ACF 88 — Verification Dossier

A Finite Axiomatic Framework for the Interior Chrysene Lattice and Protocell Organization

Charles D. Schaper

Annals of the Chrysene Formalism, Vol. 1, Issue 4, pp. 1778–1807

Audited companion: 31 October 2026 Publication formatting: 3 November 2026

Certification Summary

Item	Audited release result
Exact checks per run	**11,437**
Expected domain or obstruction rejections	347
Fresh execution modes	Normal and optimized (`-O`)
Scientific reports	Byte-identical
Deliberate-defect runs rejected	**8 of 8**
Dependencies	Python standard library only
Runtime	Python 3.9+; tested with 3.12.14

1 Purpose and controlling manuscript

This is the executable-verification companion to Article 88, the publication version of the audited 11 October 2026 revised release originally identified as Article 85. The original audited mathematical body and executable code are unchanged. Publication formatting and a contextual literature appendix (Appendix G) have been added. The complete program is printed below and is also carried, with every runtime dependency, in the digital PDF and in this single TeX source. It checks finite constructions and exact arithmetic; it is not a claim that passing a test suite proves every theorem of the article.

The controlling release defines finite counterparts F1–F12 of the native framework and distinguishes the 27-cell existence construction from the separate seven-cell finite-count specialization. Its genetic carrier is the fixed resident type AAA. State-valued preparation is not the subsequent register operation, and element-count balance is not a complete atom map. These distinctions are retained here.

The original complete release archive has SHA-256

```
f5956f59ecd009368e784708bb44dc8860
70421ec1981601fc8143aeaeb49ade
```

The runtime's `SOURCE_PIN.json` records the exact reviewed TeX file hashes. The manuscript's native, genetic and protocell source hierarchy remains Articles 40, revised 75 and revised 77. This companion does not replace their assumptions with older axioms.

For reproducibility, executable code, Lean namespaces, recovery archives, historical filenames and source hashes keep their original Article 85 identities. The frozen `SOURCE_PIN.json` governs coverage of the original audited statements; the new contextual Appendix G is outside that audit.

2 Scope of the executable checks

The checker uses integers and exact rational arithmetic for its stated finite assertions. The complete executable and its negative controls are authoritative for the enumerated inputs; the following is the mathematical reading of that scope.

Manuscript result	Executable coverage and qualification
Theorem 2.2; Proposition 5.1	Reconstruct directional incidence, integrate a spanning tree, and check all remaining edges, coordinate separation and induced-neighbor completeness. Verify the native molecule and the nonempty 27-cell box, including 486 core carbons, 567 core edges, 36 directional connectors, 18 stacking relations, 54 boundary ports, 36 plaquettes and 8 cubes. The enormous tensor dimension is represented symbolically.
Propositions 4.1–4.4; Theorem 6.1	Check explicit partial-shift domains and matrix-unit products, selected standard-form/spectral identities and weighted graph-distance witnesses. Check the supplied F10 motif resolvents and F11 response fractions at explicit rational or imaginary-frequency arguments. These finite calculations are not general complex-operator or norm proofs.
Theorem 9.1 and terminal laws	Enumerate the declared one-error domains of the path and cycle encodings, their decoders, legal operations and typed preparations. Compute exact rational terminal reductions and selected passive-context replacements. A finite list of contexts does not prove the universal Theorem 9.2.
Theorem 10.1; Section 10	Enumerate all 256 tagged address states and the 64, 36 and 20 quotients of their distinct retained-data conventions. Check transported reversal, normal forms, the two selected preparations, and fixed AAA type.
Proposition 11.1; Theorem 11.2	Check selected common-state lifts, material identities, equilibrium water arithmetic and exact inequality certificates used in the example. No numerical trajectory is substituted for the ODE existence, uniqueness, hyperbolic-tangent or continuous enclosure proof.
Appendices D and E	Check the $(264, 270, 10, 48)$ element-count block, the distinct seven-cell carbon ledger, and exact rational constants behind the finite-count stock, threshold, volume and energy estimates. These are arithmetic certificates, not a construction of atom-resolved product graphs or a proof of every attenuation-dependent hybrid execution.

The suite does not mechanize the complete F1–F12 existence theorem, the native-generator Dirac theorem, complex spectral metric separation, causal response integrals and holomorphy, arbitrary real continuous dynamics, complete molecular preparation, reaction barriers, catalysis, folding, coherence, chirality selection, or biological function. The separate Lean companion states exactly which general theorems and rational/finite specializations have kernel-checked proofs.

3 Recovery and execution

The runtime requires Python 3.9 or newer and only its standard library (tested with Python 3.12.14). Neither an original manuscript directory nor a network service is needed to execute it. PDF construction uses ordinary TeX Live packages declared in the preamble; it uses no shell escape and does not execute Python.

3.1 From the TeX file alone

Copy only `Article88_Verification_Code_Overleaf_LaTeX.tex` into an empty writable directory:

```
pdflatex -interaction=nonstopmode -halt-on-error Article88_Verification_Code_Overleaf_LaTeX.tex
pdflatex -interaction=nonstopmode -halt-on-error Article88_Verification_Code_Overleaf_LaTeX.tex
python3 extract_article85_verification.py --output-dir article85_code
python3 article85_code/run_verification.py --output-dir results
```

The TeX compilation writes its embedded payload and recovery helper. The displayed listings are written directly in this LaTeX source. The final Python command performs fresh execution in normal and optimized modes and compares the scientific reports.

3.2 From the digital PDF alone

Save its two original embedded files, or use Poppler:

```
pdfdetach -saveall Article88_Verification_Code_Overleaf_LaTeX.pdf
python3 extract_article85_verification.py --output-dir article85_code
python3 article85_code/run_verification.py --output-dir results
```

The attachments are `article85_verification_payload.b64` and `extract_article85_verification.py`. A printed or rasterized copy does not retain attachments. The payload recovers a complete ZIP, not merely the modules selected for display.

The recovery helper checks the pinned archive digest, exact member inventory, duplicate names, safe relative paths, non-symlink entries, file sizes and individual SHA-256 values before writing any member. It refuses a nonempty destination. Hashes establish integrity; they are not mathematical proofs or digital author signatures.

4 Reproducibility and sensitivity

The fresh-release evidence records normal and `-O` execution, their scientific-report equality, and rejection controls. Ordinary Python `assert` statements are not the scientific guard mechanism. An unexpectedly accepted negative control fails the outer test. Selected controls distinguish a periodic winding obstruction, coordinate collision, omitted induced incidence, a missing partial-map intermediate, bad genetic reference/dispatch, invalid component decoding, and poisoned material or arithmetic claims. This is sensitivity testing, not exhaustive software verification.

The final suite performs **11,437 exact checks** per run, including 347 expected domain or obstruction rejections. Its fresh normal and optimized JSON reports are byte-identical. Four deliberately altered scientific implementations are rejected in each mode (eight rejected mutant runs): wrong cycle conductance, wrong address normalization, corrupted carbon balance and an improperly widened cycle-decoder domain. The geometric/domain rejection fixtures described above are additional checks within the primary suite, not extra compiler-proved theorems.

Both TeX-only and PDF-only recovery were exercised in fresh isolated directories, followed by fresh program execution. The release package contains machine-readable reproducibility evidence and an exact file manifest. Neither cached success reports nor precompiled proof objects are used as substitutes for execution.

4.1 Executable payload identity

The archive SHA-256 is, split only for display,

```
8e78317bd6c063ddb59de9c4db658e7e
d088bc97d78fb5728d443651ac1b03d9
```

The listings that follow reproduce source files from this payload. Line wrapping is typographic only.

5 Complete exact verification program

verify_article85.py

```python
#!/usr/bin/env python3
"""Article 85 exact finite-instance verification, standard library only.

Run: python3 verify_article85.py --output results.json
No source manuscript, network, packages, or floating-point arithmetic required.
Every runtime check remains active under python3 -O. The finite computations
corroborate selected displayed constructions; they are not general proofs.
"""
import argparse
from collections import Counter, deque
from fractions import Fraction as Q
from hashlib import sha256
from itertools import combinations, product
import json
from math import factorial
from pathlib import Path
import sys


class VerificationFailure(RuntimeError):
    pass


class DomainError(ValueError):
    pass


class RecognitionError(DomainError):
    pass


class Suite:
    def __init__(self, mutation=None):
        self.counts = Counter()
        self.trace = sha256()
        self.values = {}
        self.mutation = mutation
        self.expected_rejections = 0

    def check(self, group, label, condition):
        if not condition:
            raise VerificationFailure(group + ': ' + label)
        self.counts[group] += 1
```

```
        self.trace.update((group + ': ' + label + '\n').encode('ascii'))

    def reject(self, group, label, function, exception=DomainError, reason=None):
        try:
            function()
        except exception as error:
            self.check(group, label,
                       reason is None or str(error).startswith(reason))
            self.expected_rejections += 1
        else:
            raise VerificationFailure(group + ': failed to reject ' + label)


ZERO = (0, 0, 0)
AXES = ((1, 0, 0), (0, 1, 0), (0, 0, 1))
DIRS = tuple(tuple(s * x for x in a) for a in AXES for s in (-1, 1))


def plus(a, b):
    return tuple(x + y for x, y in zip(a, b))


def minus(a, b):
    return tuple(x - y for x, y in zip(a, b))


def neg(a):
    return tuple(-x for x in a)


def induced_graph(points):
    points = set(points)
    return {u: {plus(u, d): d for d in DIRS if plus(u, d) in points}
            for u in sorted(points)}


def recognize(graph):
    """Integrate a spanning tree, check all chords, then injectivity/incidence."""
    if not graph:
        raise RecognitionError('empty')
    for u, outgoing in graph.items():
        if len(set(outgoing.values())) != len(outgoing):
            raise RecognitionError('repeated direction')
        for v, d in outgoing.items():
            if u == v or d not in DIRS or v not in graph:
                raise RecognitionError('edge type')
            if graph[v].get(u) != neg(d):
                raise RecognitionError('reverse label')
    root = next(iter(graph))
    coord = {root: ZERO}
    queue = deque([root])
    while queue:
        u = queue.popleft()
        for v, d in graph[u].items():
            candidate = plus(coord[u], d)
            if v not in coord:
                coord[v] = candidate
                queue.append(v)
            elif coord[v] != candidate:
                raise RecognitionError('nonzero period')
    if len(coord) != len(graph):
        raise RecognitionError('disconnected')
    if len(set(coord.values())) != len(graph):
        raise RecognitionError('coordinate collision')
    inverse = {v: u for u, v in coord.items()}
    for u, a in coord.items():
        for d in DIRS:
            v = inverse.get(plus(a, d))
```

```
            if v is not None and v not in graph[u]:
                raise RecognitionError('incomplete incidence')
    return coord


def labelled_path(increments):
    graph = {i: {} for i in range(len(increments) + 1)}
    for i, d in enumerate(increments):
        graph[i][i + 1] = d
        graph[i + 1][i] = neg(d)
    return graph


def reachable(graph, root):
    seen = {root}
    queue = [root]
    while queue:
        for v in graph[queue.pop()]:
            if v not in seen:
                seen.add(v)
                queue.append(v)
    return seen


def eye(n):
    return [[Q(i == j) for j in range(n)] for i in range(n)]


def transpose(a):
    return [list(row) for row in zip(*a)]


def multiply(a, b):
    if not a or not b or len(a[0]) != len(b):
        raise DomainError('matrix dimensions')
    return [[sum(x * y for x, y in zip(row, col))
             for col in zip(*b)] for row in a]


def madd(a, b, scale=1):
    return [[x + scale * y for x, y in zip(ar, br)]
            for ar, br in zip(a, b)]


def scaled(a, scale):
    return [[scale * x for x in row] for row in a]


def inverse(a):
    n = len(a)
    if not n or any(len(row) != n for row in a):
        raise DomainError('not square')
    b = [list(map(Q, row)) + unit for row, unit in zip(a, eye(n))]
    for col in range(n):
        pivot = next((r for r in range(col, n) if b[r][col]), None)
        if pivot is None:
            raise DomainError('singular interior')
        b[col], b[pivot] = b[pivot], b[col]
        b[col] = [x / b[col][col] for x in b[col]]
        for r in range(n):
            if r != col:
                b[r] = [x - b[r][col] * y for x, y in zip(b[r], b[col])]
    return [row[n:] for row in b]


def kron(a, b):
    return [[x * y for x in ar for y in br] for ar in a for br in b]
```

```
def trace(a):
    return sum(a[i][i] for i in range(len(a)))


def matrix_unit(n, row, col):
    return [[Q(i == row and j == col) for j in range(n)] for i in range(n)]


def native_and_recognition(s):
    g = 'native_and_recognition'
    points = [(-2, 2), (-1, 3), (1, 3), (2, 2), (4, 0), (5, -1),
              (5, -3), (4, -4), (2, -4), (1, -3), (-1, -1), (-2, 0),
              (-1, 1), (1, -1), (1, 1), (2, -2), (2, 0), (4, -2)]
    perimeter = [0, 1, 2, 3, 14, 16, 4, 5, 17, 6, 7, 8, 9, 15,
                 13, 10, 11, 12]
    edges = {tuple(sorted((perimeter[i], perimeter[(i + 1) % 18])))
             for i in range(18)} | {(12, 14), (13, 16), (15, 17)}
    dot = lambda a, b: (a[0] * b[0] + 3 * a[1] * b[1]) / Q(4)
    s.check(g, '18 distinct vertices and 21 distinct edges',
            len(set(points)) == 18 and len(edges) == 21)
    for u, v in sorted(edges):
        d = minus(points[u], points[v])
        s.check(g, 'unit carbon edge ' + str((u, v)), dot(d, d) == 1)
    s.check(g, 'carbon degree histogram',
            Counter(sum(i in e for e in edges) for i in range(18)) == {2: 12, 3: 6})
    carbon_adj = {i: {v if u == i else u for u, v in edges if i in (u, v)}
                  for i in range(18)}
    hexagons = set()
    def visit(path):
        if len(path) == 6:
            if path[0] in carbon_adj[path[-1]]:
                reverse = (path[0],) + tuple(reversed(path[1:]))
                hexagons.add(min(tuple(path), reverse))
            return
        for vertex in carbon_adj[path[-1]]:
            if vertex > path[0] and vertex not in path:
                visit(path + [vertex])
    for vertex in range(18):
        visit([vertex])
    faces = []
    for face in sorted(hexagons):
        increments = [minus(points[face[(i + 1) % 6]], points[face[i]]) for i in range(6)]
        s.check(g, 'regular hexagon ' + str(face),
                all(dot(increments[i], increments[(i + 1) % 6]) == Q(1, 2)
                    for i in range(6)))
        faces.append({tuple(sorted((face[i], face[(i + 1) % 6]))) for i in range(6)})
    face_graph = {i: {j for j in range(len(faces)) if i != j and faces[i] & faces[j]}
                  for i in range(len(faces))}
    s.check(g, 'edge-connected union of exactly four hexagons',
            len(faces) == 4 and set().union(*faces) == edges
            and len(reachable(face_graph, 0)) == 4)
    centroid = tuple(sum(Q(p[i]) for p in points) / 18 for i in range(2))
    s.check(g, 'centroid', centroid == (Q(3, 2), Q(-1, 2)))
    radii = [dot(minus(p, centroid), minus(p, centroid)) for p in points]
    s.check(g, 'unique smallest radius pair',
            [i for i, r in enumerate(radii) if r == min(radii)] == [13, 16]
            and min(radii) == Q(1, 4))
    s.check(g, 'unique largest radius pair',
            [i for i, r in enumerate(radii) if r == max(radii)] == [1, 7]
            and max(radii) == Q(43, 4))
    s.check(g, 'diameter squared',
            max(dot(minus(a, b), minus(a, b)) for a, b in product(points, repeat=2)) == 43)
    half = {i: points.index((3 - p[0], -1 - p[1])) for i, p in enumerate(points)}
    s.check(g, 'half turn preserves all carbon edges',
            {tuple(sorted((half[u], half[v]))) for u, v in edges} == edges)
    # An independent exhaustive planar isometry check: images of two
    # independent radius vectors determine every candidate linear isometry.
```

```
    radi = [minus(p, centroid) for p in points]
    base = [[radi[0][0], radi[1][0]], [radi[0][1], radi[1][1]]]
    stabilizers = set()
    metric = [[1, 0], [0, 3]]
    for u, v in product(radi, repeat=2):
        a = multiply([[u[0], v[0]], [u[1], v[1]]], inverse(base))
        if multiply(multiply(transpose(a), metric), a) != metric:
            continue
        image = {tuple(row[0] for row in multiply(a, [[x], [y]])) for x, y in radi}
        if image == set(radi):
            stabilizers.add(tuple(tuple(row) for row in a))
    s.check(g, 'complete planar stabilizer',
            stabilizers == {((1, 0), (0, 1)), ((-1, 0), (0, -1))})
    dx, dy = minus(points[8], points[2]), minus(points[5], points[11])
    s.check(g, 'ordered port Gram matrix',
            (dot(dx, dx), dot(dy, dy), dot(dx, dy)) == (37, 13, 7))
    s.check(g, 'ordered port determinant coefficient',
            Q(dx[0] * dy[1] - dx[1] * dy[0], 4) == 12)
    for locant in (3, 6, 9, 12):
        s.check(g, 'perimeter pairing ' + str(locant),
                (perimeter.index(locant - 2) - perimeter.index(locant - 1)) % 18 == 17)
    # Coordinates here are (rational, coefficient of sqrt(3)) in each axis.
    physical_dx = ((Q(1, 2), 0), (0, Q(-7, 2)), (0, 0))
    physical_dy = ((Q(7, 2), 0), (0, Q(-1, 2)), (0, 0))
    bridges = (((Q(15, 2), 0), (0, Q(7, 2)), (0, 0)),
               ((Q(-7, 2), 0), (8, Q(1, 2)), (0, 0)))
    for axis, (d, b) in enumerate(zip((physical_dx, physical_dy), bridges)):
        s.check(g, 'exact bridge closure ' + str(axis),
                tuple(plus(x, y) for x, y in zip(d, b))
                == tuple((8 * (i == axis), 0) for i in range(3)))
    s.check(g, 'intercell spacing exceeds molecular diameter', 64 > 43)
    box = tuple(product((-1, 0, 1), repeat=3))
    graph = induced_graph(box)
    coordinates = recognize(graph)
    origin = next(iter(graph))
    for p in box:
        s.check(g, 'recognized coordinate ' + str(p), coordinates[p] == minus(p, origin))
    edge_count = sum(map(len, graph.values())) // 2
    boundary = {(p, d) for p in box for d in DIRS if plus(p, d) not in graph}
    interior = [p for p in box if len(graph[p]) == 6]
    squares = [(p, a, b) for p in box for a, b in combinations(AXES, 2)
               if all(q in graph for q in (plus(p, a), plus(p, b), plus(plus(p, a), b)))]
    cubes = [p for p in box if all(plus(p, d) in graph for d in product((0, 1), repeat=3))]
    s.check(g, '27 cell counts',
            (len(box), 18 * len(box), 21 * len(box), edge_count,
             len(boundary), len(squares), len(cubes)) == (27, 486, 567, 54, 54, 36, 8))
    s.check(g, 'genuine graph interior', interior == [ZERO])
    s.check(g, 'typed edge counts',
            [sum(plus(p, a) in graph for p in box) for a in AXES] == [18, 18, 18])
    for p, a, b in squares:
        s.check(g, 'plaquette period ' + str((p, a, b)),
                plus(plus(a, b), plus(neg(a), neg(b))) == ZERO)
    # Exhaust all 255 nonempty subsets of a 2x2x2 ambient cube; accept
    # every connected induced graph and explicitly reject all others.
    small = tuple(product((0, 1), repeat=3))
    connected = 0
    for mask in range(1, 256):
        sub = [p for i, p in enumerate(small) if mask & (1 << i)]
        h = induced_graph(sub)
        if len(reachable(h, sub[0])) == len(sub):
            coord = recognize(h)
            connected += 1
            s.check(g, 'connected small region ' + str(mask),
                    all(minus(coord[p], coord[sub[0]]) == minus(p, sub[0]) for p in sub))
            s.check(g, 'small region boundary ' + str(mask),
                    sum(6 - len(v) for v in h.values()) == 6 * len(h) - sum(map(len, h.values()))
                    and sum(6 - len(v) for v in h.values()) >= 6)
        else:
```

```
            s.reject(g, 'disconnected region ' + str(mask),
                     lambda h=h: recognize(h), RecognitionError, 'disconnected')
    collision = labelled_path((AXES[0], AXES[1], neg(AXES[0]), neg(AXES[1])))
    incomplete = labelled_path((AXES[0], AXES[1], neg(AXES[0])))
    torus = {(i, j): {} for i, j in product(range(3), repeat=2)}
    for i, j in torus:
        for sign in (-1, 1):
            torus[i, j][((i + sign) % 3, j)] = (sign, 0, 0)
            torus[i, j][(i, (j + sign) % 3)] = (0, sign, 0)
    for i, j in torus:
        square = [(i, j), ((i + 1) % 3, j),
                  ((i + 1) % 3, (j + 1) % 3), (i, (j + 1) % 3), (i, j)]
        period = ZERO
        for u, v in zip(square, square[1:]):
            period = plus(period, torus[u][v])
        s.check(g, 'periodic local square closes ' + str((i, j)), period == ZERO)
    winding = [(0, 0), (1, 0), (2, 0), (0, 0)]
    period = ZERO
    for u, v in zip(winding, winding[1:]):
        period = plus(period, torus[u][v])
    s.check(g, 'nonzero winding despite closed local squares', period == (3, 0, 0))
    s.reject(g, 'periodic torus', lambda: recognize(torus), RecognitionError, 'nonzero period')
    s.reject(g, 'distinct vertices at same coordinate', lambda: recognize(collision),
             RecognitionError, 'coordinate collision')
    s.reject(g, 'missing available neighbor edge', lambda: recognize(incomplete),
             RecognitionError, 'incomplete incidence')
    s.reject(g, 'empty region', lambda: recognize({}), RecognitionError, 'empty')
    s.values['native_and_box'] = {
        'molecular_vertices': 18, 'molecular_edges': 21, 'regular_hexagons': 4,
        'diameter_squared': 43,
        'planar_isometries': 2, 'cells': 27, 'core_vertices': 486,
        'core_edges': 567, 'directional_connectors': 36, 'stacking_relations': 18,
        'boundary_incidences': 54, 'interior_cells': 1, 'plaquettes': 36, 'cubes': 8,
        'small_ambient_subsets': 255, 'connected_small_regions': connected}
    return box, graph


def compose(a, b):
    """Sparse partial-permutation operators, with columns as dictionary keys."""
    return {x: a[y] for x, y in b.items() if y in a}


def partial_shift(box, m):
    return {x: plus(x, m) for x in box if plus(x, m) in box}


def shifts_and_metrics(s, box, graph):
    g = 'shifts_and_metrics'
    shifts = tuple(product(range(-2, 3), repeat=3))
    for m in shifts:
        a = partial_shift(box, m)
        adjoint = {y: x for x, y in a.items()}
        s.check(g, 'adjoint ' + str(m), adjoint == partial_shift(box, neg(m)))
        s.check(g, 'initial projection ' + str(m), compose(adjoint, a) == {x: x for x in a})
        s.check(g, 'final projection ' + str(m), compose(a, adjoint) == {y: y for y in a.values()})
        s.check(g, 'no nonzero total translation ' + str(m), (len(a) == len(box)) == (m == ZERO))
    for m, n in product(DIRS, repeat=2):
        composite = compose(partial_shift(box, m), partial_shift(box, n))
        direct = partial_shift(box, plus(m, n))
        expected = {x: y for x, y in direct.items() if plus(x, n) in graph}
        s.check(g, 'composition actual domain ' + str((m, n)), composite == expected)
    # Endpoint returns to its source, but the missing intermediate makes
    # a shift followed by its inverse fail to equal the total identity.
    edge = (1, 0, 0)
    s.check(g, 'strict partial-domain counterexample',
            edge not in compose(partial_shift(box, (-1, 0, 0)), partial_shift(box, (1, 0, 0)))
            and partial_shift(box, ZERO)[edge] == edge)
    pairs = 0
```

```
    for root in box:
        predecessor = {root: None}
        distance = {root: 0}
        queue = deque([root])
        while queue:
            v = queue.popleft()
            for w in graph[v]:
                if w not in distance:
                    predecessor[w] = v
                    distance[w] = distance[v] + 8
                    queue.append(w)
        for target in box:
            path = [target]
            while path[-1] != root:
                path.append(predecessor[path[-1]])
            path.reverse()
            word = {root: root}
            for u, v in zip(path, path[1:]):
                directed_unit = compose({v: v}, compose(partial_shift(box, minus(v, u)), {u: u}))
                word = compose(directed_unit, word)
            s.check(g, 'path matrix unit ' + str((root, target)), word == {root: target})
            s.check(g, 'weighted distance ' + str((root, target)),
                    distance[target] == 8 * sum(abs(x - y) for x, y in zip(root, target)))
            pairs += 1
        for u in graph:
            for v in graph[u]:
                s.check(g, 'distance dual witness ' + str((root, u, v)),
                        abs(distance[u] - distance[v]) <= 8)
    star = [frozenset()] + [frozenset((p,)) for p in box]
    s.check(g, '28-state F12 star', len(star) == 28 and all(len(v) < 2 for v in star))
    star_edges = {v: {w for w in star if len(v ^ w) == 1} for v in star}
    leaf_distance = {star[1]: 0}
    queue = deque([star[1]])
    while queue:
        v = queue.popleft()
        for w in star_edges[v]:
            if w not in leaf_distance:
                leaf_distance[w] = leaf_distance[v] + 1
                queue.append(w)
    q, n, lambda_squared = 4 ** 27, 27 * 4 ** 27, 2 * 4 ** 27
    # Distinct orthogonal tensor-state projectors give HS norm squared 2;
    # the independent uniform cell factor contributes 1/27. Substitute
    # into the article's proved trace-Dirac distance formula, squared.
    hs_squared = Q(2, 27)
    spectral_squared = n * hs_squared / lambda_squared
    s.check(g, 'star leaf metric differs from discrete spectral metric',
            leaf_distance[star[2]] == 2 and spectral_squared == 1)
    s.values['partial_shifts'] = {'integer_shifts': len(shifts), 'direction_pair_products': 36,
        'constructed_matrix_units': pairs, 'graph_weight': 8, 'admissible_star_vertices': 28,
        'regional_tensor_dimension': 4 ** 27, 'full_matrix_dimension': 27 * 4 ** 27,
        'enormous_tensor_matrix_materialized': False}


def standard_form_samples(s):
    g = 'standard_form_samples'
    for n in (2, 3):
        identity = eye(n)
        vec_i = [Q(i == j) for j in range(n) for i in range(n)]
        projection = [[x * y / n for y in vec_i] for x in vec_i]
        s.check(g, 'orthogonal trace projection ' + str(n),
                multiply(projection, projection) == projection and transpose(projection) == projection)
        for i, j in product(range(n), repeat=2):
            a = matrix_unit(n, i, j)
            left = kron(identity, a)
            comm = madd(multiply(projection, left), multiply(left, projection), -1)
            a0 = madd(a, scaled(identity, trace(a) / n), -1)
            expected = [[-a0[r][c]] for c in range(n) for r in range(n)]
            s.check(g, 'scalar-kernel-on-I basis ' + str((n, i, j)),
```

```
                    multiply(comm, [[v] for v in vec_i]) == expected)
            for k, l in product(range(n), repeat=2):
                right = kron(transpose(matrix_unit(n, k, l)), identity)
                s.check(g, 'left right commute basis ' + str((n, i, j, k, l)),
                        multiply(left, right) == multiply(right, left))
        a = [[Q((i + 1) * (j + 1) + (i == j)) for j in range(n)] for i in range(n)]
        left = kron(identity, a)
        comm = madd(multiply(projection, left), multiply(left, projection), -1)
        a0 = madd(a, scaled(identity, trace(a) / n), -1)
        variance = trace(multiply(a0, a0)) / n
        s.check(g, 'commutator norm polynomial ' + str(n), variance > 0
                and transpose(comm) == scaled(comm, -1)
                and multiply(multiply(comm, comm), comm) == scaled(comm, -variance))
    z = [[1, 0], [0, -1]]
    vec_i = [1, 0, 0, 1]
    projection = [[Q(x * y, 2) for y in vec_i] for x in vec_i]
    left, right = kron(eye(2), z), kron(z, eye(2))
    comm = madd(multiply(projection, left), multiply(left, projection), -1)
    double = madd(multiply(comm, right), multiply(right, comm), -1)
    s.check(g, 'first order condition explicitly fails',
            multiply(double, [[x] for x in vec_i]) == [[2 * x] for x in vec_i])
    s.values['spectral_scope'] = {'standard_form_dimensions': [2, 3],
        'exact_real_matrix_instances': True, 'general_complex_theorem_proved_here': False,
        'first_order_condition_claimed': False}


def hamiltonian_response_samples(s):
    g = 'hamiltonian_response_samples'
    for bridge, z in product((1, 2), (Q(3), Q(4), Q(7, 2))):
        h = [[0, 1, 0], [1, bridge, 1], [0, 1, 0]]
        s.check(g, 'self-adjoint directional motif ' + str((bridge, z)), transpose(h) == h)
        full = inverse(madd(scaled(eye(3), z), h, -1))
        terminal = [[full[i][j] for j in (0, 2)] for i in (0, 2)]
        coupling = [[Q(1, 1) / (z - bridge)] * 2 for _ in range(2)]
        reduced = madd(scaled(eye(2), z), coupling, -1)
        s.check(g, 'terminal Schur resolvent ' + str((bridge, z)), terminal == inverse(reduced))
        s.check(g, 'typed scalar transfer ' + str((bridge, z)),
                coupling[0][1] == inverse([[z - bridge]])[0][0])
    s.check(g, 'unequal directional transfer at common parameter', Q(1, 3 - 1) != Q(1, 3 - 2))
    energies = (0, 1, 2, 3)
    dipoles = [madd(matrix_unit(4, 0, 1), matrix_unit(4, 1, 0)),
               madd(matrix_unit(4, 0, 2), matrix_unit(4, 2, 0)),
               [[Q(0)] * 4 for _ in range(4)]]
    def reciprocal(real, imag):
        denominator = real * real + imag * imag
        if not denominator:
            raise DomainError('spectral pole')
        return real / denominator, -imag / denominator
    for eta, i, j in product((Q(0), Q(1, 3), Q(1), Q(2)), range(3), range(3)):
        answer = (Q(0), Q(0))
        for k, gap in enumerate(energies):
            for coefficient, sign in ((dipoles[i][0][k] * dipoles[j][k][0], -1),
                                      (dipoles[j][0][k] * dipoles[i][k][0], 1)):
                # Zero numerator terms vanish identically before taking
                # the eta=0 boundary limit; no 0/0 is evaluated.
                if coefficient:
                    term = reciprocal(Q(gap), sign * eta)
                    answer = plus(answer, tuple(coefficient * x for x in term))
        expected = Q(0)
        if i == j == 0:
            expected = 2 / (1 + eta * eta)
        elif i == j == 1:
            expected = 4 / (4 + eta * eta)
        s.check(g, 'exact imaginary-frequency spectral sum ' + str((eta, i, j)),
                answer == (expected, 0))
    s.check(g, 'nonzero unequal static response', (Q(2), Q(4, 4), Q(0)) == (2, 1, 0))
    # Native-factor channel example in the explicitly declared tensor
    # order |00>,|01>,|10>,|11> (distinct from F11's displayed order).
```

```
    v = [[1, 0], [0, 0], [0, 1], [0, 0]]
    x = [[0, 1], [1, 0]]
    w = kron(x, eye(2))
    s.check(g, 'encoded channel isometry', multiply(transpose(v), v) == eye(2))
    s.check(g, 'encoded channel unitary intertwining', multiply(w, v) == multiply(v, x))
    s.check(g, 'unitary target channel', multiply(transpose(w), w) == eye(4))
    s.values['hamiltonian_response_scope'] = {'motif_resolvent_samples': 6,
        'imaginary_frequency_parameters': ['0', '1/3', '1', '2'],
        'spectral_matrix_entries_checked': 36, 'causal_integral_evaluated': False,
        'static_values': [2, 1, 0], 'native_channel_isometry': [4, 2],
        'physical_hamiltonian_or_channel_implementation_claimed': False}


def words(n):
    return tuple(product((0, 1), repeat=n))


def hamming(a, b):
    return sum(x != y for x, y in zip(a, b))


def encode_block(kind, bit):
    return (bit,) * 3 if kind == 'P' else (bit, bit, 1 - bit, 1 - bit)


def block_domain(kind):
    return tuple(w for w in words(3 if kind == 'P' else 4)
                 if min(hamming(w, encode_block(kind, b)) for b in (0, 1)) <= 1)


def decode_block(kind, block, bypass_guard=False):
    expected = 3 if kind == 'P' else 4
    if len(block) != expected or any(b not in (0, 1) for b in block):
        raise DomainError('bit block')
    corrected = block if kind == 'P' else block[:2] + tuple(1 - b for b in block[2:])
    if 2 * sum(corrected) == len(corrected) and not bypass_guard:
        raise DomainError('strict majority tie')
    return int(2 * sum(corrected) > len(corrected))


def encode_state(kind, state):
    return tuple(encode_block(kind, b) for b in state)


def decode_state(kind, state):
    if len(state) != 2:
        raise DomainError('two blocks required')
    return tuple(decode_block(kind, b) for b in state)


def action(state):
    m, r = state
    return m, r ^ m


def laplacian(vertices, edges):
    vertices = list(vertices)
    index = {v: i for i, v in enumerate(vertices)}
    k = [[Q(0)] * len(vertices) for _ in vertices]
    for u, v, conductance in edges:
        if u == v or conductance <= 0 or u not in index or v not in index:
            raise DomainError('passive edge')
        i, j = index[u], index[v]
        k[i][i] += conductance
        k[j][j] += conductance
        k[i][j] -= conductance
        k[j][i] -= conductance
    return k
```

```
def dtn(vertices, edges, boundary):
    """Exact Schur complement; singular interiors reject, never pseudoinvert."""
    vertices = list(vertices)
    if len(set(boundary)) != len(boundary) or any(v not in vertices for v in boundary):
        raise DomainError('boundary')
    internal = [v for v in vertices if v not in boundary]
    order = list(boundary) + internal
    k = laplacian(order, edges)
    b = len(boundary)
    bb = [row[:b] for row in k[:b]]
    if not internal:
        return bb
    bi = [row[b:] for row in k[:b]]
    ii = [row[b:] for row in k[b:]]
    return madd(bb, multiply(multiply(bi, inverse(ii)), transpose(bi)), -1)


def component_network(kind, conductance, prefix):
    names = [prefix + str(i) for i in range(3 if kind == 'P' else 4)]
    pairs = [(0, 1), (1, 2)] if kind == 'P' else [(0, 1), (1, 2), (2, 3), (3, 0)]
    edges = [(names[u], names[v], conductance * (2 if kind == 'P' else 1)) for u, v in pairs]
    return names, edges, [names[0], names[2]]


def components_and_contexts(s):
    g = 'components_and_contexts'
    for kind, expected in (('P', 64), ('L', 100)):
        domain = block_domain(kind)
        full = tuple(product(domain, repeat=2))
        s.check(g, 'full register domain ' + kind, len(full) == expected)
        for bit in (0, 1):
            for w in domain:
                if hamming(w, encode_block(kind, bit)) <= 1:
                    s.check(g, 'one-error correction ' + str((kind, bit, w)),
                            decode_block(kind, w) == bit)
        for state in words(2):
            encoded = encode_state(kind, state)
            s.check(g, 'encoded realization ' + str((kind, state)),
                    decode_state(kind, encoded) == state
                    and encode_state(kind, action(decode_state(kind, encoded))) == encode_state(kind,
     action(state)))
        for z in full:
            decoded = decode_state(kind, z)
            target = encode_state(kind, action(decoded))
            s.check(g, 'exact quotient ' + str((kind, z)), target in full
                    and decode_state(kind, target) == action(decoded))
            # Repetition checks an actual invariant image, not merely DTE=u.
            second = encode_state(kind, action(decode_state(kind, target)))
            s.check(g, 'two-step execution ' + str((kind, z)),
                    second == encode_state(kind, decoded))
    for w in words(4):
        if w not in block_domain('L'):
            s.reject(g, 'excluded L word ' + str(w),
                     lambda w=w: decode_block('L', w, s.mutation == 'decoder_guard'),
                     DomainError, 'strict majority tie')
    s.check(g, 'nonidentity control', action((1, 0)) == (1, 1))
    # Test one-channel equality for several exact positive conductances.
    for conductance in (Q(1, 3), Q(1), Q(2), Q(7, 2)):
        responses = []
        for kind in ('P', 'L'):
            vertices, edges, boundary = component_network(kind, conductance, 'c')
            if s.mutation == 'cycle_conductance' and kind == 'L':
                u, v, w = edges[0]
                edges[0] = (u, v, w + 1)
            response = dtn(vertices, edges, boundary)
            responses.append(response)
```

```
            s.check(g, 'terminal matrix ' + str((kind, conductance)),
                    response == [[conductance, -conductance], [-conductance, conductance]])
        s.check(g, 'path-cycle equality ' + str(conductance), responses[0] == responses[1])
    # Two channels and a fixed family of genuine multi-terminal passive
    # contexts. Terminals of the module are internal to the assembly.
    # Four P/L replacements are checked for each control and context.
    context_count = 0
    for control, scale in product(words(2), (Q(1, 3), Q(1), Q(5, 2))):
        responses = []
        for kinds in product(('P', 'L'), repeat=2):
            vertices = ['e0', 'e1', 'e2']
            edges = []
            ports = []
            for channel, (kind, bit) in enumerate(zip(kinds, control)):
                vs, es, bs = component_network(kind, Q(1 + bit), 'c' + str(channel) + '.')
                vertices += vs
                edges += es
                ports += bs
            context = [('e0', ports[0], scale), ('e1', ports[1], 2 * scale),
                       ('e0', ports[2], 3 * scale), ('e2', ports[3], 4 * scale),
                       (ports[1], ports[2], scale / 2), ('e1', 'e2', scale / 3)]
            responses.append(dtn(vertices, edges + context, ['e0', 'e1', 'e2']))
            # Eliminate module first and substitute its terminal edges.
            reduced = [(ports[0], ports[1], Q(1 + control[0])),
                       (ports[2], ports[3], Q(1 + control[1]))] + context
            s.check(g, 'Schur elimination order ' + str((control, scale, kinds)),
                    responses[-1] == dtn(['e0', 'e1', 'e2'] + ports, reduced, ['e0', 'e1', 'e2']))
        s.check(g, 'context replacements ' + str((control, scale)),
                all(r == responses[0] for r in responses[1:]))
        context_count += 1
    s.reject(g, 'floating disconnected interior',
             lambda: dtn([0, 1, 2], [(0, 1, Q(1))], [0]), DomainError, 'singular interior')
    s.check(g, 'P and L not graph isomorphic', (3, 2, 0) != (4, 4, 1))
    s.values['components'] = {'P_two_block_states': 64, 'L_two_block_states': 100,
        'excluded_L_blocks': 6, 'positive_conductances': ['1/3', '1', '2', '7/2'],
        'two_channel_contexts': context_count, 'assemblies': 4 * context_count,
        'arbitrary_passive_context_theorem_proved_here': False}


def complement(w):
    return tuple(1 - b for b in w)


def normal(w, a, d, omit_complement=False):
    out = w[::-1] if d else w
    return complement(out) if a and not omit_complement else out


def codon(w):
    alphabet = {(0, 0): 'A', (0, 1): 'C', (1, 0): 'G', (1, 1): 'U'}
    return ''.join(alphabet[w[i:i + 2]] for i in (0, 2, 4))


M0 = ((1, 0, 0, 1, 1, 0), (1, 0, 0, 1, 1, 1))


def initialization(w):
    if w not in M0:
        raise DomainError('unassigned address')
    return 1 - w[-1], 1


def dispatch(w, a, d):
    return 'prep', initialization(normal(w, a, d))


def orbit_partition(states, actions):
    remaining = set(states)
```

```
    sizes = []
    while remaining:
        seed = min(remaining)
        orbit = {seed}
        queue = [seed]
        while queue:
            state = queue.pop()
            for act in actions:
                other = act(state)
                if other not in orbit:
                    orbit.add(other)
                    queue.append(other)
        remaining -= orbit
        sizes.append(len(orbit))
    return sizes


def genetics_and_preparation(s):
    g = 'genetics_and_preparation'
    raw = words(6)
    addressed = tuple(product(raw, (0, 1), (0, 1), ('AAA',)))
    for w, a, d, carrier in addressed:
        n = normal(w, a, d, s.mutation == 'normalization')
        s.check(g, 'reference invariance ' + str((w, a, d)),
                n == normal(complement(w), a ^ 1, d, s.mutation == 'normalization')
                == normal(w[::-1], a, d ^ 1, s.mutation == 'normalization'))
        orbit = {(normal(w, f, r), a ^ f, d ^ r, carrier)
                 for f, r in product((0, 1), repeat=2)}
        zeros = [state for state in orbit if state[1:3] == (0, 0)]
        s.check(g, 'unique fully normalized representative ' + str((w, a, d)),
                zeros == [(n, 0, 0, 'AAA')] and len(orbit) == 4)
    full = orbit_partition(addressed, (
        lambda x: (complement(x[0]), x[1] ^ 1, x[2], x[3]),
        lambda x: (x[0][::-1], x[1], x[2] ^ 1, x[3])))
    one = orbit_partition(tuple(product(raw, (0, 1))), (
        lambda x: (complement(x[0]), x[1] ^ 1), lambda x: (x[0][::-1], x[1])))
    none = orbit_partition(raw, (complement, lambda w: w[::-1]))
    s.check(g, 'three distinct quotient counts', (len(full), len(one), len(none)) == (64, 36, 20))
    fixes = [sum(normal(w, a, d) == w for w in raw)
             for a, d in ((0, 0), (1, 0), (0, 1), (1, 1))]
    s.check(g, 'unreferenced fixed-point counts', fixes == [64, 0, 8, 8])
    tau = str.maketrans('CG', 'GC')
    for w in raw:
        s.check(g, 'transported reversal ' + str(w), codon(w[::-1]) == codon(w)[::-1].translate(tau))
    s.check(g, 'ordinary reversal is not transported reversal',
            codon(M0[0][::-1]) != codon(M0[0])[::-1])
    assigned = 0
    for w, a, d, carrier in addressed:
        if normal(w, a, d) not in M0:
            s.reject(g, 'unassigned preparation ' + str((w, a, d)),
                     lambda w=w, a=a, d=d: dispatch(w, a, d), DomainError, 'unassigned address')
            continue
        assigned += 1
        symbol, initial = dispatch(w, a, d)
        s.check(g, 'preparation typed symbol ' + str((w, a, d)),
                symbol == 'prep' and initial in words(2) and carrier == 'AAA')
    for n in M0:
        initial = initialization(n)
        for kind in ('P', 'L'):
            full = tuple(product(block_domain(kind), repeat=2))
            for y in words(2):
                s.check(g, 'encoded preparation ' + str((n, kind, y)),
                        encode_state(kind, initial) in full and decode_state(kind, encode_state(kind,
    initial)) == initial)
            for index, z in enumerate(full):
                source_reset = initial
                component_reset = encode_state(kind, initial)
                s.check(g, 'quotient preparation ' + str((n, kind, index)),
```

```
                        decode_state(kind, component_reset) == source_reset and decode_state(kind, z) in
      words(2))
    s.check(g, 'two stated initialization values', tuple(map(initialization, M0)) == ((1, 1), (0, 1)))
    s.check(g, 'init differs from subsequent operation', action(initialization(M0[0])) == (1, 0))
    # Ala is the manuscript-supplied coarse observation, not an independent
    # computational validation of the biological standard genetic code.
    coarse = {M0[0]: 'Ala', M0[1]: 'Ala'}
    s.check(g, 'supplied coarse observation loses initialization distinction',
            coarse[M0[0]] == coarse[M0[1]] and initialization(M0[0]) != initialization(M0[1]))
    for retained_values, desired_values in product(product((0, 1), repeat=2),
                                                   product(words(2), repeat=2)):
        retained, desired = dict(zip(M0, retained_values)), dict(zip(M0, desired_values))
        fiber_constant = all(retained[x] != retained[y] or desired[x] == desired[y]
                             for x, y in product(M0, repeat=2))
        factors = any(all(candidate[retained[x]] == desired[x] for x in M0)
                      for candidate in product(words(2), repeat=2))
        s.check(g, 'finite factorization criterion ' + str((retained_values, desired_values)),
                factors == fiber_constant)
    s.values['genetics'] = {'fully_addressed_states': 256, 'fixed_carrier': 'AAA',
        'fully_referenced_orbits': 64, 'complement_only_orbits': 36,
        'unreferenced_orbits': 20, 'assigned_preparation_states': assigned,
        'selected_external_codons': list(map(codon, M0)), 'resident_carrier_changed': False}


def membrane_and_material(s):
    g = 'membrane_and_material'
    for m, xi, v in product((0, 1), (Q(0), Q(1, 3), Q(9, 10), Q(99, 100)),
                            (Q(100), Q(200), Q(300))):
        shared = (m, xi, v)
        lattice = (shared, 'fixed native lattice')
        genetic = (shared, M0[0] if m else M0[1], 'AAA')
        functional = (shared, encode_state('P', (m, 1)))
        membrane = (shared, 1 - xi, 1 - xi, xi, xi, 500 + xi, 1, 20, 0, v + 1)
        s.check(g, 'shared readings ' + str(shared),
                all(x[0] == shared for x in (lattice, genetic, functional, membrane)))
        dxi = (1 + m) * (1 - xi * xi)
        dv = (500 + xi) / 100 * (20 / v - Q(1, 10))
        s.check(g, 'constitutive field identity ' + str(shared),
                dxi == (1 + m) * (1 + membrane[3]) * membrane[1])
        s.check(g, 'two material invariants ' + str(shared),
                membrane[1] + membrane[3] == membrane[2] + membrane[4] == 1)
        s.check(g, 'stationary water criterion ' + str(shared), (dv == 0) == (v == 200))
    # Rational power-series bracket of e. The omitted tail is bounded by
    # a geometric series: consecutive term ratios are at most 1/(N+2).
    terms = 10
    e_lower = sum(Q(1, factorial(k)) for k in range(terms + 1))
    e_upper = e_lower + Q(1, factorial(terms + 1)) / (1 - Q(1, terms + 2))
    s.check(g, 'series bracket stronger than stated constants', Q(8, 3) < e_lower < e_upper < 3)
    s.check(g, 'GCG tanh2 threshold certificate', e_lower ** 4 > 19)
    s.check(g, 'GCU tanh1 upper certificate', e_upper ** 2 < 9)
    s.check(g, 'feedback ablation certificate', e_upper ** 2 < 10)
    s.check(g, 'capsule area-volume strict certificate', 36 * Q(22, 7) * 201 ** 2 < 500 ** 3)
    s.check(g, 'capsule radius lower certificate', Q(402, 501) > Q(4, 5))
    s.check(g, 'bounding ball radical certificate', Q(7, 4) ** 2 > 3 and 7 ** 2 > 43
            and 8 * Q(7, 4) + Q(7, 2) < 18)
    s.check(g, 'positive clearance', Q(18, 100) + Q(2, 100) == Q(1, 5) < Q(4, 5))
    # The separate seven-cell specialization: arithmetic certificates of
    # its comparison proof, not a floating-point threshold simulation.
    epsilon = Q(1, 100)
    pmax = 3 + 2 * epsilon
    jmax = 10 + 20 * pmax / (1 + pmax)
    block_bound = (40 * jmax).numerator // (40 * jmax).denominator
    mu = Q(30, 31) * Q(100, 101) * Q(19, 20)
    F = lambda p: mu * (1 + Q(99, 50) * p / (1 + p)) - p
    delta = F(Q(81, 50))
    hit = Q(161, 100) / delta
    s.check(g, 'count rate comparison constants', (pmax, jmax, block_bound) == (Q(151, 50), Q(5030, 201),
      1000))
```

```
    s.check(g, 'count availability and positive endpoint', mu == Q(2850, 3131)
            and delta == Q(8298609, 20508050) and 0 < delta < F(0))
    # Exact second derivative sign certificate at finite rational samples;
    # the analytic concavity proof is explicitly not inferred from samples.
    for p in (Q(0), Q(1, 2), Q(1), Q(81, 50)):
        s.check(g, 'concavity rational sample ' + str(p), -2 * mu * Q(99, 50) / (1 + p) ** 3 < 0)
    s.check(g, 'count hitting time', hit == Q(66035921, 16597218) and hit < 4)
    s.check(g, 'count threshold floor margin', Q(161, 100) - epsilon == Q(8, 5))
    s.check(g, 'count ablation upper bound', 1 + 2 * epsilon == Q(51, 50))
    s.check(g, 'water lower inward certificate', Q(24, 100) - Q(1, 10) == Q(7, 50) > 0)
    s.check(g, 'water upper inward certificate', Q(54, 590) + Q(4, 590 ** 2) - Q(1, 10) == Q(-1473,
     174050) < 0)
    energy = 160 - 100 - 4 * (2 + pmax) - 18
    s.check(g, 'energy reserve comparison', energy == Q(548, 25) > 0)
    for B, K in product(range(1001), (1, 2)):
        b, c, S = 130 - Q(B, 10), 30 - Q(B, 50), Q(B, 10)
        A, U, N = 518 + 2 * K + Q(8, 5) * S, 8 + K, 24 + Q(3, 10) * S
        inside = (81315 if s.mutation == 'carbon_balance' else 81314) + 44 * K + 264 * B
        outside = 22 * (13000 - 10 * B) + 22 * (3000 - 2 * B) + 44 * (2 - K)
        s.check(g, 'count balance ' + str((B, K)), inside + outside == 433402)
        s.check(g, 'stock area volume envelopes ' + str((B, K)),
                b >= 30 and c >= 10 and 0 <= S <= 100
                and 520 <= A <= 682 and 24 <= N <= 54
                and 90 <= 100 - U and 600 - U <= 591)
    formulas = {'D2': (22, 20, 0, 4), 'NH3': (0, 3, 1, 0),
                'Gly4': (8, 14, 4, 5), 'Gly': (2, 5, 1, 2),
                'R': (20, 16, 0, 2), 'H2': (0, 2, 0, 0), 'H2O': (0, 2, 0, 1)}
    total = lambda terms: tuple(sum(k * formulas[f][i] for k, f in terms) for i in range(4))
    left = total(((12, 'D2'), (10, 'NH3')))
    right = total(((2, 'Gly4'), (2, 'Gly'), (10, 'R'), (2, 'D2'), (10, 'H2'), (6, 'H2O')))
    s.check(g, 'productive block element balance', left == right == (264, 270, 10, 48))
    s.check(g, 'seven-cell and twenty-seven-cell inventories remain distinct',
            (27 - 7) * 26 == 520 and 27 * 26 == 486 + 216)
    s.values['membrane'] = {'shared_state_samples': 24,
        'e_rational_lower': str(e_lower), 'e_rational_upper': str(e_upper),
        'count_carbon_total': 433402, 'count_stock_cases': 2002,
        'count_block_upper_bound': 1000, 'count_hitting_time_upper': str(hit),
        'count_energy_lower_bound': str(energy), 'count_horizon': 4,
        'count_native_cells': 7, 'main_model_native_cells': 27,
        'hybrid_trajectory_simulated': False, 'atom_resolved_reaction_verified': False}


MUTATIONS = ('cycle_conductance', 'normalization', 'carbon_balance', 'decoder_guard')


def run(mutation=None):
    suite = Suite(mutation)
    box, graph = native_and_recognition(suite)
    shifts_and_metrics(suite, box, graph)
    standard_form_samples(suite)
    hamiltonian_response_samples(suite)
    components_and_contexts(suite)
    genetics_and_preparation(suite)
    membrane_and_material(suite)
    return {'status': 'PASS', 'article': 85,
        'title': 'A finite axiomatic framework for the interior chrysene lattice and protocell
     organization',
        'exact_check_count': sum(suite.counts.values()),
        'expected_rejection_count': suite.expected_rejections,
        'checks_by_group': dict(sorted(suite.counts.items())),
        'ordered_check_trace_sha256': suite.trace.hexdigest(),
        'values': suite.values,
        'scope': 'Exact finite enumerations, rational matrix identities and comparison constants. '
                 'Not proofs for all finite regions or dimensions; not ODE existence/uniqueness, '
                 'arbitrary passive-context equivalence, hybrid event timing, chemical synthesis, '
                 'full molecular realization, infinite lattice, or measured biological response.'}
```

```
def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('--output', type=Path, help='write deterministic JSON report')
    parser.add_argument('--mutate', choices=MUTATIONS, help='intentional fault: must exit nonzero')
    args = parser.parse_args()
    try:
        report = run(args.mutate)
    except (VerificationFailure, DomainError) as error:
        print('FAIL: ' + str(error), file=sys.stderr)
        return 1
    if args.mutate:
        print('FAIL: intentional mutation escaped detection', file=sys.stderr)
        return 2
    serialized = json.dumps(report, indent=2, sort_keys=True) + '\n'
    if args.output:
        args.output.write_text(serialized, encoding='utf-8')
    print('PASS: ' + str(report['exact_check_count']) + ' exact Article 85 checks.')
    return 0


if __name__ == '__main__':
    sys.exit(main())
```

6 Fresh execution and negative-control harness

run_verification.py

```
#!/usr/bin/env python3
"""Fresh-copy reproducibility and falsification harness for Article 85.

Both this file and verify_article85.py are standalone Python standard-library
payloads. Run this file in the same directory as verify_article85.py. Outputs
are written only to --output-dir (default: a new ./evidence directory).
"""
import argparse
import ast
from hashlib import sha256
import json
from pathlib import Path
import shutil
import subprocess
import sys
import tempfile


EXPECTED_COUNTS = {
    'components_and_contexts': 437,
    'genetics_and_preparation': 1246,
    'hamiltonian_response_samples': 59,
    'membrane_and_material': 4122,
    'native_and_recognition': 545,
    'shifts_and_metrics': 4913,
    'standard_form_samples': 115,
}


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


def digest(path):
    return sha256(path.read_bytes()).hexdigest()
```

```

def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('--output-dir', type=Path, default=Path('evidence'))
    args = parser.parse_args()
    source = Path(__file__).resolve().with_name('verify_article85.py')
    require(source.is_file(), 'Missing standalone verifier beside this harness')
    tree = ast.parse(source.read_text(encoding='utf-8'))
    require(not any(isinstance(node, ast.Assert) for node in ast.walk(tree)),
            'Verifier contains optimization-sensitive assert statement')
    own_tree = ast.parse(Path(__file__).read_text(encoding='utf-8'))
    require(not any(isinstance(node, ast.Assert) for node in ast.walk(own_tree)),
            'Harness contains optimization-sensitive assert statement')
    output = args.output_dir.resolve()
    require(not output.exists() or (output.is_dir() and not any(output.iterdir())),
            'Refusing nonempty evidence destination; choose a new --output-dir')
    output.mkdir(parents=True, exist_ok=True)
    transcript = []
    runs = []
    expected_faults = {
        'cycle_conductance': 'components_and_contexts: terminal matrix',
        'normalization': 'genetics_and_preparation: reference invariance',
        'carbon_balance': 'membrane_and_material: count balance',
        'decoder_guard': 'components_and_contexts: failed to reject excluded L word',
    }
    reports = []
    with tempfile.TemporaryDirectory(prefix='article85-fresh-') as name:
        fresh = Path(name)
        shutil.copyfile(source, fresh / source.name)
        require(set(p.name for p in fresh.iterdir()) == {source.name},
                'Fresh workspace has unexpected dependencies')
        for optimized in (False, True):
            mode = 'optimized' if optimized else 'normal'
            command = [sys.executable, '-I'] + (['-O'] if optimized else [])
            command += [source.name, '--output', 'report-' + mode + '.json']
            process = subprocess.run(command, cwd=fresh, text=True,
                                     capture_output=True, timeout=120)
            transcript.append('$ python3 -I ' + ('-O ' if optimized else '')
                              + 'verify_article85.py --output report-' + mode + '.json\n'
                              + process.stdout + process.stderr)
            require(process.returncode == 0, mode + ' fresh run failed: ' + process.stderr)
            path = fresh / ('report-' + mode + '.json')
            report = json.loads(path.read_text(encoding='utf-8'))
            require(report['status'] == 'PASS'
                    and report['checks_by_group'] == EXPECTED_COUNTS
                    and report['exact_check_count'] == sum(EXPECTED_COUNTS.values())
                    and report['expected_rejection_count'] == 347,
                    'Invalid result report')
            reports.append(path.read_bytes())
            shutil.copyfile(path, output / path.name)
            runs.append({'mode': mode, 'exit_code': process.returncode,
                         'exact_check_count': report['exact_check_count'],
                         'report_sha256': digest(path)})
            for mutation, prefix in expected_faults.items():
                command = [sys.executable, '-I'] + (['-O'] if optimized else [])
                command += [source.name, '--mutate', mutation]
                process = subprocess.run(command, cwd=fresh, text=True,
                                         capture_output=True, timeout=120)
                transcript.append('$ python3 -I ' + ('-O ' if optimized else '')
                                  + 'verify_article85.py --mutate ' + mutation + '\n'
                                  + process.stdout + process.stderr)
                require(process.returncode == 1, mode + ' mutation not detected: ' + mutation)
                require(process.stderr.startswith('FAIL: ' + prefix),
                        mode + ' mutation failed for an unexpected reason: ' + mutation)
                runs.append({'mode': mode, 'mutation': mutation,
                             'exit_code': process.returncode,
                             'diagnostic': process.stderr.strip()})
```

```
        require(reports[0] == reports[1], 'Normal and optimized reports differ')
    evidence = {
        'status': 'PASS', 'python_version': sys.version.split()[0],
        'verifier_sha256': digest(source), 'harness_sha256': digest(Path(__file__)),
        'fresh_workspace_files_at_start': ['verify_article85.py'],
        'isolated_interpreter': True, 'assert_statements_in_verifier': 0,
        'assert_statements_in_harness': 0,
        'normal_and_optimized_json_byte_identical': True,
        'successful_runs': 2, 'detected_mutant_runs': 8, 'runs': runs,
        'scope': 'Reproducibility and deliberate-defect sensitivity of the finite Python checks; '
                 'not certification of a Python interpreter, mathematical generalization, or chemistry.'
    }
    (output / 'test_evidence.json').write_text(json.dumps(evidence, indent=2, sort_keys=True) + '\n',
     encoding='utf-8')
    (output / 'test_transcript.txt').write_text('\n'.join(transcript), encoding='utf-8')
    print('PASS: 2 fresh isolated runs agree; 8 deliberate-defect runs rejected.')
    print('Evidence: ' + str(output))


if __name__ == '__main__':
    try:
        main()
    except RuntimeError as error:
        print('FAIL: ' + str(error), file=sys.stderr)
        sys.exit(1)
```

7 Complete recovery helper

```
#!/usr/bin/env python3
"""Recover the complete pinned Article 85 runtime; standard library only."""
from pathlib import Path, PurePosixPath
import argparse, base64, hashlib, io, json, stat, zipfile
EXPECTED_SHA256 = "8e78317bd6c063ddb59de9c4db658e7ed088bc97d78fb5728d443651ac1b03d9"
DEFAULT_PAYLOAD = "article85_verification_payload.b64"

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

def recover(payload, output):
    encoded = payload.read_bytes()
    raw = base64.b64decode(b"".join(encoded.split()), validate=True)
    require(hashlib.sha256(raw).hexdigest() == EXPECTED_SHA256,
            "Payload archive digest mismatch")
    with zipfile.ZipFile(io.BytesIO(raw)) as z:
        infos = z.infolist()
        names = [i.filename for i in infos]
        require(len(names) == len(set(names)), "Duplicate archive member")
        for i in infos:
            p = PurePosixPath(i.filename)
            require(not p.is_absolute() and ".." not in p.parts
                    and str(p) == i.filename and "\\" not in i.filename
                    and ":" not in i.filename
                    and not i.is_dir(), "Unsafe archive member")
            require(not stat.S_ISLNK(i.external_attr >> 16), "Symbolic link")
            require(not (i.flag_bits & 1), "Encrypted archive member")
        manifest = json.loads(z.read("PAYLOAD_MANIFEST.json"))
        require(manifest["format"] == "article85-portable-payload-v1",
                "Unexpected manifest format")
        files = manifest["files"]
        require(set(names) == set(files) | {"PAYLOAD_MANIFEST.json"},
                "Archive inventory mismatch")
        contents = {}
        for n, row in files.items():
```

```
            data = z.read(n)
            require(len(data) == row["size"], "Size mismatch: " + n)
            require(hashlib.sha256(data).hexdigest() == row["sha256"],
                    "Member digest mismatch: " + n)
            contents[n] = data
        contents["PAYLOAD_MANIFEST.json"] = z.read("PAYLOAD_MANIFEST.json")
    require(not output.exists() or (output.is_dir() and
            not any(output.iterdir())), "Output directory must be empty")
    output.mkdir(parents=True, exist_ok=True)
    for n, data in contents.items():
        dest = output/n
        dest.parent.mkdir(parents=True, exist_ok=True)
        dest.write_bytes(data)
    print("Recovered", len(contents), "verified files into", output)

if __name__ == "__main__":
    ap = argparse.ArgumentParser(description=__doc__)
    ap.add_argument("--payload", type=Path,
                    default=Path(__file__).resolve().parent/DEFAULT_PAYLOAD)
    ap.add_argument("--output-dir", type=Path, required=True)
    args = ap.parse_args()
    recover(args.payload, args.output_dir)
```