,2)/(Y*F(1,2)+B)<P,'unequal_delivery_counterexample')
    return {'terminal_paths':[{'kind':k,'chain_length':n,'probability':exact_string(p)} for k,n,p in
     outcomes],
            'whole_box_basis':'Monotonicity and the telescoping first-failure identity; corner enumeration
      is diagnostic only.',
            'delivery_lower_example':exact_string(bound)}

def dot(a,b): return sum(x*y for x,y in zip(a,b))
def sub(a,b): return tuple(F(x)-F(y) for x,y in zip(a,b))
def point_segment_distance2(p,a,b):
    """Exact minimum over the entire closed segment, not sampled points."""
    v=sub(b,a); w=sub(p,a); norm=dot(v,v)
    t=max(F(0),min(F(1),dot(w,v)/norm)) if norm else F(0)
    delta=tuple(wi-t*vi for wi,vi in zip(w,v))
    return dot(delta,delta)

def transport_checks():
    points=[(4*i+j,4*j,4*k) for i,j,k in product(range(4),range(2),range(2))]
    b=(0,4,8); routes=[[(0,0,8),b]]
    routes += [[(4*j,0,8),(4*j,8,8),(0,8,8),(0,5,8)] for j in range(1,4)]
    minima=[]
    for j,path in enumerate(routes):
        ds=[point_segment_distance2(q,a,c) for a,c in zip(path,path[1:]) for q in points]
        m=min(ds); minima.append(m)
        check(m>=16,'whole_segment_framework_clearance')
        # Unit core plus packet radius <= 1/2 gives strict disjointness.
        check(m>F(3,2)**2,'packet_core_disjointness')
        if j:
            check(min(point_segment_distance2(b,a,c) for a,c in zip(path,path[1:]))>=1,'
     retained_packet_path_clearance')
            check(1> (F(1,2)+F(1,4))**2,'retained_incoming_radius_gap')
        for k in range(j+1,4):
            parked=(4*k,0,8)
            check(min(point_segment_distance2(parked,a,c) for a,c in zip(path,path[1:]))>1,'
     other_waiting_packet_clearance')
    delivery=[b,(0,8,8),(0,12,8)]
    check(min(point_segment_distance2(q,a,c) for a,c in zip(delivery,delivery[1:]) for q in points)>=16,'
     delivery_framework_clearance')
    check(F(1,2)<2 and 1+F(1,4)<2,'reaction_region_capacity')
    # Every framework edge tube has z<=4+1/10; cores have z<=5.
    check(8-F(1,2)>5 and 8-F(1,2)>4+F(1,10),'framework_envelope_vertical_gap')
    # Tether portions below z=6 are separated vertically from moving packets.
    check(8-F(1,2)>6+F(1,10),'low_tether_vertical_gap')
    # Irrational first tether segments lie along outward free boundary rays.
    # Their x-y direction has squared norm 17 and positive normal projection:
    # input y<0, receiver y>4. The attachment collar is intentional contact.
    check(1**2+4**2==17 and 4>0,'boundary_ray_analytic_data')
    return {'route_minimum_center_distance_squared':[exact_string(x) for x in minima],
```

```
            'method':'Exact closest-point projection on each full rational segment, plus vertical-envelope bounds.',
            'scope':'Does not model atomistic reaction paths or certify all tether collisions. Irrational boundary collars use the manuscript analytic outward-ray argument.'}

def extra_receiver_checks():
    signatures=[]
    for letters in product('GA',repeat=4):
        s=sum(2**j for j,a in enumerate(letters) if a=='A'); signatures.append(s)
        lam=F(2,5)*s; u=F(3,4)*lam; a=F(11,4)
        check(16*(a-3)+4==0 and F(4,3)*u-lam==0,'receiver_stationary_equations')
        energy=8*(a-3)**2+4*a-10+F(2,3)*u*u-lam*u
        check(energy==F(3,2)-F(3,8)*lam*lam,'receiver_minimum_energy_identity')
    check(sorted(signatures)==list(range(16)),'injective_four_slot_signature')
    for t in (F(3,2),F(117,50),F(96,25),F(9,2),F(6)):
        N=40; r=t/F(N+2); lo,hi=exp_positive_bounds(t,N)
        check(0<r<1 and 0<lo<hi,'taylor_tail_sign_hypotheses')
        # Successive omitted term ratios are t/(N+2+j)<=r for all j>=0.
        # This monotonic denominator identity, not testing j, is the proof.
        first=F(1)
        for k in range(1,N+2): first*=t/k
        check((hi-lo)*(1-r)==first,'taylor_geometric_tail_identity')
        check(0<1/hi<1/lo,'reciprocal_interval_order')
    p=occupancy_negative(F(6),F(9,2)); q=occupancy_negative(F(96,25),F(117,50))
    lo,hi=exp_positive_bounds(F(3,2)); absent=(1/(2+hi),1/(2+lo))
    check(p[0]-q[1]-F(2,10000)>F(9,100),'mixing_adjusted_sequence_contrast')
    check(p[0]-absent[1]-F(2,10000)>F(83,100),'mixing_adjusted_absent_contrast')
    return {'signatures':sorted(signatures),'taylor_tail_argument':'All omitted successive-term ratios are at most t/(N+2)<1; sum the dominating geometric series. The rational endpoints are rigorous conditional on the standard exponential series identity.'}

class PhaseMachine:
    """Finite protocol implementation, not a continuous-time simulator.

    Each history starts after successful seed initialization. Selection is not
    a chemical event. No resource consumption is hidden in transport.
    """
    def __init__(self, program):
        self.program=program; self.g=Graph(); self.packets={}
        for j in range(1,5):
            for kind in 'GA': self.packets[kind+str(j)]=residue(self.g,kind+str(j),kind)
        self.ledger=ResourceLedger(); seed=program[0]+'1'
        self.ledger.hold(seed); self.ledger.incorporate(seed,'C_'+program[0])
        self.chain_atoms=set(self.packets[seed].values()); self.waters=[]
        self.last=seed; self.stage=1; self.phase='waiting'; self.selected=None
        self.tickets=3; self.export_slots=3; self.word=program[0]; self.stop=None
        self.waste=set(); self.initial_atoms=dict(self.g.elements)
    def resources(self):
        return (self.tickets,self.export_slots,tuple(self.ledger.record),dict(self.ledger.status),dict(self.g.bonds))
    def select(self, kind):
        require(self.phase=='waiting','Selection requires waiting phase')
        require(kind in 'GA' and len(kind)==1,'Unknown residue species')
        name=kind+str(self.stage+1)
        require(self.ledger.status.get(name)=='unused','Slot packet not fresh')
        self.selected=name; self.phase='selected'
    def transport(self):
        require(self.phase=='selected','Transport requires selection')
        self.ledger.hold(self.selected); self.phase='transporting'
    def commit(self):
        require(self.phase=='transporting','Commit requires completed selected transport')
        require(self.tickets>0 and self.export_slots>0,'No ticket or water export slot')
        require(self.ledger.status[self.selected]=='held','Incoming packet not held')
        prev=self.packets[self.last]; incoming=self.packets[self.selected]
        removed=[edge(prev['c'],prev['w']),edge(incoming['n'],incoming['hn1'])]
        require(all(self.g.bonds.get(e)==1 for e in removed),'Join precondition fails')
        for e in removed: del self.g.bonds[e]
        self.g.bond(prev['c'],incoming['n']); self.g.bond(prev['w'],incoming['hn1'])
        water={prev['w'],prev['ho'],incoming['hn1']}; self.waters.append(water)
```

```
        self.chain_atoms |= set(incoming.values()); self.chain_atoms -= water
        self.ledger.incorporate(self.selected,'C_'+self.selected[0])
        self.word+=self.selected[0]; self.last=self.selected
        self.tickets-=1; self.export_slots-=1; self.phase='committed'
    def advance(self):
        require(self.phase=='committed','Only a committed event may advance')
        if self.word[-1]!=self.program[self.stage]: self.stop='W'; self.phase='stopped'
        elif self.stage==3: self.stop='C'; self.phase='stopped'
        else: self.stage+=1; self.selected=None; self.phase='waiting'
    def lose(self):
        require(self.phase=='waiting','Loss is an active waiting-clock event')
        self.waste=set(self.chain_atoms); self.chain_atoms=set()
        self.stop='D'; self.phase='stopped'
    def validate(self):
        water=set().union(*self.waters) if self.waters else set()
        unused=set().union(*(set(self.packets[k].values()) for k,s in self.ledger.status.items() if s=='
    unused'))
        held=set().union(*(set(self.packets[k].values()) for k,s in self.ledger.status.items() if s=='held'
    )) if 'held' in self.ledger.status.values() else set()
        self.ledger.validate_partition(self.g.elements,[self.chain_atoms,self.waste,water,unused,held])
        return self.g.valid() and self.g.elements==self.initial_atoms

def phase_checks():
    histories=[]
    branches=[('C',3)]+[(kind,stage) for stage in range(1,4) for kind in ('W','D')]
    for letters in product('GA',repeat=4):
        program=''.join(letters)
        for outcome,terminal_stage in branches:
            m=PhaseMachine(program)
            check(m.validate() and len(m.ledger.record)==1 and m.tickets==m.export_slots==3,'
    phase_initialized_seed')
            check(sum(s=='unused' for s in m.ledger.status.values())==7,'phase_seed_alternative_retained')
            while m.phase!='stopped':
                stage=m.stage
                if outcome=='D' and stage==terminal_stage:
                    before=m.resources(); prefix=set(m.chain_atoms)
                    m.lose()
                    check(m.resources()==before and m.waste==prefix,'phase_loss_no_join_or_consumption')
                    check(m.validate(),'phase_loss_conservation')
                    break
                chosen=program[stage]
                if outcome=='W' and stage==terminal_stage: chosen='A' if chosen=='G' else 'G'
                before=m.resources(); m.select(chosen)
                check(m.phase=='selected' and m.resources()==before,'phase_selection_not_commit')
                tickets,slots,record,_,bonds=before; m.transport()
                check(m.phase=='transporting' and m.ledger.status[m.selected]=='held','
    phase_transport_holds_packet')
                check((m.tickets,m.export_slots,tuple(m.ledger.record),dict(m.g.bonds))==(tickets,slots,
    record,bonds),'phase_transport_no_consumption')
                check(m.validate(),'phase_held_partition')
                m.commit()
                check(m.phase=='committed' and m.tickets==tickets-1 and m.export_slots==slots-1,'
    phase_commit_resource_consumption')
                check(len(m.ledger.record)==len(record)+1 and m.ledger.record[-1]=='C_'+chosen,'
    phase_commit_actual_symbol')
                check(m.validate() and len(m.waters)==3-m.export_slots,'phase_commit_atom_ledger')
                m.advance()
                check(m.phase in ('waiting','stopped'),'phase_advance_state')
            joins=len(m.word)-1
            check(m.stop==outcome and m.tickets==3-joins and m.export_slots==3-joins,'
    phase_terminal_outcome')
            check(''.join(x[2:] for x in m.ledger.record)==m.word,'phase_terminal_truthful_record')
            check(m.validate(),'phase_terminal_conservation')
            histories.append({'program':program,'outcome':outcome,'terminal_stage':terminal_stage,'
    actual_word':m.word,'joins':joins,'tickets_remaining':m.tickets,'export_slots_remaining':m.
    export_slots})
    m=PhaseMachine('GAGA'); m.select('A')
    reject(m.commit,'selected event cannot bypass transport and commit')
    reject(m.advance,'selection cannot advance as if committed')
```

```
    m.transport(); m.tickets=0
    reject(m.commit,'commit with insufficient work tickets')
    m.tickets=3; m.export_slots=0
    reject(m.commit,'commit with no water export slot')
    m=PhaseMachine('GAGA'); m.lose()
    reject(lambda:m.select('A'),'selection after stopping')
    # Uniform gap outside designated contact collars. These rational data
    # supplement, but do not replace, the manuscript's geometric argument.
    delta=F(1,10); tube=F(1,10)
    check(F(3,4)-tube-F(1,2)==F(3,20)>delta,'reaction_delivery_collar_margin')
    check(F(1,2)-tube-F(1,4)==F(3,20)>delta,'input_collar_margin')
    check(13-(12+F(1,2))>delta,'receiver_body_delivery_gap')
    check(point_segment_distance2((0,12,8),(0,12,8),(0,13,8))==0,'delivery_terminal_contact_designated')
    return {'histories':histories,'history_count':len(histories),'terminal_branches_per_program':7,
            'collar_margin':exact_string(F(3,20)),'conservative_delta':exact_string(delta),
            'geometry_scope':'Margins outside designated collars only; terminal contact is intentional.
     Receiver body is assumed contained in y>=13. No complete tether collision certificate is claimed.'}

def main():
    results={'scope':'Exact finite model checks, not empirical chemical validation and not a Lean formal
     proof.','graph_cases':graph_checks(),'kinetics':kinetic_checks(),'receiver':receiver_checks(),'
     geometry':geometry_checks()}
    results['packet_ledger']=ledger_checks()
    results['kinetic_independent']=kinetic_independent_checks()
    results['transport']=transport_checks()
    results['receiver_independent']=extra_receiver_checks()
    results['phase_protocol']=phase_checks()
    results['negative_controls']=NEGATIVE_CONTROLS
    results['checks_by_category']=dict(sorted(CHECKS.items())); results['assertions_passed']=sum(CHECKS.
     values()); results['status']='PASS'
    out=Path(__file__).with_name('results_article104.json'); out.write_text(json.dumps(results,indent=2)+'\
     n')
    print(f"PASS: {results['assertions_passed']} exact assertions; 16 graph words; 512 kinetic corner
     cases; 3 rigorous occupancy enclosures.")
    print(f"Negative controls rejected: {len(NEGATIVE_CONTROLS)}; assertion totals count repeated cases,
     not distinct theorems.")
    print('Results:',out)
if __name__=='__main__': main()
```

Recorded console output

The machine-specific output-directory line is omitted below; the full JSON results accompany the source package.

```
PASS: 4936 exact assertions; 16 graph words; 512 kinetic corner cases; 3 rigorous occupancy enclosures.
Negative controls rejected: 14; assertion totals count repeated cases, not distinct theorems.
```

Relation to the formal proof dossier

The companion `Article104.lean` contains 49 actually kernel-checked theorems in a partial finite/rational formalization. It verifies formula balances and generic incidence identities, not the complete Python atom-graph generator. Its rational Taylor-expression inequalities require the same separate analytic exponential argument. The two artifacts have complementary coverage; neither should be described as experimental validation or as a formal proof of the entire molecular paper.