Verification Code Analysis

Article 75: Constructive Genetic Code-Table Realization On A Typed Chrysene Carrier

Charles D. Schaper, Ph.D.

Annals of the Chrysene Formalism
Volume 1, Number 4 · Manuscript opening page 1503
23 October 2026

Executed finite verification

Executable	`verify_article75.py`, entirely self-contained
Original modules / pinned artifacts	4 modules + guard audit + combined entrypoint / 17
Registry / addressed states / refinement	20 / 64 / 61
Independent Hall subsets examined	$3 \times 1{,}048{,}576$
Rooted product-identity comparisons	400 comparisons; 20 unique matches
Individual product paths / sulfur path	20 paths, 668 primitives / 174 primitives
Resource-composition witness	19 modules, 1,504 actual atoms, 40 ports
Junction/controller finite cases	4,096 six-letter words; 128 cases, 56 accepted
Additional reset traces	80 traces, 840 successful segments
Additional rejection controls	88 rejected as intended
Runtime / dependencies	Python 3.12.14 / standard library only
Recorded result	**PASS**; normal, `-O`, and environment runs agree

What the pass means

This is a reproducible verification of finite formal graph, registry, routing, and controller witnesses. Counts refer to cases, predicates, or executions, not independent theorems. The program does not certify chemical feasibility, stereoselective synthesis, the precursor-to-carrier atom map, a complete physical apparatus, or independent/unique selection of a genetic code. Those exclusions agree with the controlling manuscript's stated logical scope.

1. Reproduction and provenance

The complete executable source is embedded in Section 8. Save that listing as `verify_article75.py`; alternatively use the identical distributed file. Run:

```
python3 verify_article75.py
python3 -O verify_article75.py --output optimized_results.json
PYTHONOPTIMIZE=2 python3 verify_article75.py --output environment_optimized_results.json
```

The last command uses a POSIX-style environment assignment. Python 3.10 or later suffices. No companion data, installation, network access, randomness, external algebra system, or chemical toolkit is required. The program creates temporary files, executes the supplied definitions, checks their products, removes those temporary files, and writes `results_article75.json`. Only the requested results path is retained. This does not mutate the manuscript tree.

Compiling *this document* with pdfLaTeX, including on Overleaf, only produces the dossier PDF. It does not execute Python or Lean, and needs no shell escape. The `.tex` file has no external `\input`

or code-listing dependency. The Python listing was extracted from the standalone `.tex`, compared byte-for-byte with the executable, and run again. Syntax compilation also passed.

The controlling input is the attached Article 75 manuscript and its packaged certificates, not an earlier draft or a replacement reference implementation. Its original combined checker was first run separately on an isolated copy and exited successfully. The supplement embeds all six original programs without changing their source bytes. It embeds the retention rows and fixed reference-table snapshot as readable structured data. The paired junction input is regenerated from the product module, then required to match its original SHA-256 value. All 17 executable/data/CSV artifacts required for this reproduction are checked against pinned original hashes; those hashes appear in the source and results JSON. A separate `input_manifest.json` records the original document/source inventory as provenance. It is not a claim that later publication metadata, typography, bibliography capitalization, or the new supplement pointer have unchanged hashes.

Executable SHA-256:
`cd049f3ed6bc911f363e6758ca027aca4cf55dae22fd79ebfde1cce8fe338c95`

Recorded results JSON SHA-256:
`7e3f4dbacfb7674bf9bbea4b4986e8c61bd482c5338b0a634cbbb10ca08da029`

2. What is supplied and what is calculated

A successful reconstruction is meaningful only if its mathematical inputs are distinguished from its outputs.

Layer	Supplied model data	Recomputed consequences
Registry	Binary frame action, pair encoding, class naming order, and standard-code reference string	Complete six-bit partition; adjacent-difference invariant; transported actions; 24 letter bijections; matching and refinement counts.
Products	Initial carbon graph and Kekulé assignment; target-shaped retention rows; finite donor stock; explicit target side-chain graphs; neutral capacity convention	Persistent labelled atoms; permitted split/join paths; intermediate capacity and extraction guards; terminal formula/component equality; full remnants and donors.
Routing	Seventeen selected demands, W/M anchors, seed choices, refinement names, dormant/locked cells, local exchange schema	840 distinct representatives; 87 spare triples; legal path through 18 seeded cells to 64 assignments; both endpoint tables.
Selector	Placement and affine coefficients of the AUR, UGR, and AGR modules; strict-positive maximal-score rule	All affine zero/equality breakpoints; both open regimes, the tie wall, four changed codons, and exact alternative routing count.

Layer	Supplied model data	Recomputed consequences
Junction	Six-word family, outgoing register transport, identity maps, controller transition schemas, and successful control/release/halt predicates	All 4,096 words; 256 frame comparisons; 16 framed gates; explicit paired path and stages; finite controller traces and rejected invalid attempts.

The amino-acid targets and genetic-code reference data are not inferred from the number twenty. The seed/refinement/selector locations are table-informed. Expected counts are test oracles, not derived physical parameters. The scalar flags for stereochemical requirements, formal neutrality, carrier architecture, and the count of original precursor cores are not coordinate-level molecular certificates.

The fixed NCBI snapshot records Tables 1 and 2, base order UCAG, page update 2024-09-23, and access date 2026-10-05. The supplement compares all 128 stored internal assignments; it does not perform a live NCBI query or verify historical access metadata. Internal codon assignments are not the separate biological initiation annotations. The source address is `https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi`.

3. Executed coverage and manuscript correspondence

Block	Actual nonempty coverage	Boundary
V1	All 64 six-bit markers, all 64^2 marker pairs, and orbit enumerations for $N = 1, \ldots, 6$; addressed/reference-only quotient enumeration	Finite tests support the general quotient proof, but are not its universal quantifier.
V2	All 24 letter bijections; three 20-vertex incidence graphs; every one of 2^{20} left subsets for each graph; supplied attaining matchings	Table-conditioned compatibility, not a chemical compatibility law.
V3	All 20 product graph witnesses and 174 edits of the paired sulfur path, with full atoms, bonds, remnants, and donor stock	Neutral valence-port bookkeeping, not electronic states or mechanisms.
V4	All 20×20 product/target rooted graph comparisons; complete terminal component partitions; six new malformed certificates rejected	Target graphs are explicit definitions; no stereochemical isomorphism test.
V5	778 round-robin edits on 19 disjoint modules, comprising 18 non-sulfur paths plus one two-carrier sulfur path	One concrete interleaving, not enumeration of every interleaving; the manuscript gives the general proof.
V6	17 demands, 18 seed cells, 840 matchings, 52 propagated codons, 24 completed cells, two exchanges; all affine roots and exact rational regime representatives	Affine order is constant between computed roots. No molecular control trajectory is supplied.

Block	Actual nonempty coverage	Boundary
V7	All 4,096 six-letter words, 256 marker/frame comparisons, 16 framed release gates, all 24 outgoing-letter permutations	These check the specified transport and gate; they do not derive physical polarity.
V8	Original 128 frame/letter/stage/extension cases, 56 accepted; original three-packet traces; 80 additional complete traces with $n = 1, \ldots, 20$	Tapes are AUG–UAA or AUG–GCU–UAA. These are not all tapes or unbounded controller histories.
V9	All 17 pinned artifacts regenerated, all endpoint assignments compared, three product/junction identity projections, exact full paired-path equality	Reproducibility and interface agreement are distinct from an external validation of the assumptions.

Registry and Hall certificates

The WC, RY, and MK graph deficiencies are respectively $2, 1, 0$, giving matching numbers $18, 19, 20$. The supplement's extra Hall check does not call the original matching algorithm. For each subset, it computes the union of neighbor bitmasks, then evaluates $|S| - |N(S)|$. The empty subset is included. A separately checked matching attains the resulting upper bound in each graph. For WC, the displayed maximal-deficiency witness is

$$S = \{2, 4, 6, 8, 12, 13, 17, 18, 20\}, \qquad |S| = 9, \quad |N(S)| = 7.$$

The standard-output refinement has 61 cells. Its only nonsingleton codon cells are $\{AGC, UCG\}$, $\{AGU, UCA\}$, and $\{CUU, UUG\}$. STOP is included in that refinement. The count is not twenty deterministic output classes.

Affine selection and routing freedom

The only internal wall is $u = 1/2$. Before it, the readout is the mitochondrial reference; afterward, it is the standard reference. At the wall, AUA has the tied set $\{I, M\}$, while UGA, AGA, and AGG have empty sense sets. Strict positivity is implemented literally; zero is not counted as a positive score. All root comparisons use rational arithmetic.

The $u = 0$ exchange graph has components of sizes $8, 8, 3, 2$, with distinct labels in each. The computed orbit size is

$$8!\,8!\,3!\,2! = 19{,}508{,}428{,}800.$$

An explicit alternative exchange preserves AUG $\mapsto M$ while yielding neither named endpoint. These facts are positive checks of nonuniqueness, not a failure of the conditional realization statement.

4. Product identities, material accounts, and negative controls

The original builder constructs target graphs and checks formulas. Its unused `rooted_signature` helper is not counted as an executed identity proof. The supplement instead compares every actual terminal heavy graph with each of twenty independently written, conventional rooted target specifications using exact backtracking graph isomorphism. The five distinguished roots are the alpha carbon, carboxyl carbon, amino nitrogen, carbonyl oxygen, and hydroxy oxygen. Vertex elements,

bond orders, adjacency and nonadjacency are preserved. Each of the twenty graphs matches exactly its own target; the equal-formula Ile/Leu pair is explicitly distinguished.

Those specifications remain supplied constitutional targets. The test does not infer stereochemistry or claim that a particular aromatic drawing is a physical pathway. Hydrogen identities and complete product components are checked separately by the full graph/replay predicates. Every individual witness retains all original atoms, and every listed remnant and unused resource remains in the global endpoint. The 20 single-row paths contain 668 primitive edits. The coupled sulfur certificate contains 174 edits, reaches its disulfide bridge after step 8, and its separated sulfur topology after step 10.

For a repertoire, the supplement does not incorrectly add two independent sulfur packets. It uses eighteen unary non-sulfur modules and one coupled module with its shared sulfur pair. Prefixing module identities yields 1,504 actual atoms and 40 ports, with exact inventory

$$C_{400}H_{882}N_{80}O_{140}S_2.$$

A round-robin interleaving executes 778 primitives and reaches the exact disjoint union of the nineteen endpoints. The original local extraction guards were checked before this interleaving; disjoint prefixes provide the noninterference needed to reuse them. Every intermediate global bond/capacity guard is checked. This inventory describes the active local scaffold graphs and finite donor packets. It is not a full atomization of 120 original precursor cores, resident adenosines, or the external carrier context.

The seven original guard tests reject an invalid mask, nonneighbor whole-removal vertex, protected backbone edit, new core adjacency, out-of-support edit, donor-to-remnant bond, and paired protected edit. The supplement adds six malformed-certificate controls (lost atom, changed element, duplicate bond, undeclared primitive, self-loop, and missing final edit), eighty fresh-packet reuse rejections, one wrong outgoing constitution at START, and one deliberately false assertion under optimized compilation. These 88 additional controls are targeted examples, not exhaustive invalid-input testing.

5. Controller scope and trusted conditions

The original finite branch table covers four frames, four outer letters, four initialized routing stages, and optional single-Ala continuation. Of 128 cases, 56 complete successfully under the declared guards. The supplement adds 80 complete traces: $n = 1, \ldots, 20$, two endpoint stages, and two tapes. They execute 6,300 transitions and produce 840 successful Met-first segments, consuming exactly n fresh packets per trace and no preloaded $(n + 1)$st packet.

FORM identifies the outgoing Met graph through the separately checked library; START requires that identity and consumes a unique token. Extension in the executable is only the GCU/Ala fixture. OBSERVE, RELEASE and HALT are distinct transitions, with explicit control-certificate predicates. Their default successful values are assumptions of the formal witness. They are not measured signals, physical release proofs, or molecular reset implementations. Controller tokens are not an atom-by-atom re-execution of every emitted packet. The material composition and controller tests therefore establish their declared interfaces, not an omitted molecular mechanism joining the two.

The manuscript's induction, rather than finite repetition alone, supports arbitrary finite accepted prefixes. A Lean proof has a separate trust boundary and declared coverage in its companion dossier. Passing Python tests or compiling this LaTeX document is not evidence of Lean kernel checking.

6. Assertions, optimization, and interpretation of counts

Legacy scripts use Python `assert`. Running those scripts directly with optimization can remove checks. This supplement does not rely on the caller remembering that restriction. Before execution, it parses each unmodified embedded program and transforms every assertion into an explicit predicate call followed, if false, by an explicit `AssertionError`. It then compiles with `optimize=0`. Assertion messages remain lazy: a message expression is evaluated only for a failed test. A dedicated self-test compiles a deliberately false transformed assertion with `optimize=2`, verifies its rejection, and verifies message-evaluation order.

The original combined entrypoint is also run, including its own child-process reruns. During that call, `PYTHONOPTIMIZE=0` is explicitly set for children and the previous environment value is restored afterward. The standalone normal run, the `-O` run, and the `PYTHONOPTIMIZE=2` run all exited zero and generated byte-identical results. No checks are silently skipped in those tested modes.

Instrumented original source	Predicate evaluations	False on rejection paths
Registry	4,376	0
Product builder	88,852	0
Routing/reachability	364	0
Junction/controller	66,249	1,050
Extraction-guard audit	135,895	13
Combined entrypoint	51	0

These include calls made by the added tests to the original controller and guard functions. False evaluations are intentionally caught invalid examples, not unexpected failed tests. The 2,057 extra explicit supplement predicates all succeed. Child-process evaluations, individual terms inside compound predicates, and each inner Hall subset calculation are not included in the instrumented-source predicate totals. Hence neither those totals nor the case counts should be marketed as a count of proved mathematical claims.

7. Limits and publication use

This dossier verifies the stated nonempty formal construction without replacing its hypotheses by undocumented chemistry. In particular, it does not supply:

- the missing precursor/carrier origin map, full external-context graft, or a common atomistic apparatus;
- reaction feasibility, electronic-state completion, rates, yields, stereoselection, or compatible continuous molecular paths;
- physical docking/readout/release/termination/reset certificates;
- a target-independent derivation of the initialization data or a unique reachable code;
- general analytic proofs of positive path probability, geometric lifting, robustness, or all arbitrary-length runs;
- independent verification of every cited source article or the patent's disputed atom indices.

The result is nevertheless nonvacuous: it reconstructs every packaged formal path, checks explicit forbidden examples, tests material conservation on full labelled graphs, independently identifies all twenty constitutional endpoints, exhausts all relevant Hall subsets, and checks the intended shared interfaces. A later modification of source data or rules requires a new run and new provenance hashes. Editing a status flag is not an alternative to reproducing the checks.

The distributed files are this standalone `.tex`/PDF, the identical Python executable, results JSON, input manifest, and source/data archive. The archive preserves the original program/data bytes for inspection. No companion file is needed to run the Python executable.

8. Complete executable Python source

The following consecutive listings together contain the exact 2950-line executable used for the recorded result. Original source blocks and supplied fixtures are visibly separated from the new wrapper and cross-checks. There is no encoded, compressed, or externally fetched payload. For reliable execution, extract from the `.tex` listing or use the distributed `.py`; PDF text extraction may alter indentation.

```
#!/usr/bin/env python3
"""Self-contained Article 75 finite verification, Python 3.10+ standard library.

Run: python3 verify_article75.py
     python3 -O verify_article75.py --output optimized_results.json
No network, random input, external files, or third-party packages are needed.
All bundled source blocks are the unaltered controlling manuscript programs.
An AST transformation replaces Python assert with explicit checks before their
execution, so optimization cannot erase tests. Fixed outputs are regenerated in
a temporary directory and compared against original SHA-256 values. Additional
data-consuming checks follow. Source data, formulas, targets, initial routing,
affine coefficients, and model guards remain declared inputs, not chemistry.
"""
from pathlib import Path
from collections import Counter
from contextlib import redirect_stdout
from fractions import Fraction
from itertools import product, combinations
import argparse, ast, copy, hashlib, io, json, os, sys, tempfile, types

SOURCES = {}

# BEGIN UNMODIFIED SOURCE: certificates/registry/verify_registry.py
SOURCES['certificates/registry/verify_registry.py'] = r'''#!/usr/bin/env python3
"""Independent, Python-stdlib-only verification of the registry chapter.

Run `python3 verify_registry.py --write` to regenerate the accompanying data,
or `python3 verify_registry.py` to verify the packaged data byte-independently.
No source code from the supplied articles is imported or executed. The standard
code below is transcribed from active Article 47, article46-appA.tex:270--291.
Use --source-root PATH optionally to check that transcription against the active
TeX table and to check the pinned source dependency hashes.
"""
from collections import Counter, deque
from itertools import product, permutations
from pathlib import Path
import argparse, csv, io, json, re, hashlib

HERE = Path(__file__).resolve().parent
BASES = 'UCAG'
# Order: first base U,C,A,G; second base U,C,A,G; third base U,C,A,G.
CODE_STRING = 'FFLLSSSSYY**CC*W' 'LLLLPPPPHHQQRRRR' 'IIIMTTTTNNKKSSRR' 'VVVVAAAADDEEGGGG'
CODONS = [''.join(c) for c in product(BASES, repeat=3)]
CODE = dict(zip(CODONS, CODE_STRING))
AAS = tuple(sorted(set(CODE.values()) - {'*'}))
D_ORDER = ('00000','00100','01010','01110','10001','10101','11011','11111',
           '00001','00010','00011','00101','00110','00111','01001','01011',
           '01101','01111','11001','11101')
LOCAL = tuple(product((0,1), repeat=2))
MAPS = dict(WC='ACGU', RY='ACUG', MK='GACU')
CERTS = dict(WC=(2,4,6,8,12,13,17,18,20), RY=(2,6,8,12,13,14,17,18,20))
WITNESSES = {
 'WC': {1:'K',2:'T',3:'Y',4:'S',5:'D',6:'R',7:'Q',8:'P',9:'L',10:'I',11:'F',12:'C',14:'W',15:'M',16:'H',17:'A',18:'G',19:'E'},
 'RY': {1:'K',2:'V',3:'R',4:'M',5:'Y',6:'S',7:'H',8:'F',9:'N',10:'E',11:'G',12:'L',13:'T',14:'P',15:'D',16:'Q',17:'I',18:'A',19:'C'},
 'MK': {1:'G',2:'D',3:'V',4:'A',5:'R',6:'T',7:'S',8:'P',9:'F',10:'W',11:'L',12:'H',13:'Y',14:'E',15:'M',16:'C',17:'Q',18:'K',19:'I',20:'N'},
}

def flip(m): return tuple(1-x for x in m)
def reverse(m): return m[::-1]
def orbit(m): return frozenset((m, flip(m), reverse(m), flip(reverse(m))))
def diff(m): return tuple(x^y for x,y in zip(m,m[1:]))
```

```
def bitstr(m): return ''.join(map(str,m))
def marker(d):
    m=[0]
    for x in d: m.append(m[-1]^int(x))
    return tuple(m)
def class_key(m): return min(diff(m), diff(m)[::-1])
def encode(m, assignment='ACGU'):
    L=dict(zip(LOCAL,assignment))
    return ''.join(L[m[i:i+2]] for i in range(0,len(m),2))

def maximum_matching(adj):
    """Augmenting paths found breadth-first; independent of earlier DFS audit."""
    left, right = {}, {}
    for root in sorted(adj):
        if root in left: continue
        q=deque([root]); parent_l={root:None}; parent_r={}; finish=None
        while q and finish is None:
            u=q.popleft()
            for v in sorted(adj[u]):
                if v in parent_r: continue
                parent_r[v]=u
                if v not in right: finish=v; break
                nxt=right[v]
                if nxt not in parent_l: parent_l[nxt]=v; q.append(nxt)
        if finish is not None:
            v=finish
            while v is not None:
                u=parent_r[v]; old=left.get(u)
                left[u]=v; right[v]=u; v=old
    assert len(left)==len(right)
    return left

def partition(assignment):
    return '|'.join(sorted((''.join(sorted((assignment[0],assignment[3]))),
                            ''.join(sorted((assignment[1],assignment[2])))))

def calculate():
    all_m=tuple(product((0,1), repeat=6))
    classes=[orbit(marker(d)) for d in D_ORDER]
    assert len(set(classes))==20 and set().union(*classes)==set(all_m)
    q={m:i for i,O in enumerate(classes,1) for m in O}
    assert len(q)==64
    for m in all_m:
        for n in all_m:
            assert (class_key(m)==class_key(n)) == (q[m]==q[n])
    group_counts={}
    for N in range(1,7):
        universe=tuple(product((0,1),repeat=2*N))
        obs=set(map(orbit,universe)); sizes=Counter(map(len,obs))
        assert len(obs)==2**(2*N-2)+2**(N-1)
        assert sizes[2]==2**N and sizes[4]==2**(2*N-2)-2**(N-1)
        group_counts[str(N)]={'framed':len(universe),'classes':len(obs),'orbit_sizes':dict(sorted(sizes.items()))}

    rho=str.maketrans('ACGU','UGCA'); tau=str.maketrans('ACGU','AGCU')
    for m in all_m:
        assert encode(flip(m))==encode(m).translate(rho)
        assert encode(reverse(m))==encode(m)[::-1].translate(tau)
    rows=[]; graphs={}
    for name,assignment in MAPS.items():
        graph={}; grows=[]
        for i,O in enumerate(classes,1):
            words=sorted(encode(m,assignment) for m in O)
            neigh=sorted({CODE[w] for w in words}-{'*'})
            graph[i]=set(neigh)
            grows.append({'class':i,'codons':words,'neighbors':neigh,
                          'incidences':[{'marker':bitstr(m),'codon':encode(m,assignment),
                                         'product':CODE[encode(m,assignment)]} for m in sorted(O)]})
        mt=maximum_matching(graph); wit=WITNESSES[name]
        assert len(set(wit.values()))==len(wit)
        assert all(a in graph[i] for i,a in wit.items())
        assert len(mt)==len(wit)=={'WC':18,'RY':19,'MK':20}[name]
        cert={}
        if name in CERTS:
            S=CERTS[name]; neigh=sorted(set().union(*(graph[i] for i in S)))
            cert={'classes':list(S),'neighbors':neigh,'deficiency':len(S)-len(neigh)}
```

```
            assert len(mt)==20-cert['deficiency']
        graphs[name]={'letter_assignment':dict(zip(('00','01','10','11'),assignment)),
                      'partition':partition(assignment),'maximum_matching':len(mt),
                      'displayed_matching':wit,'hall_certificate':cert,'rows':grows}
    # Exhaust all 24 bijections, checking family counts and matching invariance.
    all24=[]
    for letters in permutations('ACGU'):
        assignment=''.join(letters)
        graph={i:{CODE[encode(m,assignment)] for m in O}-{'*'} for i,O in enumerate(classes,1)}
        size=len(maximum_matching(graph)); part=partition(assignment)
        assert size=={'AU|CG':18,'AG|CU':19,'AC|GU':20}[part]
        all24.append({'assignment':assignment,'partition':part,'maximum_matching':size})
    assert Counter(r['partition'] for r in all24)=={'AU|CG':8,'AG|CU':8,'AC|GU':8}
    # Direct decorated-carrier orbit enumeration, no inferred division formula.
    def action(state, f, r, keep_dir):
        m,a,*rest=state
        if f: m=flip(m)
        if r: m=reverse(m)
        return (m,a^f,rest[0]^r) if keep_dir else (m,a^f)
    decorated={}
    for keep_dir in (False,True):
        states=[(m,a,d) if keep_dir else (m,a) for m in all_m for a in (0,1) for d in ((0,1) if keep_dir else (0,)
     )]
        obs={frozenset(action(s,f,r,keep_dir) for f,r in product((0,1),repeat=2)) for s in states}
        count=len(obs); assert count==(64 if keep_dir else 36)
        decorated['addressed' if keep_dir else 'reference_only']={'presentations':len(states),'orbits':count}
        if keep_dir:
            def normal(s):
                m,a,d=s
                return reverse(flip(m) if a else m) if d else (flip(m) if a else m)
            assert len(set(map(normal,states)))==64
            for O in obs: assert len({normal(s) for s in O})==1
    # Cardinalities of all residual subgroup quotients.
    residual={'identity':64,'flip':len({frozenset((m,flip(m))) for m in all_m}),
              'reversal':len({frozenset((m,reverse(m))) for m in all_m}),
              'flip_reversal':len({frozenset((m,flip(reverse(m)))) for m in all_m}),'full':20}
    assert residual=={'identity':64,'flip':32,'reversal':36,'flip_reversal':36,'full':20}
    # Counterexample to a supposedly forced central selector.
    def select(m):
        d=diff(m)
        return 2 if d==d[::-1] else (1 if d<d[::-1] else 3)
    assert all(select(flip(m))==select(m) and select(reverse(m))==4-select(m) for m in all_m)
    selector=dict(sorted(Counter(map(select,all_m)).items())); assert selector=={1:24,2:16,3:24}
    feature=Counter((sum(diff(next(iter(O)))),len(O),diff(next(iter(O)))[2]) for O in classes)
    assert len(feature)==12 and Counter(feature.values())=={1:4,2:8}
    # Minimal common refinement by the standard output (including STOP).
```

```
    cells=[]
    for i,O in enumerate(classes,1):
        outputs={CODE[encode(m)] for m in O}
        for aa in sorted(outputs):
            cells.append({'class':i,'output':aa,'codons':sorted(encode(m) for m in O if CODE[encode(m)]==aa)})
    doubles=[c['codons'] for c in cells if len(c['codons'])>1]
    assert len(cells)==61
    assert sorted(doubles)==[['AGC','UCG'],['AGU','UCA'],['CUU','UUG']]
    histogram=Counter(len({CODE[encode(m)] for m in O}) for O in classes)
    assert histogram=={1:1,2:7,3:2,4:10}
    # Validate the sharp two-edge illustration, without assigning physical meaning.
    extension=dict(WITNESSES['WC']); extension.update({13:'N',20:'V'})
    assert set(extension)==set(range(1,21)) and set(extension.values())==set(AAS)
    for i,O in enumerate(classes,1):
        m=min(O); stabilizer='R' if reverse(m)==m else ('FR' if flip(reverse(m))==m else '1')
        rows.append({'class':i,'D_representative':D_ORDER[i-1],'D_canonical':bitstr(class_key(m)),
                     'size':len(O),'stabilizer':stabilizer,'markers':[bitstr(n) for n in sorted(O)]})
    return {'schema_version':1,'conventions':{'F':'global bit complement','R':'six-site reversal',
            'local_pair_order':['00','01','10','11'],'physical_tether':'AAA (inert; never transformed into a
     codon)',
            'code_source':'Article 47 active article46-appA.tex, Table 6, lines 270--291'},
            'standard_code':CODE,'amino_acids':AAS,'classes':rows,'graphs':graphs,'all24':all24,
            'counts':group_counts,'decorated_carriers':decorated,'residual_quotients':residual,
            'minimal_refinement':{'cell_count':len(cells),'cells':cells,'within_orbit_output_histogram':dict(
     histogram)},
            'selector_histogram':selector,'feature_cells':{'count':len(feature),'cell_size_histogram':dict(Counter
     (feature.values()))}}
```

```
def csv_data(data):
    out=io.StringIO(newline=''); writer=csv.writer(out,lineterminator='\n')
    writer.writerow(['class','D_representative','D_canonical','size','stabilizer','markers',
                     'WC_codons','WC_neighbors','RY_codons','RY_neighbors','MK_codons','MK_neighbors'])
    for r in data['classes']:
        row=[r[k] for k in ('class','D_representative','D_canonical','size','stabilizer')]+[' '.join(r['markers'])]
        for name in MAPS:
            g=data['graphs'][name]['rows'][r['class']-1]
            row += [' '.join(g['codons']),' '.join(g['neighbors'])]
        writer.writerow(row)
    out2=io.StringIO(newline=''); writer2=csv.writer(out2,lineterminator='\n')
    writer2.writerow(['marker','class','F_marker','R_marker','FR_marker','WC_codon','WC_product','RY_codon','RY_product','MK_codon','MK_product'])
    q={m:r['class'] for r in data['classes'] for m in r['markers']}
    for m in product((0,1),repeat=6):
        row=[bitstr(m),q[bitstr(m)],bitstr(flip(m)),bitstr(reverse(m)),bitstr(flip(reverse(m)))]
        for v in MAPS.values(): w=encode(m,v); row += [w,CODE[w]]
        writer2.writerow(row)
    return {'orbit_incidence.csv':out.getvalue(),'marker_incidence.csv':out2.getvalue()}

def check_sources(root):
    source=root/'FortySevenArticleCodeTable/article46-appA.tex'
    txt=source.read_text(); table=txt.split('%%  TABLE 6 -- AFTER TAU_2 = STANDARD CODE')[1].split('%%  TABLE 7')[0]
    parsed={c:(plain or moved or '*') for c,plain,moved,stop in re.findall(r'&([ACGU]{3})&(?:([ACDEFGHIKLMNPQRSTVWY])|\\mv\{([ACDEFGHIKLMNPQRSTVWY])\}\{[^}]*\}|(\\Stopcd))',table)}
    assert parsed==CODE,'Embedded code differs from active source table'
    inventory=HERE/'registry_source_inventory.json'
    if inventory.exists():
        data=json.loads(inventory.read_text())
        for entry in data['active_source_hashes']:
            assert hashlib.sha256((root/entry['path']).read_bytes()).hexdigest()==entry['sha256'],entry['path']

def main():
    parser=argparse.ArgumentParser();parser.add_argument('--write',action='store_true');parser.add_argument('--source-root',type=Path)
    args=parser.parse_args(); data=calculate()
    content={'registry_data.json':json.dumps(data,indent=2,sort_keys=True)+'\n', **csv_data(data)}
    for name,text in content.items():
        p=HERE/name
        if args.write: p.write_text(text)
        else: assert p.read_text()==text, f'{name}: data mismatch; do not overwrite before investigation'
    if args.source_root: check_sources(args.source_root)
    print(json.dumps({'status':'PASS','registry_classes':20,'orbit_profile':{'2':8,'4':12},
                      'matching_sizes':{k:v['maximum_matching'] for k,v in data['graphs'].items()},
                      'decorated_quotients':data['decorated_carriers'],'all24_checked':24,
                      'source_check':bool(args.source_root),'tables_verified':list(content)},indent=2))
if __name__=='__main__': main()
'''
# END UNMODIFIED SOURCE

# BEGIN UNMODIFIED SOURCE: certificates/products/build_product_certificates.py
SOURCES['certificates/products/build_product_certificates.py'] = r'''#!/usr/bin/env python3
"""New, explicitly relaxed graph witnesses. No external packages or source writes.
These certificates establish typed graph reachability, not chemical feasibility.
"""
import json, collections, itertools, hashlib
from pathlib import Path
HERE=Path(__file__).resolve().parent
SOURCE=HERE/'source_retention_rows.json'
rows=json.loads(SOURCE.read_text())['construction_rows']
source_rows={r['target']:r for r in rows}
PER=['6','6a','7','8','9','10','10a','10b','11','12','12a','1','2','3','4','4a','4b','5']
E={tuple(sorted((PER[i],PER[(i+1)%18]))) for i in range(18)}
E|={tuple(sorted(e)) for e in [('4a','12a'),('4b','10b'),('6a','10a')]}
CAP={'C':4,'N':3,'O':2,'S':2,'H':1,'*':1}
def edge(u,v):return tuple(sorted((u,v)))
def chain(xs,orders=None):
 return {edge(a,b): (orders[i] if orders else 1) for i,(a,b) in enumerate(zip(xs,xs[1:]))}
def merge(*parts):
 d={}
 for p in parts:
  for e,b in p.items():
```

```
   assert e not in d or d[e]==b,(e,d.get(e),b)
   d[e]=b
 return d
# Complete sidechain heavy-atom graphs. Carbon names are inherited locants.
side={
 'Gly':{},'Ala':{},'Ser':{edge('5','On1'):1},'Cys':{edge('5','S1'):1},
 'Asp':merge(chain(['5','4b']),{edge('4b','On1'):2,edge('4b','On2'):1}),
 'Asn':merge(chain(['5','4b']),{edge('4b','On1'):2,edge('4b','Nn1'):1}),
 'Thr':merge(chain(['5','4b']),{edge('5','On1'):1}),
 'Glu':merge(chain(['5','4b','4a']),{edge('4a','On1'):2,edge('4a','On2'):1}),
 'Gln':merge(chain(['5','4b','10b']),{edge('10b','On1'):2,edge('10b','Nn1'):1}),
 'Pro':chain(['5','4b','10b','Na']),
 'Met':chain(['5','4b','S1','10b']),
 'Lys':chain(['5','4b','10b','11','Nn1']),
 'Arg':merge(chain(['5','4b','10b','Nn1','11']),{edge('11','Nn2'):2,edge('11','Nn3'):1}),
 'His':merge(chain(['5','4b']),chain(['4b','10b','Nn1','11','Nn2','4b'],[2,1,1,2,1])),
 'Leu':merge(chain(['5','4b','4a']),{edge('4b','10b'):1}),
 'Val':merge(chain(['5','6']),{edge('5','4b'):1}),
 'Ile':merge(chain(['5','4b','10b']),{edge('5','6'):1}),
 'Phe':merge(chain(['5','4b']),chain(['4b','10b','11','12','12a','4a','4b'],[2,1,2,1,2,1])),
 'Tyr':merge(chain(['5','4b']),chain(['4b','10b','11','12','12a','4a','4b'],[2,1,2,1,2,1]),{edge('12','O12'):1}),
 'Trp':merge(chain(['5','4b']),chain(['4b','10b','Nn1','12a','4a','4b'],[2,1,1,2,1]),chain(['12a','1','2','3','4',
     '4a'],[1,2,1,2,1])),
}
expected={'Ala':'C3H7NO2','Arg':'C6H14N4O2','Asn':'C4H8N2O3','Asp':'C4H7NO4','Cys':'C3H7NO2S','Gln':'C5H10N2O3','
     Glu':'C5H9NO4','Gly':'C2H5NO2','His':'C6H9N3O2','Ile':'C6H13NO2','Leu':'C6H13NO2','Lys':'C6H14N2O2','Met':'
     C5H11NO2S','Phe':'C9H11NO2','Pro':'C5H9NO2','Ser':'C3H7NO3','Thr':'C4H9NO3','Trp':'C11H12N2O2','Tyr':'
     C9H11NO3','Val':'C5H11NO2'}
class Graph:
 def __init__(self):self.atoms={};self.bonds={}
 def atom(self,a,e):assert a not in self.atoms;self.atoms[a]=e
 def bond(self,u,v,b=1):assert u in self.atoms and v in self.atoms;self.bonds[edge(u,v)]=b
 def deg(self,a):return sum(b for e,b in self.bonds.items() if a in e)
 def valid(self,closed=False):
  assert all(u in self.atoms and v in self.atoms and u!=v and 0<b<=3 for (u,v),b in self.bonds.items())
  for a,e in self.atoms.items():assert self.deg(a)<=CAP[e],(a,e,self.deg(a))
  if closed:
   assert all(self.deg(a)==CAP[e] for a,e in self.atoms.items())
 def saturate(self,prefix):
  for a,e in list(self.atoms.items()):
   if e=='H':continue
   for j in range(CAP[e]-self.deg(a)):
    h=f'{prefix}:{a}:{j}';self.atom(h,'H');self.bond(a,h)
 def copy(self):
  x=Graph();x.atoms=dict(self.atoms);x.bonds=dict(self.bonds);return x
 def components(self):
  unseen=set(self.atoms);ans=[]
  while unseen:
   root=min(unseen);seen={root};q=[root]
   while q:
    x=q.pop()
    for e in self.bonds:
     if x in e:
      y=e[0] if e[1]==x else e[1]
      if y not in seen:seen.add(y);q.append(y)
   ans.append(seen);unseen-=seen
  return ans
 def formula(self,vertices):
  c=collections.Counter(self.atoms[a] for a in vertices if self.atoms[a]!='*')
  return ''.join(e+(str(c[e]) if c[e]>1 else '') for e in ['C','H','N','O','S'] if c[e])
 def serial(self):
  return {'atoms':self.atoms,'bonds':[[u,v,b] for (u,v),b in sorted(self.bonds.items())]}
def initial(sulfur=False):
 g=Graph()
 for c in PER:g.atom(c,'C')
 for e in E:g.bond(*e)
 # Explicit alternating matching in the perimeter, one pi unit per carbon.
 for i in range(0,18,2):g.bond(PER[i],PER[i+1],2)
 for a,e in [('Ca','C'),('Cc','C'),('Na','N'),('Ox','O'),('Ot','O'),('O6','O'),('O12','O'),('Bt','*'),('B12','*')
     ]:g.atom(a,e)
 for u,v,b in [('5','Ca',1),('Ca','Cc',1),('Ca','Na',1),('Cc','Ox',2),('Cc','Ot',1),('Ot','Bt',1),('6','O6',1),('
     12','O12',1),('O12','B12',1)]:g.bond(u,v,b)
 # Identical finite non-sulfur packet for every module: 3 NH3, 3 H2O, 8 H2.
 # On0 is the free-acid oxygen; On1/On2 are available side-chain oxygens.
 for i in range(1,4):g.atom(f'Nn{i}','N')
```

```
 for i in range(3):g.atom(f'On{i}','O')
 if sulfur:g.atom('S1','S');g.atom('S2','S');g.bond('S1','S2') # H-S-S-H after saturation.
 g.saturate('hi')
 for i in range(8):
  a=f'h2:{i}:a';b=f'h2:{i}:b';g.atom(a,'H');g.atom(b,'H');g.bond(a,b)
 g.valid(True);return g
```

```
def endpoint(name,g,paired=False):
 r=source_rows[name];S=set(r['retained']);out=g.copy()
 # Temporarily remove all H bonds; atoms are retained and reassigned below.
 out.bonds={e:b for e,b in out.bonds.items() if all(out.atoms[x]!='H' for x in e)}
 removed=set(r['removed_entire']);cuts={edge(*e) for e in r['severed']}
 if name=='Cys':removed=set();cuts={edge('4b','5'),edge('5','6')}
 for e in list(out.bonds):
  if e in cuts or any(x in removed for x in e):del out.bonds[e]
 # Remove all S-retained incidences other than C-C sidechain edges / backbone.
 for e in list(out.bonds):
  if any(x in S for x in e):
   if set(e)<=(S|{'Ca'}):continue
   if name=='Tyr' and e==edge('12','O12'):continue
   del out.bonds[e]
 # Replace only retained-heavy subgraph by the specified sidechain. Remaining C-C edges stay.
 for e in list(out.bonds):
  if set(e)<=S:del out.bonds[e]
 out.bonds.update(side[name])
 if name=='Gly':out.bonds.pop(edge('5','Ca'))
 # Hydrolysis projection: carbonyl Ox persists, OnO replaces ester Ot; Ot-Bt is unchanged.
 out.bonds.pop(edge('Cc','Ot'));out.bonds[edge('Cc','OnO')]=1
 if name=='Tyr':out.bonds.pop(edge('O12','B12'))
 if name in ['Cys','Met']:out.bonds.pop(edge('S1','S2'))
 # Retain original H bonds whenever still possible. Then allocate free mapped H.
 h_used=set()
 for e,b in sorted(g.bonds.items()):
  hs=[a for a in e if g.atoms[a]=='H'];heavy=[a for a in e if g.atoms[a]!='H']
  if len(hs)==1 and out.deg(heavy[0])<CAP[g.atoms[heavy[0]]]:
   out.bonds[e]=1;h_used.add(hs[0])
 free=sorted(a for a,e in g.atoms.items() if e=='H' and a not in h_used)
 nneed=sum(CAP[e]-out.deg(a) for a,e in out.atoms.items() if e!='H')
 assert nneed<=len(free),(name,nneed,len(free))
 for a,e in sorted(out.atoms.items()):
  if e=='H':continue
  while out.deg(a)<CAP[e]:out.bonds[edge(a,free.pop(0))]=1
 assert len(free)%2==0,(name,'odd residual hydrogens',len(free))
 for a,b in zip(free[::2],free[1::2]):out.bonds[edge(a,b)]=1
 out.valid(True)
 pcomp=next(c for c in out.components() if 'Ca' in c)
 assert set(c for c in pcomp if out.atoms[c]=='C')==S|{'Ca','Cc'}
 assert out.formula(pcomp)==expected[name],(name,out.formula(pcomp),expected[name])
 # Target product excludes all boundary ports, and contains no unassigned atom.
 assert all(out.atoms[a]!='*' for a in pcomp)
 return out,pcomp,cuts,removed

def derive(g,t):
 cur=g.copy();steps=[]
 # Common-subgraph normal form. Each split/join changes one bond-order unit.
 for e,b in sorted(g.bonds.items()):
  for j in range(b-t.bonds.get(e,0)):
   cur.bonds[e]-=1
   if not cur.bonds[e]:del cur.bonds[e]
   cur.valid();steps.append(['split',*e])
 for e,b in sorted(t.bonds.items()):
  for j in range(b-g.bonds.get(e,0)):
   cur.bonds[e]=cur.bonds.get(e,0)+1;cur.valid();steps.append(['join',*e])
 assert cur.atoms==g.atoms==t.atoms and cur.bonds==t.bonds
 return steps

def rooted_signature(g,vs):
 # A complete rooted graph string by exhaustive permutations within refinement cells.
 # WL alone separates the 20 rooted graphs here; retain vectors for audit, not claim a universal canonizer.
 labels={a:(g.atoms[a],a if a in ['Ca','Cc','Na'] else '',g.deg(a)) for a in vs}
 palette={x:i for i,x in enumerate(sorted(set(labels.values())))}
 col={a:palette[x] for a,x in labels.items()}
 for _ in range(len(vs)):
```

```
  sig={a:(col[a],tuple(sorted((b,col[v if u==a else u]) for (u,v),b in g.bonds.items() if a in (u,v) and u in vs
     and v in vs))) for a in vs}
  pal={x:i for i,x in enumerate(sorted(set(sig.values())))}
  nxt={a:pal[x] for a,x in sig.items()}
  if nxt==col:break
  col=nxt
 return sorted((labels[a],col[a]) for a in vs)

out={};summaries=[]
for name in sorted(side):
 g=initial(name in ['Cys','Met']);t,vs,cuts,removed=endpoint(name,g);steps=derive(g,t)
 # Atom count excludes two non-atom interface ports.
 origins={a:('framework' if a in PER else 'arm_derived_backbone' if a in ['Ca','Cc'] else 'boundary_port' if g.
     atoms[a]=='*' else 'prepared_scaffold' if a in ['Na','Ox','Ot','O6','O12'] else 'declared_resource') for a in
      g.atoms}
 for a in g.atoms:
  if a.startswith('hi:'):
   parent=a.split(':')[1]
   origins[a]='hydrogen_on_initial_'+origins[parent]

 stereo={} if name=='Gly' else {'Ca':'R' if name=='Cys' else 'S'}
 if name=='Ile':stereo['5']='S'
 if name=='Thr':stereo['5']='R'
 # Check source-guided restrictions separately from valence-port reachability.
 S=set(r for r in source_rows[name]['retained'])
 gone=set() if name=='Cys' else set(source_rows[name]['removed_entire'])
 active=S|gone
 changed={e for e in set(g.bonds)|set(t.bonds) if g.bonds.get(e,0)!=t.bonds.get(e,0)}
 protected={edge('Ca','Cc'),edge('Ca','Na'),edge('Cc','Ox')}
 assert not (changed&protected)
 for e in changed:
  if all(g.atoms[a]=='C' and a in PER for a in e):
   assert e in E and any(a in active for a in e),(name,'out-of-support C-C edit',e)
   assert t.bonds.get(e,0)<=2,(name,e)
 for e in t.bonds:
  if all(g.atoms[a]=='C' and a in PER for a in e):assert e in E
 assert len(S)<=9 and len(gone)<=2
 internal_lost=[e for e in E if set(e)<=S and e not in t.bonds]
 assert len(internal_lost)<=1
 assert all(t.bonds[e]>=1 for e in E if set(e)<=S and e not in internal_lost)
 # No non-framework carbon can enter a side chain; only the two fixed backbone carbons are permitted.
 assert set(a for a in vs if g.atoms[a]=='C')==S|{'Ca','Cc'}
 row={'restricted_grammar_checks_passed':True,'name':name,'certificate_id':'chem:'+name,'source57':source_rows[
     name],'selected_variant':'59 boundary-cut' if name=='Cys' else '57 retained-set/cut convention','initial':g.
     serial(),'endpoint':t.serial(),'persistent_origin':origins,'product_vertices':sorted(vs),'product_formula':t.
     formula(vs),'product_heavy_bonds':[[u,v,b] for (u,v),b in sorted(t.bonds.items()) if u in vs and v in vs and
     g.atoms[u]!='H' and g.atoms[v]!='H'],'source_carbon_retained':sorted(set(PER)&vs),'stereo_required':stereo,'
     stereo_proved_by_graph_rules':False,'rewrite_steps':steps,'step_count':len(steps),'
     atom_count_excluding_ports':sum(e!='*' for e in g.atoms.values()),'element_counts':dict(collections.Counter(e
      for e in g.atoms.values() if e!='*')),'unmodified_remnant_cc_edges':[[*e,b] for e,b in sorted(g.bonds.items
     ()) if all(a in set(PER)-set(source_rows[name]['retained'])-removed for a in e) and e not in cuts],'
     endpoint_component_formulas':[t.formula(v) for v in t.components()],'all_valences_closed':True,'
     all_atoms_conserved':True,'total_charge_initial':0,'total_charge_endpoint':0,'formal_only':True}
 # Source-boundary isolation and remnant bonds conserved exactly.
 for u,v,b in row['unmodified_remnant_cc_edges']:assert t.bonds[edge(u,v)]==b
 out[name]=row;summaries.append({k:row[k] for k in ['name','product_formula','step_count','
     atom_count_excluding_ports','stereo_required']})
# Prove structural distinctness: all formulas distinct except Leu/Ile, distinguished by root branch structure.
formulas=collections.defaultdict(list)
for name,row in out.items():formulas[row['product_formula']].append(name)
assert [x for x in formulas.values() if len(x)>1]==[['Ile','Leu']]
# At retained root C5, Ile has 3 carbon neighbors including Ca, Leu has 2.
for name,n in [('Ile',3),('Leu',2)]:
 b=out[name]['endpoint']['bonds'];ats=out[name]['endpoint']['atoms'];assert sum('5' in (u,v) and ats[v if u=='5'
     else u]=='C' for u,v,_ in b)==n
# Simultaneous sulfur cell: construct disjoint module union and use S3--S4 supply, not duplicated HSSH.
def rename(g,p):
 h=Graph();h.atoms={p+a:e for a,e in g.atoms.items()};h.bonds={edge(p+u,p+v):b for (u,v),b in g.bonds.items()};
     return h
c=initial(False);m=initial(False)
for g in [c,m]:g.atom('S1','S')
C=rename(c,'X3:');M=rename(m,'X4:');J=Graph();J.atoms=C.atoms|M.atoms;J.bonds=C.bonds|M.bonds;J.bond('X3:S1','X4:
     S1')
for p in ['X3:','X4:']:
 h=p+'hS';J.atom(h,'H');J.bond(p+'S1',h)
```

```
J.valid(True)
# Reuse target graph construction with temporary spare sulfur; discard spare and its initial hydrogen identically from both endpoints.
# The global normal-form theorem avoids claiming a coupled elementary mechanism.
endparts=[]
for name,p in [('Cys','X3:'),('Met','X4:')]:
 a=initial(True);b,_,_,_=endpoint(name,a)
 # The product atom map plus untouched initial graph is all that is needed for simultaneous allocation.
 # Rebuild heavy endpoint using the one-S initial module; saturation is solved globally below.
 hh=rename(b,p)
 keep=set((C if p=='X3:' else M).atoms)|{p+'hS'}
 endparts.append({edge(u,v):bo for (u,v),bo in hh.bonds.items() if u in keep and v in keep and hh.atoms[u]!='H' and hh.atoms[v]!='H'})
JT=J.copy();JT.bonds=endparts[0]|endparts[1]
free=sorted(a for a,e in JT.atoms.items() if e=='H')
for a,e in sorted(JT.atoms.items()):
 if e=='H':continue
 while JT.deg(a)<CAP[e]:JT.bonds[edge(a,free.pop(0))]=1
assert len(free)%2==0
for a,b in zip(free[::2],free[1::2]):JT.bonds[edge(a,b)]=1
JT.valid(True)
# A genuinely bridge-ordered joint prefix, not merely disconnected endpoint assembly.
Jcur=J.copy();prefix=[];stages={}
def jedit(kind,u,v):
 e=edge(u,v)
 if kind=='split':
  assert Jcur.bonds.get(e,0)>0
  Jcur.bonds[e]-=1
  if not Jcur.bonds[e]:del Jcur.bonds[e]
 else:Jcur.bonds[e]=Jcur.bonds.get(e,0)+1
 Jcur.valid();prefix.append([kind,u,v])
jedit('split','X4:4b','X4:10b')
for u,v in [('X3:5','X3:6'),('X3:5','X3:4b')]:
 while Jcur.bonds.get(edge(u,v),0):jedit('split',u,v)
for p in ['X3:','X4:']:jedit('split',p+'S1',p+'hS')
jedit('join','X3:5','X3:S1')
jedit('join','X4:S1','X4:4b')
stages['disulfide_bridge']={'after_step':len(prefix),'graph':Jcur.serial(),'path':['X3:5','X3:S1','X4:S1','X4:4b']}
assert all(Jcur.bonds.get(edge(a,b))==1 for a,b in zip(stages['disulfide_bridge']['path'],stages['disulfide_bridge']['path'][1:]))
jedit('split','X3:S1','X4:S1')
jedit('join','X4:S1','X4:10b')
stages['sulfur_topologies']={'after_step':len(prefix),'graph':Jcur.serial()}
assert Jcur.bonds.get(edge('X3:S1','X4:S1'),0)==0
assert Jcur.bonds[edge('X4:4b','X4:S1')]==Jcur.bonds[edge('X4:S1','X4:10b')]==1
jsteps=prefix+derive(Jcur,JT)

comps=JT.components()
assert JT.formula(next(c for c in comps if 'X3:Ca' in c))==expected['Cys']
assert JT.formula(next(c for c in comps if 'X4:Ca' in c))==expected['Met']
junction={'bridge_ordered':True,'stages':stages,'initial':J.serial(),'endpoint':JT.serial(),'rewrite_steps':jsteps,'step_count':len(jsteps),'sulfur_initial_bond':['X3:S1','X4:S1'],'sulfur_atom_count':2,'all_atoms_conserved':True,'all_valences_closed':True,'formal_only':True}
(HERE/'product_certificates.json').write_text(json.dumps({'scope':'New relaxed valence-port graph grammar only; no mechanism, geometry, kinetics, chemical feasibility, or stereoselectivity certification. Source-guided 20 complete constitutional graph witnesses.','common_packet':'Per module 3 NH3 + 3 H2O + 8 H2, plus one shared HSSH for paired Cys/Met; resource excess remains represented.','rows':out,'paired_sulfur':junction},indent=2)+'\n')
(HERE/'product_verification_summary.json').write_text(json.dumps({'source_sha256':hashlib.sha256(SOURCE.read_bytes()).hexdigest(),'checks_passed':True,'products':summaries,'source_count':len(rows),'ordinary_non_sulfur':16,'glycine_backbone_only':1,'tryptophan_extended':1,'sulfur_paired':2,'constitutional_graphs_distinct':True,'max_single_row_steps':max(r['step_count'] for r in out.values()),'sulfur_pair_steps':len(jsteps),'sulfur_bridge_order_verified':True,'resource_h2_per_module':8,'batch_active_scaffolds':20,'batch_original_cores_if_each_ns_uses_one_hexamer_and_sulfur_uses_two':120,'stereo_selection_certified':False,'molecular_feasibility_certified':False,'restricted_grammar_checks_passed':True,'ablations':{'no_extended_constructor_max_product_carbons':10,'tryptophan_product_carbons':11,'no_sulfur_excludes':['Cys','Met'],'no_sidechain_nitrogen_excludes':['Arg','Asn','Gln','His','Lys','Trp'],'new_core_carbon_adjacency_forbidden':True,'protected_backbone_bonds':['Ca-Cc','Ca-Na','Cc=Ox']}},indent=2)+'\n')
print(json.dumps({'products':summaries,'paired_sulfur_steps':len(jsteps),'all_checks_passed':True},indent=2))
'''
# END UNMODIFIED SOURCE
```

```
# BEGIN UNMODIFIED SOURCE: certificates/reachability/verify_reachability.py
SOURCES['certificates/reachability/verify_reachability.py'] = r'''#!/usr/bin/env python3
"""Independent, stdlib-only certificate for the assignment/reachability chapter.

No source scripts are imported or executed. No source files are changed.
Run: python3 verify_reachability.py [output.json]
The conventional table strings below are disclosed reference data; they are
used only by assertions comparing outputs, never by move guards or scores.
"""
from collections import Counter, deque
from copy import deepcopy
from fractions import Fraction as F
from itertools import product, combinations
from math import factorial
from pathlib import Path
import json
import sys

B = 'UCAG'
WORDS = [''.join(x) for x in product(B, repeat=3)]
PALETTE = set('ACDEFGHIKLMNPQRSTVWY')
REFERENCE_STANDARD = ('FFLLSSSSYY**CC*W' 'LLLLPPPPHHQQRRRR'
                      'IIIMTTTTNNKKSSRR' 'VVVVAAAADDEEGGGG')
REFERENCE_MITO = ('FFLLSSSSYY**CCWW' 'LLLLPPPPHHQQRRRR'
                  'IIMMTTTTNNKKSS**' 'VVVVAAAADDEEGGGG')
STD = dict(zip(WORDS, REFERENCE_STANDARD))
MITO = dict(zip(WORDS, REFERENCE_MITO))

# Twenty frame classes, reconstructed without a source table.
def flip(x): return ''.join('B' if c == 'F' else 'F' for c in x)
def orbit(x): return frozenset((x, flip(x), x[::-1], flip(x[::-1])))
def dcan(x):
    d = ''.join(str(int(a != b)) for a,b in zip(x,x[1:]))
    return min(d,d[::-1])
orbits = set(orbit(''.join(x)) for x in product('FB',repeat=6))
ordered = sorted(orbits, key=lambda q:(len(q),dcan(sorted(q)[0])))
marker = {'FF':'A','FB':'C','BF':'G','BB':'U'}
classes = {i+1:{''.join(marker[x[k:k+2]] for k in (0,2,4)) for x in q}
           for i,q in enumerate(ordered)}
assert len(classes)==20 and Counter(map(len,classes.values()))=={2:8,4:12}
assert set.union(*classes.values()) == set(WORDS)
assert all(sum(w.startswith(b) for w in q)<=1
           for q in classes.values() for b in map(''.join, product(B,repeat=2)))

# Slots are cells of a fixed partition; Y/R/N describe the third base.
def codons(slot): return {slot[:2]+x for x in {'Y':'UC','R':'AG','N':B}[slot[2]]}
QUARTETS = ['UCN','CUN','CCN','CGN','ACN','GUN','GCN','GGN']
HALVES = [b+h for b in ('UU','UA','UG','CA','AU','AA','AG','GA') for h in 'YR']
CELLS = QUARTETS+HALVES
assert len(CELLS)==24 and sum(map(lambda s:len(codons(s)),CELLS))==64
assert set.union(*(codons(s) for s in CELLS))==set(WORDS)
assert all(not(codons(s)&codons(t)) for s,t in combinations(CELLS,2))

# The source50 geometrical role graph with its reference-informed W/M anchors.
# Selected labels C at UAY and Y at UGY are source57 initial data.
ROWS = [
 ('F',['UUY'],1,[1,11]), ('S',['UCN','AGY'],4,[4,17]),
 ('C',['UAY'],3,[3,15]), ('Y',['UGY'],13,[12,13]),
 ('W',['UGR'],14,[14]), ('L',['CUN'],9,[5,9,15,19]),
 ('P',['CCN'],18,[8,14,18,20]), ('H',['CAY'],19,[16,19]),
 ('R',['CGN'],12,[6,12,17,20]), ('I',['AUY'],16,[10,16]),
 ('M',['AUR'],15,[15]), ('T',['ACN'],2,[2,12,13,14]),
 ('N',['AAY'],10,[9,10]), ('V',['GUN'],7,[7,11,16,19]),
 ('A',['GCN'],6,[6,12,17,20]), ('D',['GAY'],5,[5,15]),
 ('G',['GGN'],8,[8,14,18,20])]
ANCHORS = {'UGR':{'UGG'},'AUR':{'AUG'}}
def eligible(q,slots):
    return all(classes[q] & ANCHORS.get(s,codons(s)) for s in slots)
for r,slots,q,expected in ROWS:
    assert [j for j in classes if eligible(j,slots)]==expected
    assert q in expected
assert len(ROWS)==17 and len(set(q for _,_,q,_ in ROWS))==17
seed_cells = [s for _,ss,_,_ in ROWS for s in ss]
assert len(seed_cells)==18 and len(set(seed_cells))==18
```

```
spare_hist=Counter()
mandatory=set(classes)
def matchings(k,used):
    global mandatory
    if k==len(ROWS):
        spare_hist[tuple(sorted(set(classes)-used))]+=1
        mandatory &= used
        return 1
    return sum(matchings(k+1,used|{q}) for q in ROWS[k][3] if q not in used)
matching_count=matchings(0,set())
assert matching_count==840 and len(spare_hist)==87
assert mandatory=={3,5,9,10,14,15,16,19}

# Common carrier/decoder are immutable typed data. The only edited fields below
# are routing/reading-control records, not atom identities or physical AAA.
CARRIER = {'original_cores':6,'residual_cores':3,'resident_vessels':3,
           'fixed_reference':'AAA','frame':[0,0]}
DECODER = {r:r for r in sorted(PALETTE)}  # Interface to identity-checked product graphs.
REFINEMENTS = {'UUR':'L','CAR':'K','AAR':'Q','GAR':'E'}
DORMANT = {'UAR','AGR'}
LOCKED = {'AUR'}
state={'phase':'seed','a':{},'used':set(),'seeded':set(),'ledger':[],
       'carrier':CARRIER.copy(),'decoder':DECODER.copy(),'u':F(0),'equipped':False}
trace=[]
def record(move):
    trace.append({'move':move,'phase':state['phase'],
                  'u':str(state['u']),'assigned_codons':len(state['a']),
                  'table':{w:state['a'].get(w) for w in WORDS},
                  'classes_used':sorted(state['used']),
                  'decoded_sets':({w:sorted(v) for w,v in readout(state['u']).items()}
                                  if state['equipped'] else None)})
    assert state['carrier']==CARRIER and state['decoder']==DECODER
    assert set(state['a'].values()) <= PALETTE|{'*'}
def seed(r,slots,q):
    assert state['phase']=='seed' and q not in state['used']
    assert any(r==rr and slots==ss for rr,ss,_,_ in ROWS)
    assert eligible(q,slots) and not(set(slots)&state['seeded'])
    anchors={s:next(iter(classes[q]&ANCHORS.get(s,codons(s)))) for s in slots}
    assert all(w not in state['a'] for w in anchors.values())
    state['used'].add(q); state['seeded'].update(slots)
    for s,w in anchors.items():
        state['a'][w]=r; state['ledger'].append((s,w,r,q))
    record('seed '+r+' by class '+str(q))
def propagate(s,w,r,q):
    assert state['phase']=='seed' and (s,w,r,q) in state['ledger']
    assert w in codons(s) and state['a'][w]==r
    assert all(v not in state['a'] or state['a'][v]==r for v in codons(s))
    for v in codons(s): state['a'][v]=r
    record('propagate '+s)
for r,slots,q,_ in ROWS: seed(r,slots,q)
assert len(state['a'])==18
for item in list(state['ledger']): propagate(*item)
assert len(state['a'])==52 and set(state['seeded'])==set(seed_cells)
# Unseeded cells have not been filled by block propagation.
assert all(w not in state['a'] for s in set(CELLS)-set(seed_cells) for w in codons(s))
def advance_phase(target):
    if target=='refine':
        assert state['phase']=='seed' and len(state['used'])==17
        assert state['seeded']==set(seed_cells)
        assert all(w in state['a'] for s in seed_cells for w in codons(s))
    elif target=='production':
        assert state['phase']=='refine' and set(state['a'])==set(WORDS)
        assert all(state['a'][w]==r for s,r in REFINEMENTS.items() for w in codons(s))
        assert all(state['a'][w]=='*' for s in DORMANT for w in codons(s))
    else: raise AssertionError('unpermitted phase advance')
    state['phase']=target
    record('advance phase to '+target)
advance_phase('refine')
for s,r in REFINEMENTS.items():
    assert s not in state['seeded'] and r in PALETTE
    assert all(w not in state['a'] for w in codons(s))
    state['a'].update({w:r for w in codons(s)})
    record('refine '+s+' to '+r)
for s in sorted(DORMANT):
    assert all(w not in state['a'] for w in codons(s))
```

```
    state['a'].update({w:'*' for w in codons(s)})
    record('mark initial no-release '+s)
assert len(state['a'])==64 and len(set(state['a'].values())-{'*'})==20
SELECTED=dict(state['a'])
assert all(SELECTED[w]=='C' for w in codons('UAY'))
assert all(SELECTED[w]=='M' for w in codons('AUR'))
comp=str.maketrans('ACGU','UGCA')
assert {w.translate(comp) for w in codons('UAY')}==codons('AUR')
complementary=[(s,t) for s,t in combinations(seed_cells,2)
               if {w.translate(comp) for w in codons(s)}==codons(t)]
assert len(complementary)==3
assert [(s,t) for s,t in complementary if len(codons(s))==len(codons(t))==2]==[('UAY','AUR')]
advance_phase('production')
```

```
# Modules are maps port -> [(product,a,b)] with margin a+b*u.
def const_module(s,r): return {w[2]:[(r,F(1),F(0))] for w in codons(s)}
mods={s:const_module(s,SELECTED[next(iter(codons(s)))]) for s in CELLS
      if s not in DORMANT}
mods['UAR']={x:[] for x in 'AG'}
mods['AUR']={'A':[('I',F(1),F(3)),('M',F(2),F(1))],
             'G':[('M',F(2),F(1))]}
mods['UGR']={'A':[('W',F(1),F(-2))],'G':[('W',F(1),F(0))]}
mods['AGR']={x:[('R',F(-1),F(2))] for x in 'AG'}
def selected_module(m,x,u):
    scores={r:a+b*u for r,a,b in m[x]}
    positive={r:v for r,v in scores.items() if v>0}
    return set() if not positive else {r for r,v in positive.items() if v==max(positive.values())}
def readout(u):
    return {w:selected_module(mods[s],w[2],u) for s in CELLS for w in codons(s)}
def singular(table):
    return {w:('*' if not rs else next(iter(rs))) for w,rs in table.items() if len(rs)<=1}
def equip():
    assert state['phase']=='production' and not state['equipped']
    assert set(mods)==set(CELLS) and singular(readout(F(0)))==state['a']
    state['equipped']=True
    record('equip declared branch modules')
def close_production():
    assert state['phase']=='production' and state['equipped']
    assert all(state['a'][w]=='C' for w in codons('UAY'))
    assert all(state['a'][w]=='M' for w in codons('AUR'))
    assert {w.translate(comp) for w in codons('UAY')}==codons('AUR')
    state['production_certificate']={'Cys_cell':'UAY','Met_cell':'AUR','closed':True}
    state['phase']='routing'
    record('close paired-production stage')
equip()
close_production()
sense_cells=set(CELLS)-DORMANT

def exchange(s,t):
    assert state['phase']=='routing' and state['u']==0
    assert s in sense_cells and t in sense_cells and s not in LOCKED and t not in LOCKED
    assert s[2]==t[2] and sum(x!=y for x,y in zip(s[:2],t[:2]))==1
    assert all(len(selected_module(mods[z],x,F(0)))==1
               for z in (s,t) for x in mods[z])
    mods[s],mods[t]=mods[t],mods[s]
    state['a']=singular(readout(F(0)))
    record('exchange '+s+' '+t)
    assert state['a']['AUG']=='M'
exchange('UAY','UGY')
AFTER_ONE=dict(state['a'])
exchange('AAR','CAR')
assert state['a']==MITO
assert sum(SELECTED[w]!=MITO[w] for w in WORDS)==8
assert sum(SELECTED[next(iter(codons(s)))]!=MITO[next(iter(codons(s)))] for s in CELLS)==4
assert state['production_certificate']=={'Cys_cell':'UAY','Met_cell':'AUR','closed':True}
assert {w.translate(comp) for w in codons('UGY')}!=codons('AUR')
mito_state=deepcopy(state)
mito_modules=deepcopy(mods)

# Full permitted control curve u(t)=t, including all threshold/tie states.
# All scores are affine; exhaustive breakpoints determine every open regime.
roots={F(0),F(1)}
for m in mods.values():
    for branches in m.values():
```

```
        for r,a,b in branches:
            if b and 0<=-a/b<=1: roots.add(-a/b)
        for (_,a,b),(_,c,d) in combinations(branches,2):
            if b!=d and 0<=(c-a)/(b-d)<=1: roots.add((c-a)/(b-d))
assert sorted(roots)==[F(0),F(1,2),F(1)]
regimes=[]
for u in [F(0),F(1,4),F(1,2),F(3,4),F(1)]:
    K=readout(u)
    assert K['AUG']=={'M'}
    if u<F(1,2): assert singular(K)==MITO
    elif u>F(1,2): assert singular(K)==STD
    else:
        assert K['AUA']=={'I','M'} and K['UGA']==set()
        assert K['AGA']==K['AGG']==set()
    regimes.append({'u':str(u),'sets':{w:sorted(v) for w,v in K.items()}})
assert {w for w in WORDS if MITO[w]!=STD[w]}=={'AGA','AGG','AUA','UGA'}
assert {w[:2]+'R' for w in WORDS if MITO[w]!=STD[w]}=={'AGR','AUR','UGR'}
assert len({(w,C[w]) for C in (MITO,STD) for w in WORDS if C[w]!='*'})==63
assert len([w for w in WORDS if MITO[w]!='*'])==60
assert len([w for w in WORDS if STD[w]!='*'])==61

def control(v):
    assert state['phase'] in ('routing','control') and state['equipped']
    u=state['u']
    assert F(0)<=u<=v<=F(1)
    # The complete segment is admitted by the fixed control schema. Affine
    # roots above certify all regimes, rather than merely these sample points.
    frozen=(deepcopy(state['carrier']),deepcopy(state['decoder']),
            deepcopy(state['a']),deepcopy(state['production_certificate']),deepcopy(mods))
    state['phase']='control';state['u']=v
    assert frozen==(state['carrier'],state['decoder'],state['a'],
                    state['production_certificate'],mods)
    record('control segment '+str(u)+' to '+str(v))
for v in (F(1,4),F(1,2),F(3,4),F(1)): control(v)
assert state['u']==1 and singular(readout(state['u']))==STD
main_trace=deepcopy(trace)
standard_endpoint_state=deepcopy(state)

# Independent all-decreasing, multi-target strict score crossing witness.
crossing=[]
for t in [F(0),F(1,3),F(2,3),F(5,6),F(1)]:
    r=3-2*t; s=2-t/2
    assert r>0 and s>0
    pick={'r'} if r>s else {'s'} if s>r else {'r','s'}
    crossing.append({'t':str(t),'r':str(r),'s':str(s),'selected':sorted(pick)})
assert crossing[0]['selected']==['r'] and crossing[-1]['selected']==['s']

# Generic exchange graph and exact fixed-lock assignment orbit count.
vertices=sorted(sense_cells-LOCKED)
edges=[(s,t) for s,t in combinations(vertices,2)
       if s[2]==t[2] and sum(x!=y for x,y in zip(s[:2],t[:2]))==1]
left=set(vertices); components=[]
while left:
    start=min(left); component={start}; todo=[start]
    while todo:
        x=todo.pop()
        for a,b in edges:
            y=b if a==x else a if b==x else None
            if y is not None and y not in component: component.add(y);todo.append(y)
    left-=component;components.append(sorted(component))
assert sorted(map(len,components))==[2,3,8,8]
orbit_count=1
for component in components:
    labels=[MITO[next(iter(codons(s)))] for s in component]
    assert len(set(labels))==len(labels)
    orbit_count*=factorial(len(component))
assert orbit_count==19508428800
# A concrete alternative obeys the same guards, preserves AUG and the paired
# production certificate, and differs from both named reference tables.
state=deepcopy(mito_state);mods=deepcopy(mito_modules);trace=[]
exchange('UUY','UAY')
ALTERNATIVE=dict(state['a'])
assert ALTERNATIVE!=MITO and ALTERNATIVE!=STD and ALTERNATIVE['AUG']=='M'

out={'status':'all assertions passed','class_counts':{'classes':20,'size2':8,'size4':12},
```

```
 'classes':{str(q):sorted(ws) for q,ws in classes.items()},
 'demands':[{'selected_product':r,'cells':ss,'class':q,'candidates':adj} for r,ss,q,adj in ROWS],
 'matching_count':matching_count,'spare_triples':len(spare_hist),
 'mandatory_classes':sorted(mandatory),'seed_demand_count':17,'seed_cell_count':18,
 'initial_seed_codon_count':18,'propagated_codon_count':52,'final_sense_cell_count':22,
 'final_cell_count':24,'distinct_products':20,'main_path_trace':main_trace,
 'alternative_branch_trace':trace,'endpoint_control':str(standard_endpoint_state['u']),
 'selected':SELECTED,'after_first_exchange':AFTER_ONE,'mitochondrial':MITO,
 'standard':STD,'control_breakpoints':[str(x) for x in sorted(roots)],
 'control_regimes':regimes,'strict_decreasing_crossing':crossing,
 'complementary_seed_cells':complementary,
 'endpoint_differences':{w:[MITO[w],STD[w]] for w in WORDS if MITO[w]!=STD[w]},
 'endpoint_sense_pairs':63,'exchange_components':components,
 'exchange_edges':edges,'locked_cells':sorted(LOCKED),
 'distinct_routing_assignments_at_u0':orbit_count,'alternative_mapping':ALTERNATIVE,
 'source_scripts_executed':False,
 'scope':'formal assignment and affine branch-gating witness; no chemical kinetics or continuous molecular lift
     claimed'}
p=Path(sys.argv[1]) if len(sys.argv)>1 else Path(__file__).with_name('reachability_checks.json')
p.write_text(json.dumps(out,indent=2,sort_keys=True)+'\n')
print('PASS: 17 demands / 18 seeded cells / 22 sense cells; 840 SDRs; two guarded exchanges;')
print('full affine control path (including tie wall); four endpoint codon changes;')
print('63 endpoint sense-pairs; exact locked routing orbit =',orbit_count)
print('Wrote',p)
'''
# END UNMODIFIED SOURCE

# BEGIN UNMODIFIED SOURCE: certificates/junction/verify_junction.py
SOURCES['certificates/junction/verify_junction.py'] = r'''#!/usr/bin/env python3
"""Portable independent finite checks for junction.tex; Python standard library only.
Usage: python3 verify_junction.py [report.json]
Reads only adjacent junction_data.json. No source scripts or external modules run.
The all-run claim is proved by induction in the chapter; tests below are finite.
"""
from collections import Counter
from dataclasses import dataclass, replace
from fractions import Fraction as F
from itertools import product, combinations, permutations
from pathlib import Path
import hashlib, json, sys
```

```
HERE = Path(__file__).resolve().parent
DATA = json.loads((HERE/'junction_data.json').read_text())
B='UCAG'; COMP=dict(zip('ACGU','UGCA')); TAU=dict(zip('ACGU','AGCU'))
def comp(w): return ''.join(COMP[x] for x in w)
def rho(w): return ''.join(TAU[x] for x in w[::-1])
def frame(w,f):
    if f&2: w=rho(w)
    return comp(w) if f&1 else w
ENC={'FF':'A','FB':'C','BF':'G','BB':'U'}
def encode(m):return ''.join(ENC[m[i:i+2]] for i in (0,2,4))
def mf(m,f):
    if f&2:m=m[::-1]
    return ''.join('B' if x=='F' else 'F' for x in m) if f&1 else m
checks=0
for bits in product('FB',repeat=6):
    m=''.join(bits)
    for f in range(4):
        assert encode(mf(m,f))==frame(encode(m),f)
        assert frame(frame(encode(m),f),f)==encode(m)
        checks+=1
assert {frame('UAC',f) for f in range(4)}=={'UAC','AUG','GAU','CUA'}

def family(h):return h[1:3]=='AU' and h[3:5]=='AU' and h[5]==COMP[h[0]]
def boundary(a,f):
    cin='UA'+a;cout='AU'+COMP[a]
    displayed=(frame(cin,f),frame(cout,f))
    normal=tuple(frame(x,f) for x in displayed)
    assert normal==(cin,cout)
    rin=normal[0][2];rout=COMP[normal[1][2]]
    return {'a':a,'b':COMP[a],'frame':f,'displayed':displayed,
            'normalized':normal,'registers':(rin,rout),'release':rin in 'UC' and rout in 'UC'}
gates=[boundary(a,f) for f in range(4) for a in B]
for row in gates:
```

```
    assert row['release']==(row['a'] in 'UC')
    assert frame(row['displayed'][0],1)==row['displayed'][1]
hexamers=[''.join(x) for x in product(B,repeat=6)]
selected=sorted(h for h in hexamers if family(h))
assert selected==sorted(a+'AUAU'+COMP[a] for a in B)
assert len(selected)==4
assert all(not((a in 'UC') and (COMP[a] in 'UC')) for a in B)
width_perms=[];exact_perms=[]
for p in permutations(B):
    t=dict(zip(B,p))
    if all((t[COMP[a]] in 'UC')==(a in 'UC') for a in B):width_perms.append(t)
    if all(t[COMP[a]]==a for a in B):exact_perms.append(t)
assert len(width_perms)==4 and exact_perms==[COMP]

SLOTS=['UUY','UCN','AGY','UAY','UGY','UGR','CUN','CCN','CAY','CGN',
       'AUR','ACN','AAY','GUN','GCN','GAY','GGN','AUY']
def expand(s):return {s[:2]+x for x in {'Y':'UC','R':'AG','N':B}[s[2]]}
pairs=[(s,t) for s,t in combinations(SLOTS,2) if {comp(c) for c in expand(s)}==expand(t)]
assert {frozenset(p) for p in pairs}=={frozenset(p) for p in [('UAY','AUR'),('CCN','GGN'),('CGN','GCN')]}
assert [(s,t) for s,t in pairs if len(expand(s))==len(expand(t))==2]==[('UAY','AUR')]
PER=['6','6a','7','8','9','10','10a','10b','11','12','12a','1','2','3','4','4a','4b','5']
E={frozenset((PER[i],PER[(i+1)%18])) for i in range(18)}
E|={frozenset(e) for e in [('4a','12a'),('4b','10b'),('6a','10a')]}
def comps(nodes,edges):
    remain=set(nodes);out=[]
    while remain:
        stack=[min(remain)];remain.remove(stack[0]);found=[]
        while stack:
            v=stack.pop();found.append(v)
            for e in edges:
                if v in e:
                    for u in sorted(e & remain):remain.remove(u);stack.append(u)
        out.append(sorted(found))
    return sorted(out,key=lambda c:(len(c),c))
remnants={
 'Cys59':comps(set(PER)-{'5'},E),
 'Cys57':comps(set(PER)-{'5','6','4b'},E),
 'Met':comps(set(PER)-{'5','4b','10b'},E)}
assert [list(map(len,remnants[k])) for k in ('Cys59','Cys57','Met')]==[[17],[15],[7,8]]
d,h,ell=map(F,('3.40','1.68','2.05'))
assert ell**2-(d-h)**2==F(12441,10000) and (d+h)>ell

# Recompute source-defined endpoint index contributions from elemental bonds.
C=lambda n=1:['C',n]; H=lambda n=1:['H',n]; O=lambda n=1:['O',n]
N=lambda n=1:['N',n]; S=lambda n=1:['S',n]
ALC=[C(),O(),H(2)];ALD=[C(),O(2),H()]
X3=[('5',[C(),C(),H()],[C(),S(),H(2)]),
 ('4b',[C(),C(),C()],[C(),C(),H()]),('6',[C(),C(),O()],ALC),
 ('Ca',[O(),C(),H(2)],[N(),C(),C(),H()]),('Cc',[O(),C(),H(2)],[O(2),O(),C()])]
X4=[('5',[C(),C(),H()],[C(),C(),H(2)]),('4b',[C(),C(),C()],[C(),S(),H(2)]),
 ('10b',[C(),C(),C()],[S(),H(3)]),('11',[C(),C(),H()],[C(),H(2)]),
 ('4a',[C(),C(),C()],[C(),C(),H()]),('10a',[C(),C(),C()],[C(),C(),H()]),
 ('6',[C(),C(),O()],ALC),('Ca',[O(),C(),H(2)],[N(),C(),C(),H()]),
 ('Cc',[O(),C(),H(2)],[O(2),O(),C()])]
def omega(bs):return sum(n*({'H':-1,'C':0,'S':1,'N':1,'O':1}[e]) for e,n in bs)
redox=[]
for k3,k4 in product((0,1),repeat=2):
    c3=sum(omega(ALD if v=='6' and k3 else aft)-omega(bef) for v,bef,aft in X3)
    c4=sum(omega(ALD if v=='6' and k4 else aft)-omega(bef) for v,bef,aft in X4)
    assert (c3,c4)==(2+2*k3,-4+2*k4)
    redox.append({'k3':k3,'k4':k4,'carbon':[c3,c4],'with_sulfur':[c3-1,c4-1],'total':c3+c4-2})
assert redox[-1]['with_sulfur']==[3,-3]

# Exact constitutional identity via an explicit rooted graph isomorphism.
BB=[('N','Ca',1),('Ca','Cc',1),('Cc','Oa',1),('Cc','Ob',2)]
REFS={
 'Cys':{'atoms':{'N':'N','Ca':'C','Cc':'C','Oa':'O','Ob':'O','Cb':'C','S':'S'},
        'bonds':BB+[('Ca','Cb',1),('Cb','S',1)]},
 'Met':{'atoms':{'N':'N','Ca':'C','Cc':'C','Oa':'O','Ob':'O','Cb':'C','Cg':'C','S':'S','Ce':'C'},
        'bonds':BB+[('Ca','Cb',1),('Cb','Cg',1),('Cg','S',1),('S','Ce',1)]},
 'Ala':{'atoms':{'N':'N','Ca':'C','Cc':'C','Oa':'O','Ob':'O','Cb':'C'},
        'bonds':BB+[('Ca','Cb',1)]}}
def edgekey(e):return tuple(sorted(e[:2]))+(e[2],)
def identify(cert):
    mapping=cert['identity_map']
```

```
    mapped_atoms={mapping[a]:e for a,e in cert['atoms'].items()}
    mapped_bonds={edgekey((mapping[a],mapping[b],n)) for a,b,n in cert['bonds']}
    names=[name for name,ref in REFS.items() if mapped_atoms==ref['atoms'] and mapped_bonds==set(map(edgekey,ref['bonds']))]
    assert len(names)==1
    return names[0]
for name,cert in DATA['identity_projections'].items():assert identify(cert)==name
assert identify(DATA['identity_projections']['Met'])=='Met'

# Independent replay of the explicitly relaxed paired graph path, not a molecular path.
P=DATA['paired_sulfur'];caps={'C':4,'N':3,'O':2,'S':2,'H':1,'*':1}
atoms=P['initial']['atoms'];assert atoms==P['endpoint']['atoms']
def bd(g):
    result={tuple(sorted((a,b))):n for a,b,n in g['bonds']}
    assert len(result)==len(g['bonds'])
    return result
def degree(a,bonds):return sum(n for e,n in bonds.items() if a in e)
def valid(bonds,closed=False):
    assert all(a!=b and a in atoms and b in atoms and 1<=n<=3 for (a,b),n in bonds.items())
    assert all(degree(a,bonds)<=caps[e] for a,e in atoms.items())
    if closed:assert all(degree(a,bonds)==caps[e] for a,e in atoms.items())
cur=bd(P['initial']);valid(cur,True)
assert sum(e=='S' for e in atoms.values())==2
assert cur[tuple(sorted(('X3:S1','X4:S1')))]==1
stage_by_step={v['after_step']:(k,v) for k,v in P['stages'].items()}
for step,(op,a,b) in enumerate(P['rewrite_steps'],1):
    edge=tuple(sorted((a,b)));assert a in atoms and b in atoms and a!=b
    if op=='split':
        assert cur.get(edge,0)>0
        cur[edge]-=1
        if cur[edge]==0:del cur[edge]
    else:
        assert op=='join' and degree(a,cur)<caps[atoms[a]] and degree(b,cur)<caps[atoms[b]]
        cur[edge]=cur.get(edge,0)+1
    valid(cur)
    if step in stage_by_step:
        name,stage=stage_by_step[step]
        assert stage['graph']['atoms']==atoms and bd(stage['graph'])==cur
        if name=='disulfide_bridge':
            path=stage['path']
            assert path==['X3:5','X3:S1','X4:S1','X4:4b']
            assert all(cur.get(tuple(sorted(e)))==1 for e in zip(path,path[1:]))
            assert tuple(sorted(('X4:4b','X4:10b'))) not in cur
        elif name=='sulfur_topologies':
            assert tuple(sorted(('X3:S1','X4:S1'))) not in cur
            assert cur[tuple(sorted(('X4:S1','X4:10b')))]==1
assert cur==bd(P['endpoint']);valid(cur,True)
for prefix,name in [('X3:','Cys'),('X4:','Met')]:
    component=next(c for c in comps(atoms, {frozenset(e) for e in cur}) if prefix+'Ca' in c)
    hv={a for a in component if atoms[a]!='H'}
    assert all(a.startswith(prefix) for a in hv)
    cert=DATA['identity_projections'][name]
    projected_atoms={a[len(prefix):]:atoms[a] for a in hv}
    projected_bonds={edgekey((a[len(prefix):],b[len(prefix):],n)) for (a,b),n in cur.items() if a in hv and b in hv}
    assert projected_atoms==cert['atoms']
    assert projected_bonds==set(map(edgekey,cert['bonds']))
```

```
# Recompute full conventional endpoint tables and common-codon stage interface.
words=[''.join(x) for x in product(B,repeat=3)]
std=dict(zip(words,'FFLLSSSSYY**CC*W' 'LLLLPPPPHHQQRRRR' 'IIIMTTTTNNKKSSRR' 'VVVVAAAADDEEGGGG'))
mito=dict(std, AUA='M', UGA='W', AGA='*', AGG='*')
sel=dict(mito)
def swap(table,a,b):
    for x,y in zip(sorted(expand(a)),sorted(expand(b))):table[x],table[y]=table[y],table[x]
swap(sel,'UAY','UGY');swap(sel,'AAR','CAR')
after_sulfur=dict(sel);swap(after_sulfur,'UAY','UGY')
after_amides=dict(after_sulfur);swap(after_amides,'AAR','CAR');assert after_amides==mito
stages={'selected':sel,'sulfur_exchange':after_sulfur,'mitochondrial':mito,'standard':std}
for table in stages.values():assert [table[c] for c in ('AUG','GCU','UAA')]==['M','A','*']
assert {comp(c) for c in expand('UGY')}==expand('ACR')
assert expand('ACR') != expand('AUR')
assert {c for c in std if std[c]=='M'}=={'AUG'}
# Along the affine control path the changing support is disjoint from common tape.
```

```
changed={c for c in std if std[c]!=mito[c]}
assert changed=={'AUA','UGA','AGA','AGG'} and not (changed & {'AUG','GCU','UAA'})

@dataclass(frozen=True)
class State:
    mode:str='Idle'
    remaining:tuple=()
    used:tuple=()
    packet:int=-1
    phase:int=-1
    pos:int=0
    word:tuple=()
    done:tuple=()
    archived_packets:tuple=()
    consumed:tuple=()
    a:str='C'
    f:int=0
    stage:str='standard'
    tape:tuple=('AUG','GCU','UAA')
    obs:str=''
    outid:str=''

def invariant(s):
    assert len(s.used)==len(set(s.used)) and not(set(s.used)&set(s.remaining))
    assert len(s.consumed)==len(set(s.consumed))
    assert all(w and w[0]=='Met' for w in s.done)
    assert len(s.done)==len(s.archived_packets)<=len(s.used)
    assert len(set(s.archived_packets))==len(s.archived_packets)
    assert set(s.archived_packets)<=set(s.used)
    if s.mode in ('Active','Stopped','Released'):
        assert s.word and s.word[0]=='Met' and s.phase==s.f and s.pos%3==0
    if s.mode in ('Ready','Formed','Primed','Rejected'):
        assert not s.word and s.phase==-1
    return True

def transition(s,rule,**kw):
    """Guarded controller projection. FORM imports an independently checked graph certificate."""
    t=s
    if rule=='RESET':
        j=kw['packet'];assert s.mode in ('Idle','Halted','Rejected') and j in s.remaining and j not in s.used and
     not s.word
        t=replace(s,mode='Ready',remaining=tuple(x for x in s.remaining if x!=j),used=s.used+(j,),packet=j,
                  phase=-1,pos=0,word=(),a=kw.get('a','C'),f=kw.get('frame',0),obs='',outid='',
                  tape=kw.get('tape',('AU'+COMP[kw.get('a','C')],'GCU','UAA')))
    elif rule=='FORM':
        assert s.mode=='Ready' and family(s.a+'AUAU'+COMP[s.a])
        # The template is copied with a fresh packet prefix; the identity map is unchanged.
        t=replace(s,mode='Formed',outid=identify(DATA['identity_projections']['Met']),obs='Bound')
    elif rule=='GATE':
        assert s.mode=='Formed' and boundary(s.a,s.f)['release']
        t=replace(s,mode='Primed',obs='Free')
    elif rule=='FAIL':
        assert s.mode=='Formed' and not boundary(s.a,s.f)['release']
        t=replace(s,mode='Rejected',obs='Bound')
    elif rule=='START':
        tok=f'{s.packet}:out';assert s.mode=='Primed' and s.obs=='Free' and not s.word and s.phase==-1 and tok not
      in s.consumed
        # Endpoint policy chooses AUG; selected/mitochondrial mode also permits AUA.
        assert s.a=='C' or (s.a=='U' and s.stage in ('selected','mitochondrial','sulfur_exchange'))
        assert s.outid==identify(DATA['identity_projections']['Met']) and s.tape[0]=='AU'+COMP[s.a]
        t=replace(s,mode='Active',word=(s.outid,),phase=s.f,pos=3,consumed=s.consumed+(tok,))
    elif rule=='STEP':
        assert s.mode=='Active' and s.phase==s.f and s.pos%3==0
        assert s.pos//3<len(s.tape) and s.tape[s.pos//3]=='GCU'
        tok=f'{s.packet}:ala';assert tok not in s.consumed and stages[s.stage]['GCU']=='A'
        aa=identify(DATA['identity_projections']['Ala'])
        t=replace(s,word=s.word+(aa,),pos=s.pos+3,consumed=s.consumed+(tok,))
    elif rule=='OBSERVE':
        assert s.mode=='Active' and s.phase==s.f and s.pos%3==0 and stages[s.stage]['UAA']=='*'
        assert s.pos//3<len(s.tape) and s.tape[s.pos//3]=='UAA'
        assert kw.get('control_certificate',True)
        t=replace(s,mode='Stopped',obs='StopSignal')
    elif rule=='RELEASE':
        assert s.mode=='Stopped' and kw.get('release_certificate',True)
        t=replace(s,mode='Released',obs='SequenceReleased')
```

```
    elif rule=='HALT':
        assert s.mode=='Released' and kw.get('halt_certificate',True)
        t=replace(s,mode='Halted',done=s.done+(s.word,),archived_packets=s.archived_packets+(s.packet,),word=(),
     phase=-1,packet=-1,obs='',outid='')
    elif rule=='ROUTE':
        assert kw.get('sealed',True) and kw['stage'] in stages
        t=replace(s,stage=kw['stage'])
    else: raise AssertionError('undeclared rule')
    invariant(t);return t

def bad(s,rule,**kw):
    try:transition(s,rule,**kw)
    except AssertionError:return True
    return False

# Constructed n=3 trace; each larger n uses the identical fresh-index induction.
s=State(remaining=(0,1,2));trace=[]
for j in range(3):
    for rule,kwargs in [('RESET',{'packet':j,'a':'C','frame':j%4}),('FORM',{}),('GATE',{}),('START',{}),
                        ('STEP',{}),('OBSERVE',{}),('RELEASE',{}),('HALT',{})]:
        s=transition(s,rule,**kwargs)
        trace.append({'rule':rule,'mode':s.mode,'packet':s.packet,'word':s.word,'done':s.done,'used':s.used,'pos':
     s.pos})
assert s.done==(('Met','Ala'),)*3 and not s.remaining
assert bad(s,'RESET',packet=0)

# Minimal three-packet witness uses no elongation tokens.
single=State(remaining=(0,1,2))
for j in range(3):
    single=transition(single,'RESET',packet=j,a='C',frame=j,tape=('AUG','UAA'))
    for rule in ('FORM','GATE','START','OBSERVE','RELEASE','HALT'):single=transition(single,rule)
assert single.done==(('Met',),)*3 and len(single.consumed)==3

# All 4 frames x 4 boundary letters x 4 initialized stages, and 0/1 continuation.
case_count=accepted=0
for f,a,stage,elong in product(range(4),B,stages,range(2)):
    z=State(remaining=(0,),stage=stage)
    assert bad(z,'START') and bad(z,'STEP')
    z=transition(z,'RESET',packet=0,a=a,frame=f,tape=('AU'+COMP[a],)+(('GCU',) if elong else ())+('UAA',))
    assert bad(z,'STEP')
    z=transition(z,'FORM')
    if a in 'AG':
        assert bad(z,'GATE');z=transition(z,'FAIL');assert not z.done
    else:
        assert bad(z,'FAIL');z=transition(z,'GATE')
        if a=='U' and stage=='standard':assert bad(z,'START')
        else:
            z=transition(z,'START')
            assert bad(z,'RESET',packet=0) and bad(z,'HALT')
            if elong:z=transition(z,'STEP')
            assert bad(replace(z,pos=z.pos+1),'STEP')
            assert bad(z,'OBSERVE',control_certificate=False)
            z=transition(z,'OBSERVE');assert bad(z,'STEP') and bad(z,'HALT')
            assert bad(z,'RELEASE',release_certificate=False)
            z=transition(z,'RELEASE');assert bad(z,'HALT',halt_certificate=False)
            z=transition(z,'HALT');assert z.done[0][0]=='Met';accepted+=1
    case_count+=1

report={'status':'PASS','scope':'Finite formal verification only; all-run property proved by transition induction.
     ',
 'marker_frame_conjugacy_checks':checks,'UAC_orbit':sorted({frame('UAC',f) for f in range(4)}),
 'hexamers_checked':len(hexamers),'selected_family':selected,'framed_gate_cases':gates,
 'width_covariant_outgoing_permutations':len(width_perms),'exact_register_transport_unique':True,
 'complementary_pairs':pairs,'remnants':remnants,'rigid_lateral_squared':str(ell**2-(d-h)**2),
 'redox_cases':redox,'identity_labels_checked':sorted(DATA['identity_projections']),
 'paired_graph_rewrite_steps':len(P['rewrite_steps']),'paired_graph_atom_count':len(atoms),
 'paired_bridge_stage_steps':{k:v['after_step'] for k,v in P['stages'].items()},
 'paired_graph_replay_scope':'Relaxed valence-port grammar, not molecular admissibility; FORM controller
     projection imports its certificate.',
 'stage_tape_cases':{k:{c:t[c] for c in ('AUG','GCU','UAA')} for k,t in stages.items()},
 'controller_case_count':case_count,'accepted_cases':accepted,'three_segment_trace':trace,'
     minimal_singleton_witness':single.done,
 'no_packet_reuse':True,'no_emission_before_start':True,'stop_release_halt_distinct':True,
 'source_data_sha256':hashlib.sha256((HERE/'junction_data.json').read_bytes()).hexdigest()}
output=Path(sys.argv[1]) if len(sys.argv)>1 else HERE/'junction_checks.json'
```

```
output.write_text(json.dumps(report,indent=2,sort_keys=True)+'\n')
print(json.dumps({k:report[k] for k in ('status','marker_frame_conjugacy_checks','hexamers_checked','
    controller_case_count','accepted_cases')},indent=2))
'''
# END UNMODIFIED SOURCE

# BEGIN UNMODIFIED SOURCE: scripts/validate_product_guards.py
SOURCES['scripts/validate_product_guards.py'] = r'''#!/usr/bin/env python3
"""Consume fixed product JSON and audit the written extraction guards.

Independent of the certificate builder: no constructor or generated function is
imported. The audit recognizes the permitted masks and replays all unit edits.
This remains a formal graph audit, not chemical or stereochemical validation.
"""
from pathlib import Path
import copy
import json

ROOT = Path(__file__).resolve().parents[1]
PER = ('6','6a','7','8','9','10','10a','10b','11','12','12a','1','2','3','4','4a','4b','5')
B6 = ('4a','4b','10b','11','12','12a')
A6 = {'1','2','3','4','4a','12a'}
CAP = {'C':4,'N':3,'O':2,'S':2,'H':1,'*':1}

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

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

E = {edge(PER[i],PER[(i+1)%18]) for i in range(18)} | {
    edge('4a','12a'),edge('4b','10b'),edge('6a','10a')}

def bondmap(graph):
    bonds = {edge(a,b):v for a,b,v in graph['bonds']}
    require(len(bonds)==len(graph['bonds']), 'Duplicate bond records')
    return bonds

def mask_allowed(S):
    if not S or S == {'5','4b','10b'} | A6:
        return True
    for optional6 in (set(), {'6'}):
        if S == {'5'} | optional6:
            return True
        for length in range(1,7):
            for start in range(6):
                arc = {B6[(start+i)%6] for i in range(length)}
                if '4b' in arc and S == {'5'} | optional6 | arc:
                    return True
    return False

def validate_mask(S,W):
    require(mask_allowed(S), 'Mask is outside the ordinary/empty/extended family')
    require(W <= set(PER)-S and len(W)<=2, 'Whole-removal set is invalid')
    require(all(any(edge(w,s) in E for s in S) for w in W),
            'Whole-removed vertex is not neighboring the retained mask')

def component(atoms,bonds,root):
    seen={root}; todo=[root]
    while todo:
        current=todo.pop()
        for a,b in bonds:
            if a==current and b not in seen: seen.add(b); todo.append(b)
            if b==current and a not in seen: seen.add(a); todo.append(a)
    return seen

def valid(atoms,bonds,closed=False):
    degree={a:0 for a in atoms}
    for (a,b),v in bonds.items():
        require(a in atoms and b in atoms and a!=b, 'Invalid bond endpoint')
        require(type(v) is int and 1<=v<=3, 'Invalid bond order')
        degree[a]+=v; degree[b]+=v
    require(all(degree[a]<=CAP[e] for a,e in atoms.items()), 'Valence capacity exceeded')
```

```
    if closed:
        require(all(degree[a]==CAP[e] for a,e in atoms.items()), 'Unclosed terminal valence')

def replay(certificate, specs):
    """specs maps module prefixes to the chosen fixed S and W records."""
    atoms=certificate['initial']['atoms']; terminal=certificate['endpoint']['atoms']
    require(atoms==terminal, 'Atom identities or elements changed')
    initial=bondmap(certificate['initial']); final=bondmap(certificate['endpoint'])
    valid(atoms,initial,True); valid(atoms,final,True)
    owner={p+c:(p,c) for p in specs for c in PER}
    protected={}
    for prefix,(S,W) in specs.items():
        validate_mask(S,W)
        for a,b,v in [('Ca','Cc',1),('Ca','Na',1),('Cc','Ox',2)]:
            protected[edge(prefix+a,prefix+b)]=v
        if S: protected[edge(prefix+'5',prefix+'Ca')]=1
        for a,b in E:
            e=edge(prefix+a,prefix+b)
            if not ({a,b}&(S|W)):
                protected[e]=initial[e]
        boundary={edge(prefix+a,prefix+b) for a,b in E if (a in S)!=(b in S)}
        require(not (boundary & final.keys()), 'Extraction boundary not severed')
        lost=[(a,b) for a,b in E if {a,b}<=S and edge(prefix+a,prefix+b) not in final]
        require(len(lost)<=1, 'More than one retained adjacency deleted')
        for w in W:
            for e in initial:
                if prefix+w in e and all(atoms[a]!='H' for a in e):
                    require(e not in final, 'Whole-removed original heavy incidence remains')
        product=component(atoms,final,prefix+'Ca')
        require({a for a in product if atoms[a]=='C'} == {prefix+a for a in S|{'Ca','Cc'}},
                'Product carbon membership is wrong')
        require(all(atoms[a]!='*' for a in product), 'Product contains a boundary port')
        require(final.get(edge(prefix+'Cc',prefix+'On0'))==1 and
                edge(prefix+'Cc',prefix+'Ot') not in final, 'Free-acid release completion is wrong')

    # Every new side-chain heavy incidence is local to the retained mask,
    # finite donor stock, or backbone N closure; Cc-On0 is the declared release exception.
    donors={p+x for p in specs for x in ('Nn1','Nn2','Nn3','On0','On1','On2','S1','S2') if p+x in atoms}
    allowed_heavy={p+x for p,(S,W) in specs.items() for x in S|{'Na'}} | donors
    release_edges={edge(p+'Cc',p+'On0') for p in specs}
    closure_edges={edge(p+'Na',p+x) for p,(S,W) in specs.items() for x in S}
    def edit_guard(kind,a,b,current):
        e=edge(a,b)
        require(e not in protected, 'Edit changes a protected bond')
        if a in owner and b in owner:
            p,x=owner[a]; q,y=owner[b]
            require(p==q and edge(x,y) in E, 'New or cross-copy core adjacency')
            S,W=specs[p]
            require(bool({x,y}&(S|W)), 'Core edit outside local support')
        if kind=='join' and e not in initial and all(atoms[x] not in ('H','*') for x in e):
            if not (a in owner and b in owner):
                require(e in release_edges or e in closure_edges or
                        (bool(set(e)&donors) and set(e)<=allowed_heavy),
                        'New heavy incidence violates donor/locality guard: '+str(e))

    current=dict(initial)
    for i,(kind,a,b) in enumerate(certificate['rewrite_steps'],1):
        require(kind in ('split','join'), 'Unknown primitive')
        require(a in atoms and b in atoms and a!=b, 'Primitive has invalid atoms')
        edit_guard(kind,a,b,current)
        e=edge(a,b)
        if kind=='split':
            require(current.get(e,0)>0, 'Split without a positive bond')
            current[e]-=1
            if not current[e]: del current[e]
        else: current[e]=current.get(e,0)+1
        valid(atoms,current)
        require(all(current.get(e)==v for e,v in protected.items()),
                'Protected incidence differs during replay')
    require(current==final, 'Replayed endpoint differs from supplied endpoint')
    return len(certificate['rewrite_steps'])

def validate_all(data):
    totals={}
    for name,row in data['rows'].items():
```

```
        S=set(row['source57']['retained'])
        W=set() if name=='Cys' else set(row['source57']['removed_entire'])
        totals[name]=replay(row,{'':(S,W)})
        require(component(row['endpoint']['atoms'],bondmap(row['endpoint']),'Ca')==set(row['product_vertices']),
                'Declared product vertex set is not its full component')
    totals['paired_sulfur']=replay(data['paired_sulfur'],{
        'X3:':({'5'},set()), 'X4:':({'5','4b','10b'},set())})
    return totals

def rejected(action):
    try: action()
    except ValueError: return True
    return False

def main():
    data=json.loads((ROOT/'certificates/products/product_certificates.json').read_text())
    counts=validate_all(data)
    tests={
        'invalid_mask':rejected(lambda:validate_mask({'5','1'},set())),
        'nonneighbor_removal':rejected(lambda:validate_mask({'5'},{'10'})),
    }
    for name,edit in {
        'protected_backbone_edit':['split','Ca','Cc'],
        'new_core_adjacency':['join','5','10'],
        'out_of_support_edit':['split','1','2'],
        'donor_to_remnant':['join','On1','1'],
    }.items():
        mutant=copy.deepcopy(data['rows']['Ala'])
        mutant['rewrite_steps'].insert(0,edit)
        tests[name]=rejected(lambda m=mutant:replay(m,{'':({'5'},{'6','4b'})}))
    mutant=copy.deepcopy(data['paired_sulfur'])
    mutant['rewrite_steps'].insert(0,['split','X4:Ca','X4:Cc'])
    tests['paired_protected_edit']=rejected(lambda:replay(mutant,{'X3:':({'5'},set()),'X4:':({'5','4b','10b'},set
     ())}))
    require(all(tests.values()), 'A representative invalid certificate was accepted')
    report={'status':'PASS','scope':'Data-consuming formal extraction-guard replay; no molecular feasibility
     claim',
            'certificate_count':len(counts),'primitive_counts':counts,'rejection_tests':tests,
            'guard_checks':['mask family','whole-removal neighbors','retained internal-edge budget',
                            'atom identity and capacity','protected bonds at every step','core support and
     original adjacency',
                            'finite donor heavy-atom incidence','free-acid completion','full terminal product
     components',
                            'paired sulfur path with distinct core copies']}
    (ROOT/'data/product_guard_audit.json').write_text(json.dumps(report,indent=2,sort_keys=True)+'\n')
    print('PASS: all 20 row witnesses and the paired sulfur path satisfy the extraction guards; 7 rejection tests
     pass.')

if __name__=='__main__': main()
'''
# END UNMODIFIED SOURCE
```

```
# BEGIN UNMODIFIED SOURCE: verify_manuscript.py
SOURCES['verify_manuscript.py'] = r'''#!/usr/bin/env python3
"""Rerun packaged finite certificates and check their shared interfaces.

This verifies the declared formal graph-and-controller model. It intentionally
makes no chemical feasibility, kinetic, stereoselective, or uniqueness claim.
"""
from pathlib import Path
import hashlib
import itertools
import json
import shutil
import subprocess
import sys
import tempfile

ROOT = Path(__file__).resolve().parent
AA3_TO_1 = dict(zip(
    'Ala Cys Asp Glu Phe Gly His Ile Lys Leu Met Asn Pro Gln Arg Ser Thr Val Trp Tyr'.split(),
    'ACDEFGHIKLMNPQRSTVWY'))
MODULES = {
    'registry': ('verify_registry.py', ['registry_data.json', 'marker_incidence.csv', 'orbit_incidence.csv']),
```

```
    'products': ('build_product_certificates.py', ['product_certificates.json', 'product_verification_summary.json']),
    'reachability': ('verify_reachability.py', ['reachability_checks.json']),
    'junction': ('verify_junction.py', ['junction_checks.json']),
}

def require(test, message):
    if not test:
        raise AssertionError(message)

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

def load(path):
    return json.loads(path.read_text())

def main():
    reports = {}
    audit = subprocess.run([sys.executable, str(ROOT/"scripts/validate_product_guards.py")],
                           cwd=ROOT, text=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    require(audit.returncode == 0, "Extraction-guard audit failed:\n"+audit.stdout)
    with tempfile.TemporaryDirectory(prefix='chrysene-certificate-') as tmp:
        for name, (script, outputs) in MODULES.items():
            src = ROOT / 'certificates' / name
            dst = Path(tmp) / name
            shutil.copytree(src, dst)
            proc = subprocess.run([sys.executable, str(dst / script)], cwd=dst,
                                  text=True, stdout=subprocess.PIPE,
                                  stderr=subprocess.STDOUT, check=False)
            require(proc.returncode == 0, f'{name} verification failed:\n{proc.stdout}')
            for output in outputs:
                require((dst / output).read_bytes() == (src / output).read_bytes(),
                        f'{name}/{output} is not the deterministic regenerated result')
            reports[name] = {'status': 'PASS', 'deterministic_outputs': outputs}

    reg = load(ROOT/'certificates/registry/registry_data.json')
    products = load(ROOT/'certificates/products/product_certificates.json')
    ps = load(ROOT/'certificates/products/product_verification_summary.json')
    reach = load(ROOT/'certificates/reachability/reachability_checks.json')
    jd = load(ROOT/'certificates/junction/junction_data.json')
    jc = load(ROOT/'certificates/junction/junction_checks.json')
    rows = products['rows']
    require(set(rows) == set(AA3_TO_1), 'Product library must contain exactly the named 20 constitutional targets')
    require(set(AA3_TO_1.values()) == set(reg['amino_acids']), 'Registry and product name alphabets disagree')
    require(reg['standard_code'] == reach['standard'], 'Registry and routing standard inputs disagree')
    codons = {''.join(x) for x in itertools.product('ACGU', repeat=3)}
    for name in ('standard', 'mitochondrial'):
        table = reach[name]
        require(set(table) == codons, f'{name} is not total on all 64 codons')
        require(set(table.values()) == set(AA3_TO_1.values()) | {'*'}, f'{name} product alphabet mismatch')
        require(table['AUG'] == 'M' and table['UAA'] == '*', f'{name} fails common singleton-Met trace interface')

    ncbi = load(ROOT/'data/ncbi_reference_tables.json')
    order = [''.join(c) for c in itertools.product(ncbi['base_order'], repeat=3)]
    for row in ncbi['tables'].values():
        require(len(row['aas']) == 64, 'NCBI snapshot must have 64 assignments')
        require(dict(zip(order, row['aas'])) == reach[row['name']],
                'NCBI reference snapshot disagrees with packaged endpoint')
    differences = sorted(c for c in codons if reach['standard'][c] != reach['mitochondrial'][c])
    require(differences == ['AGA', 'AGG', 'AUA', 'UGA'], 'Wrong endpoint difference set')
    for name, projection in jd['identity_projections'].items():
        row = rows[name]
        atoms = row['endpoint']['atoms']
        heavy = {v: atoms[v] for v in row['product_vertices'] if atoms[v] not in ('H', '*')}
        require(heavy == projection['atoms'], f'{name} junction atom projection disagrees with product certificate')
        require(sorted(map(tuple,row['product_heavy_bonds'])) == sorted(map(tuple,projection['bonds'])),
                f'{name} junction bond projection disagrees with product certificate')
        require(row['product_formula'] == projection['product_formula'], f'{name} formula mismatch')
    require(products['paired_sulfur']['bridge_ordered'], 'Missing paired sulfur bridge stage')
    require(ps['all_checks_passed'] if 'all_checks_passed' in ps else ps['checks_passed'], 'Product certificate did not pass')
    require(ps['stereo_selection_certified'] is False and ps['molecular_feasibility_certified'] is False,
            'Scope flags must not overstate the graph certificate')
```

```
    require(reg['minimal_refinement']['cell_count'] == 61, 'Wrong minimal refinement bound')
    require('CAUAUG' in jc['selected_family'], 'Approved physical Option B witness is absent')
    complement = dict(zip('ACGU','UGCA'))
    h = 'CAUAUG'
    require(h[2]+h[1]+h[0] == 'UAC' and h[3:] == 'AUG', 'Outer-index readout is inconsistent')
    require(h[5] == complement[h[0]], 'Outer switches are not complementary')
    require(jc['no_packet_reuse'] and jc['no_emission_before_start'], 'Controller violates fresh/start invariant')

    # Check the directed Met heavy graph independently of its residue-name string.
    met = jd['identity_projections']['Met']
    mb = {tuple(sorted((u,v))): order for u,v,order in met['bonds']}
    for u,v,order in [('Ca','5',1),('5','4b',1),('4b','S1',1),('S1','10b',1),
                      ('Ca','Na',1),('Ca','Cc',1),('Cc','Ox',2),('Cc','On0',1)]:
        require(mb.get(tuple(sorted((u,v)))) == order, 'Met constitutional chain/backbone check failed')

    inputs = {}
    for p in sorted((ROOT/'certificates').rglob('*')):
        if p.is_file() and p.suffix in ('.json','.py','.csv'):
            inputs[str(p.relative_to(ROOT))] = digest(p)
    report = {
        'status': 'PASS',
        'scope': 'Joint finite and restricted formal graph/controller certificate only',
        'modules': reports,
        'extraction_guard_audit': load(ROOT/'data/product_guard_audit.json'),
        'independent_reference_tables': {'source': ncbi['source_url'], 'update': ncbi['page_update'],
                                         'accessed': ncbi['accessed'], 'both_64_entry_comparisons': 'PASS'},
        'shared_interfaces': {
            'product_graph_count': 20,
            'named_amino_acid_alphabet': sorted(reg['amino_acids']),
            'registry_and_routing_reference_equal': True,
            'both_endpoint_codon_counts': 64,
            'endpoint_difference_codons': differences,
            'registry_preserving_endpoint_minimum_states': 61,
            'presented_sixword_witness': h,
            'normalized_incoming_outgoing': ['UAC','AUG'],
            'junction_graph_projections_match_products': True,
            'paired_sulfur_bridge_checked': True,
            'AUG_start_UAA_stop_common_to_named_endpoints': True,
            'resource_packets_not_reused': True,
        },
        'not_certified': ['precursor-to-carrier molecular atom map', 'chemical reaction feasibility',
                          'stereoselective synthesis', 'kinetics or numerical yields',
                          'physical docking/readout/reset', 'target-independent or unique code selection'],
        'input_sha256': inputs,
    }
    path = ROOT/'verified_results.json'
    path.write_text(json.dumps(report, indent=2, sort_keys=True)+'\n')
    print('PASS: all four modules and their shared product/frame/endpoint/controller interfaces.')
    print('Wrote', path.name)

if __name__ == '__main__':
    main()
'''
# END UNMODIFIED SOURCE
```

```
FIXTURES = json.loads(r'''{
  "certificates/products/source_retention_rows.json": {
    "construction_rows": [
      {
        "target": "Ala",
        "sulfur": false,
        "retained": [
          "5"
        ],
        "severed": [],
        "removed_entire": [
          "6",
          "4b"
        ],
        "selected_codons": [
          "GCA",
          "GCC",
          "GCG",
          "GCU"
```

```
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            null
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Arg",
        "sulfur": false,
        "retained": [
          "10b",
          "11",
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "10b",
            "10a"
          ],
          [
            "11",
            "12"
          ],
          [
            "10b",
            "11"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "CGA",
          "CGC",
          "CGG",
          "CGU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              2
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Asn",
        "sulfur": false,
        "retained": [
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
```

```
            "4b",
            "4a"
          ],
          [
            "4b",
            "10b"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "AAC",
          "AAU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              0
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Asp",
        "sulfur": false,
        "retained": [
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "4b",
            "10b"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "GAC",
          "GAU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              0
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Cys",
        "sulfur": true,
        "retained": [
          "5"
        ],
        "severed": [],
        "removed_entire": [
          "6",
          "4b"
```

```
        ],
        "selected_codons": [
          "UAC",
          "UAU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            null
          ]
```

```
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Gln",
        "sulfur": false,
        "retained": [
          "10b",
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "10b",
            "10a"
          ],
          [
            "10b",
            "11"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "AAA",
          "AAG"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              1
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Glu",
        "sulfur": false,
        "retained": [
          "4a",
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
```

```
            "4b",
            "10b"
          ],
          [
            "4a",
            "4"
          ],
          [
            "4a",
            "12a"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "GAA",
          "GAG"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              1,
              0
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Gly",
        "sulfur": false,
        "retained": [],
        "severed": [],
        "removed_entire": [],
        "selected_codons": [
          "GGA",
          "GGC",
          "GGG",
          "GGU"
        ],
        "literal_retention_fit": false,
        "canonical_retention_parameters": [],
        "cut_boundary_complete": true,
        "construction_status": "empty-side-chain backbone constructor required"
      },
      {
        "target": "His",
        "sulfur": false,
        "retained": [
          "10b",
          "11",
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "10b",
            "10a"
          ],
          [
            "11",
            "12"
          ],
          [
            "10b",
```

```
            "11"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "CAC",
          "CAU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              2
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Ile",
        "sulfur": false,
        "retained": [
          "10b",
          "4b",
          "5",
          "6"
        ],
        "severed": [
          [
            "6",
            "6a"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "10b",
            "10a"
          ],
          [
            "10b",
            "11"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "AUC",
          "AUU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            1,
            [
              0,
              1
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Leu",
        "sulfur": false,
        "retained": [
          "10b",
          "4a",
          "4b",
```

```
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4a",
            "4"
          ],
          [
            "4a",
            "12a"
          ],
          [
            "10b",
            "11"
          ],
          [
            "10b",
            "10a"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "CUA",
          "CUC",
          "CUG",
          "CUU",
          "UUA",
          "UUG"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              1,
              1
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Lys",
        "sulfur": false,
        "retained": [
          "10b",
          "11",
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "10b",
            "10a"
          ],
          [
            "11",
            "12"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
```

```
          "CAA",
          "CAG"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              2
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Met",
        "sulfur": true,
        "retained": [
          "10b",
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "10b",
            "10a"
          ],
          [
            "10b",
            "11"
          ],
          [
            "4b",
            "10b"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "AUA",
          "AUG"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              1
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Phe",
        "sulfur": false,
        "retained": [
          "10b",
          "11",
          "12",
          "12a",
          "4a",
          "4b",
          "5"
```

```
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4",
            "4a"
          ],
          [
            "12a",
            "1"
          ],
          [
            "10a",
            "10b"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "UUC",
          "UUU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              5
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Pro",
        "sulfur": false,
        "retained": [
          "10b",
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "10b",
            "10a"
          ],
          [
            "10b",
            "11"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "CCA",
          "CCC",
          "CCG",
          "CCU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
```

```
            [
              0,
              1
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Ser",
        "sulfur": false,
        "retained": [
          "5"
        ],
        "severed": [],
        "removed_entire": [
          "6",
          "4b"
        ],
        "selected_codons": [
          "AGC",
          "AGU",
          "UCA",
          "UCC",
          "UCG",
          "UCU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            null
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Thr",
        "sulfur": false,
        "retained": [
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "4b",
            "10b"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "ACA",
          "ACC",
          "ACG",
          "ACU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              0
            ]
          ]
```

```
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Trp",
        "sulfur": false,
        "retained": [
          "1",
          "10b",
          "12a",
          "2",
          "3",
          "4",
          "4a",
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "10b",
            "10a"
          ],
          [
            "10b",
            "11"
          ],
          [
            "12a",
            "12"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "UGA",
          "UGG"
        ],
        "literal_retention_fit": false,
        "canonical_retention_parameters": [],
        "cut_boundary_complete": true,
        "construction_status": "explicit fused-ring constructor required"
      },
      {
        "target": "Tyr",
        "sulfur": false,
        "retained": [
          "10b",
          "11",
          "12",
          "12a",
          "4a",
          "4b",
          "5"
        ],
        "severed": [
          [
            "5",
            "6"
          ],
          [
            "4",
            "4a"
          ],
          [
            "12a",
            "1"
          ],
          [
            "10a",
```

```
            "10b"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "UGC",
          "UGU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            0,
            [
              0,
              5
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      },
      {
        "target": "Val",
        "sulfur": false,
        "retained": [
          "4b",
          "5",
          "6"
        ],
        "severed": [
          [
            "6",
            "6a"
          ],
          [
            "4b",
            "4a"
          ],
          [
            "4b",
            "10b"
          ]
        ],
        "removed_entire": [],
        "selected_codons": [
          "GUA",
          "GUC",
          "GUG",
          "GUU"
        ],
        "literal_retention_fit": true,
        "canonical_retention_parameters": [
          [
            1,
            [
              0,
              0
            ]
          ]
        ],
        "cut_boundary_complete": true,
        "construction_status": "ordinary retained-set constructor; decoration/pathway certificate still required"
      }
    ],
    "source": "Article57 active Appendix A entries, with Article59 boundary-cut Cys variant selected in the new construction."
  },
  "data/ncbi_reference_tables.json": {
    "source_url": "https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi",
    "page_update": "2024-09-23",
    "accessed": "2026-10-05",
    "normalization": "Replace T by U; AAs rows are internal assignments, not Starts rows",
    "base_order": "UCAG",
    "tables": {
      "1": {
```

```
        "name": "standard",
        "aas": "FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG"
      },
      "2": {
        "name": "mitochondrial",
        "aas": "FFLLSSSSYY**CCWWLLLLPPPPHHQQRRRRIIMMTTTTNNKKSS**VVVVAAAADDEEGGGG"
      }
    }
  }
}'''')

JUNCTION_METADATA = json.loads(r'''{
  "identity_projections": {
    "Ala": {
      "atoms": {
        "5": "C",
        "Ca": "C",
        "Cc": "C",
        "Na": "N",
        "On0": "O",
        "Ox": "O"
      },
      "bonds": [
        [
          "5",
          "Ca",
          1
        ],
        [
          "Ca",
          "Cc",
          1
        ],
        [
          "Ca",
          "Na",
          1
        ],
        [
          "Cc",
          "On0",
          1
        ],
        [
          "Cc",
          "Ox",
          2
        ]
      ],
      "certificate_id": "chem:Ala",
      "identity_map": {
        "5": "Cb",
        "Ca": "Ca",
        "Cc": "Cc",
        "Na": "N",
        "On0": "Oa",
        "Ox": "Ob"
      },
      "product_formula": "C3H7NO2",
      "stereo_proved": false
    },
    "Cys": {
      "atoms": {
        "5": "C",
        "Ca": "C",
        "Cc": "C",
        "Na": "N",
        "On0": "O",
        "Ox": "O",
        "S1": "S"
      },
      "bonds": [
        [
```

```
          "5",
          "Ca",
          1
        ],
        [
          "5",
          "S1",
          1
        ],
        [
          "Ca",
          "Cc",
          1
        ],
        [
          "Ca",
          "Na",
          1
        ],
        [
          "Cc",
          "On0",
          1
        ],
        [
          "Cc",
          "Ox",
          2
        ]
      ],
      "certificate_id": "chem:Cys",
      "identity_map": {
        "5": "Cb",
        "Ca": "Ca",
        "Cc": "Cc",
        "Na": "N",
        "On0": "Oa",
        "Ox": "Ob",
        "S1": "S"
      },
      "product_formula": "C3H7NO2S",
      "stereo_proved": false
    },
    "Met": {
      "atoms": {
        "10b": "C",
        "4b": "C",
        "5": "C",
        "Ca": "C",
        "Cc": "C",
        "Na": "N",
        "On0": "O",
        "Ox": "O",
        "S1": "S"
      },
      "bonds": [
        [
          "10b",
          "S1",
          1
        ],
        [
          "4b",
          "5",
          1
        ],
        [
          "4b",
          "S1",
          1
        ],
        [
          "5",
          "Ca",
          1
```

```
        ],
        [
          "Ca",
          "Cc",
          1
        ],
        [
          "Ca",
          "Na",
          1
        ],
        [
          "Cc",
          "On0",
          1
        ],
        [
          "Cc",
          "Ox",
          2
        ]
      ],
      "certificate_id": "chem:Met",
      "identity_map": {
        "10b": "Ce",
        "4b": "Cg",
        "5": "Cb",
        "Ca": "Ca",
        "Cc": "Cc",
        "Na": "N",
        "On0": "Oa",
        "Ox": "Ob",
        "S1": "S"
      },
      "product_formula": "C5H11NO2S",
      "stereo_proved": false
    }
  },
  "scope": "Constitutional identity projections of new formal product certificates; no chemical or stereochemical lift."
}''')

EXPECTED_SHA256 = {'certificates/junction/junction_checks.json': '043de4388dac7d06dcba1f8371296d88fc5e35408b07942005cefb24e2fb5bbd',
 'certificates/junction/junction_data.json': 'e191bc1c1d7d31ba47e084b91803a126b3bce198567d675388145630cc59b3a5',
 'certificates/junction/verify_junction.py': 'f08fefcce90642e8b799b191718b5af46bfb49d80df8ce0ac3e78beb47c225a9',
 'certificates/products/build_product_certificates.py': '8224b1d5d996cb46e4ff177b60007f814af9fb1b7c8508e27c0a229bc9c1c869',
 'certificates/products/product_certificates.json': 'e014117195b2ec4444fcc73c9e6c62e14b34bdd238e50904beef87f37da0c903',
 'certificates/products/product_verification_summary.json': '3ec98b33d7a5880ea08c798cd33e4c2df59aecfd6dee4ff6da964db6be0751f1',
 'certificates/products/source_retention_rows.json': '443b34fc6c3e1cdded03bc6e3d3615d5cd158986e6660b554a0e8a5297ef0851',
 'certificates/reachability/reachability_checks.json': 'a3a9f3a562e8164f41a116f570ec2f496d32baf418cb0671820390c27b7757b6',
 'certificates/reachability/verify_reachability.py': 'f4481dfb523ed78f1b23f8e9226fde5563140c25466d426ebf87a8fdc753f123',
 'certificates/registry/marker_incidence.csv': '79f8c9ab7f1ab209215dfb7fdc350aa2edb9808aa13257f0f6d0e9726ad73b08',
 'certificates/registry/orbit_incidence.csv': '5ebb709f21f899c9d445e9d6917ed7725f32f916c9642b4b6c4e845a72ce031d',
 'certificates/registry/registry_data.json': 'bdf39691239ce43f9414200ce1529cdedc8b50d2532f6ffcb1f717de36063c3d',
 'certificates/registry/verify_registry.py': '70eeaba30e281fd5822eb24162e7c6d7b60d76e84d3f35ae7e34e425f2b872a8',
 'data/ncbi_reference_tables.json': 'af847d91e15371ecaf59d2c1744a371313be734676238b8757c6bfe91a6f0d8d',
 'data/product_guard_audit.json': 'e11a8807f2cab378671684ec38e8bf1fca6d3120f5f0db39953eac51e096081c',
 'scripts/validate_product_guards.py': '3c1204a16792161c1e3de83204ba886e430e5b13fd50ba1e797693137efb629a',
 'verify_manuscript.py': '86f82568100f0050b3dd9737165fc573438f7b25a1eb3021905de218c5a40ad0'}

# The following wrapper and cross-checks are new supplement code.
COUNTS = Counter()
NEGATIVE = []
MODULE_COUNTS = {}

def check(condition, label):
    COUNTS['additional_predicate_evaluations'] += 1
```

```
    if not condition:
        raise AssertionError(label)

def rejection(label, action, exceptions=(AssertionError, ValueError)):
    try:
        action()
    except exceptions:
        NEGATIVE.append(label)
        return
    raise AssertionError('Invalid example accepted: '+label)

class KeepAssertions(ast.NodeTransformer):
    """Preserve normal assert evaluation and lazy message evaluation under -O."""
    def visit_Assert(self, node):
        test=ast.Call(func=ast.Name(id='__record_assert__',ctx=ast.Load()),
                      args=[node.test,ast.Constant(node.lineno)],keywords=[])
        error=ast.Call(func=ast.Name(id='AssertionError',ctx=ast.Load()),
                       args=[] if node.msg is None else [node.msg],keywords=[])
        replacement=ast.If(test=ast.UnaryOp(op=ast.Not(),operand=test),
                           body=[ast.Raise(exc=error,cause=None)],orelse=[])
        return ast.copy_location(replacement,node)
    def visit_FunctionDef(self,node):
        self.generic_visit(node)
        if node.name=='require':
            arg=node.args.args[0].arg
            record=ast.Expr(ast.Call(func=ast.Name(id='__record_require__',ctx=ast.Load()),
                                     args=[ast.Name(id=arg,ctx=ast.Load())],keywords=[]))
            node.body.insert(0,ast.copy_location(record,node))
        return node
```

```
def run_source(root, relative, argv=()):
    counts=Counter()
    def record(value,line):
        counts['assert_evaluations']+=1
        truth=bool(value)
        counts['assert_false']+=not truth
        return truth
    def record_require(value):
        counts['require_evaluations']+=1
        counts['require_false']+=not bool(value)
    source=SOURCES[relative]
    tree=KeepAssertions().visit(ast.parse(source,filename=relative))
    tree=ast.fix_missing_locations(tree)
    check(not any(isinstance(n,ast.Assert) for n in ast.walk(tree)),
          'No removable assert remains in executed AST')
    module=types.ModuleType('__main__')
    module.__dict__.update(__file__=str(root/relative),
                           __record_assert__=record,__record_require__=record_require)
    previous=sys.modules['__main__']; old_argv=sys.argv
    capture=io.StringIO()
    try:
        sys.modules['__main__']=module
        sys.argv=[str(root/relative),*argv]
        with redirect_stdout(capture):
            exec(compile(tree,relative,'exec',optimize=0),module.__dict__)
    finally:
        sys.modules['__main__']=previous;sys.argv=old_argv
    MODULE_COUNTS[relative]=counts
    return module.__dict__,capture.getvalue()

def load(root,relative): return json.loads((root/relative).read_text())
def digest(path): return hashlib.sha256(path.read_bytes()).hexdigest()
def write_json(path,value,sort_keys=False):
    path.parent.mkdir(parents=True,exist_ok=True)
    path.write_text(json.dumps(value,indent=2,sort_keys=sort_keys)+'\n')

# Independently written element-labelled reference graph specifications.
# The fixed root names a,c,n,o,h mean alpha C, carboxyl C, amino N,
# carbonyl O, and hydroxy O. Lowercase side-chain labels mean carbon;
# N/O/S-prefixed labels identify the indicated element. These target
# connectivities are declarative input. They do not prove natural synthesis.
TARGET_EDGES = {
 'Gly':'', 'Ala':'a-b', 'Ser':'a-b b-O1', 'Cys':'a-b b-S1',
 'Asp':'a-b b-g g=O1 g-O2', 'Asn':'a-b b-g g=O1 g-N1',
```

```
 'Thr':'a-b b-g b-O1', 'Glu':'a-b b-g g-d d=O1 d-O2',
 'Gln':'a-b b-g g-d d=O1 d-N1', 'Pro':'a-b b-g g-d d-n',
 'Met':'a-b b-g g-S1 S1-d', 'Lys':'a-b b-g g-d d-e e-N1',
 'Arg':'a-b b-g g-d d-N1 N1-e e=N2 e-N3',
 'His':'a-b b-g g=d d-N1 N1-e e=N2 N2-g',
 'Leu':'a-b b-g g-d g-e', 'Val':'a-b b-g b-d',
 'Ile':'a-b b-g g-d b-e',
 'Phe':'a-b b-g g=d d-e e=f f-h1 h1=i i-g',
 'Tyr':'a-b b-g g=d d-e e=f f-h1 h1=i i-g f-O1',
 'Trp':'a-b b-g g=d d-N1 N1-e e=f f-g e-h1 h1=i i-j j=k k-f',
}
CAP={'C':4,'N':3,'O':2,'S':2,'H':1,'*':1}

def key(a,b):return tuple(sorted((a,b)))
def graph_target(name):
    atoms={'a':'C','c':'C','n':'N','o':'O','h':'O'}
    bonds={key('a','c'):1,key('a','n'):1,key('c','o'):2,key('c','h'):1}
    for token in TARGET_EDGES[name].split():
        separator='=' if '=' in token else '-'
        a,b=token.split(separator)
        for v in (a,b):
            if v not in atoms:atoms[v]=v[0] if v[0] in 'NOS' else 'C'
        bonds[key(a,b)]=2 if separator=='=' else 1
    return atoms,bonds

def graph_isomorphic(a,ab,b,bb):
    """Exact backtracking isomorphism, preserving five distinguished roots."""
    if len(a)!=len(b) or len(ab)!=len(bb):return False
    if Counter(a.values())!=Counter(b.values()):return False
    def signature(v,ats,bonds):
        return (ats[v],tuple(sorted((bo,ats[y if x==v else x])
                    for (x,y),bo in bonds.items() if v in (x,y))))
    candidates={v:[w for w in b if signature(v,a,ab)==signature(w,b,bb)] for v in a}
    mapping={'Ca':'a','Cc':'c','Na':'n','Ox':'o','On0':'h'}
    if any(x not in a or y not in candidates[x] for x,y in mapping.items()):return False
    if any(ab.get(key(x,z),0)!=bb.get(key(y,t),0)
           for x,y in mapping.items() for z,t in mapping.items() if x!=z):return False
    def search():
        if len(mapping)==len(a):return True
        unused=set(b)-set(mapping.values())
        available={v:[w for w in candidates[v] if w in unused and
                      all(ab.get(key(v,x),0)==bb.get(key(w,y),0) for x,y in mapping.items())]
                   for v in a if v not in mapping}
        v=min(available,key=lambda x:(len(available[x]),x))
        for w in available[v]:
            mapping[v]=w
            if search():return True
            del mapping[v]
        return False
    return search()

def heavy_graph(row):
    all_atoms=row['endpoint']['atoms']
    atoms={v:all_atoms[v] for v in row['product_vertices'] if all_atoms[v]!='H'}
    bonds={key(a,b):n for a,b,n in row['endpoint']['bonds'] if a in atoms and b in atoms}
    return atoms,bonds

def verify_products(data,guard):
    rows=data['rows'];identity={};primitive_sum=0
    for name,row in sorted(rows.items()):
        a,ab=heavy_graph(row)
        matches=[]
        for target in sorted(TARGET_EDGES):
            b,bb=graph_target(target)
            if graph_isomorphic(a,ab,b,bb):matches.append(target)
        check(matches==[name],name+' has exactly its declared rooted constitution')
        identity[name]=matches[0]
        initial=row['initial'];terminal=row['endpoint']
        check(initial['atoms']==terminal['atoms'],name+' conserves every atom identity')
        check(set(row['persistent_origin'])==set(initial['atoms']),name+' has complete origins')
        check(row['step_count']==len(row['rewrite_steps']),name+' primitive count')
        primitive_sum+=len(row['rewrite_steps'])
        # Independently check no atom is missing from terminal components.
        bonds={key(u,v):n for u,v,n in terminal['bonds']}
        remaining=set(terminal['atoms']);components=[]
```

```
        while remaining:
            start=min(remaining);seen={start};todo=[start]
            while todo:
                v=todo.pop()
                for u,w in bonds:
                    z=w if u==v else u if w==v else None
                    if z is not None and z not in seen:seen.add(z);todo.append(z)
            remaining-=seen;components.append(seen)
        check(sum(map(len,components))==len(terminal['atoms']),name+' full material partition')
        check(next(c for c in components if 'Ca' in c)==set(row['product_vertices']),name+' full product component')
    base=copy.deepcopy(rows['Ala'])
    S,W={'5'},{'6','4b'}
    mutants={}
    m=copy.deepcopy(base);m['endpoint']['atoms'].pop(next(x for x in m['endpoint']['atoms'] if x.startswith('h2:')));mutants['lost_atom']=m
    m=copy.deepcopy(base);m['endpoint']['atoms']['5']='N';mutants['changed_element']=m
    m=copy.deepcopy(base);m['endpoint']['bonds'].append(m['endpoint']['bonds'][0]);mutants['duplicate_bond_record']=m
    m=copy.deepcopy(base);m['rewrite_steps'].insert(0,['erase','5','Ca']);mutants['unknown_primitive']=m
    m=copy.deepcopy(base);m['rewrite_steps'].append(['join','Ca','Ca']);mutants['self_loop_primitive']=m
    m=copy.deepcopy(base);m['rewrite_steps'].pop();mutants['missing_final_primitive']=m
    for label,mutant in mutants.items():
        rejection(label,lambda m=mutant:guard['replay'](m,{'':(S,W)}))
    a,ab=heavy_graph(rows['Ile']);b,bb=graph_target('Leu')
    check(not graph_isomorphic(a,ab,b,bb),'Same-formula Ile/Leu are not interchangeable')
    # This full identity check is not merely the original formula comparison.
    return {'rooted_identity_comparisons':400,'unique_identity_matches':20,
            'individual_path_primitives':primitive_sum,'identities':identity}

def hall_exhaustion(reg):
    """All subsets: dynamic union of neighbor bitmasks, not a matcher call."""
    alphabet=sorted(reg['amino_acids']);position={a:i for i,a in enumerate(alphabet)}
    report={}
    for name in ('WC','RY','MK'):
        masks=[sum(1<<position[a] for a in r['neighbors']) for r in reg['graphs'][name]['rows']]
        unions=[0]*(1<<20);maximum=0;witness=0
        for subset in range(1,1<<20):
            low=subset & -subset
            unions[subset]=unions[subset^low] | masks[low.bit_length()-1]
            deficiency=subset.bit_count()-unions[subset].bit_count()
            if deficiency>maximum:maximum=deficiency;witness=subset
        matching=reg['graphs'][name]['displayed_matching']
        check(len(matching)==20-maximum,'Attaining matching plus exhaustive Hall bound: '+name)
        check(len(set(matching.values()))==len(matching),'Matching has distinct products')
        for q,a in matching.items():check(a in reg['graphs'][name]['rows'][int(q)-1]['neighbors'],'Matching edge exists')
        report[name]={'subsets_examined':1<<20,'maximum_deficiency':maximum,
                      'matching_size':len(matching),'one_max_deficiency_subset':
                      [q+1 for q in range(20) if witness>>q&1]}
    return report
```

```
def verify_composition(data):
    # Eighteen non-sulfur unary modules plus one coupled two-carrier module.
    modules=[(name,data['rows'][name]) for name in sorted(data['rows']) if name not in ('Cys','Met')]
    modules.append(('sulfur_pair',data['paired_sulfur']))
    atoms={};bonds={};expected={};degrees={};streams=[]
    for name,row in modules:
        prefix=name+'|'
        renamed={prefix+a:e for a,e in row['initial']['atoms'].items()}
        check(not (atoms.keys() & renamed.keys()),'Disjoint module atom prefixes')
        atoms.update(renamed)
        bonds.update({key(prefix+a,prefix+b):n for a,b,n in row['initial']['bonds']})
        expected.update({key(prefix+a,prefix+b):n for a,b,n in row['endpoint']['bonds']})
        streams.append([(op,prefix+a,prefix+b) for op,a,b in row['rewrite_steps']])
    degrees={a:0 for a in atoms}
    for (a,b),n in bonds.items():degrees[a]+=n;degrees[b]+=n
    inventory=Counter(e for e in atoms.values() if e!='*')
    check(inventory==Counter(C=400,H=882,N=80,O=140,S=2),'Exact repertoire resource sum')
    check(sum(inventory.values())==1504 and list(atoms.values()).count('*')==40,'Actual atoms and ports stay distinct')
    executed=0
    for index in range(max(map(len,streams))):
        for stream in streams:
```

```
            if index>=len(stream):continue
            op,a,b=stream[index];e=key(a,b);delta=-1 if op=='split' else 1
            check(op in ('split','join') and 0<=bonds.get(e,0)+delta<=3,'Interleaved primitive bond guard')
            degrees[a]+=delta;degrees[b]+=delta
            check(0<=degrees[a]<=CAP[atoms[a]] and 0<=degrees[b]<=CAP[atoms[b]],'Interleaved valence guard')
            bonds[e]=bonds.get(e,0)+delta
            if not bonds[e]:del bonds[e]
            executed+=1
    check(bonds==expected,'Interleaving reaches disjoint union of all nineteen endpoints')
    check(all(degrees[a]==CAP[e] for a,e in atoms.items()),'All composite terminal valences close')
    return {'modules':19,'active_carriers':20,'actual_atoms':1504,'ports':40,
            'element_inventory':dict(sorted(inventory.items())),
            'round_robin_primitives':executed,'all_terminal_bonds_match':True}

def verify_controller(junction):
    State=junction['State'];transition=junction['transition']
    traces=segments=transitions=0
    for n,stage,elong in product(range(1,21),('standard','mitochondrial'),(False,True)):
        state=State(remaining=tuple(range(n)),stage=stage)
        for packet in range(n):
            tape=('AUG',)+(('GCU',) if elong else ())+('UAA',)
            state=transition(state,'RESET',packet=packet,a='C',frame=packet%4,tape=tape)
            transitions+=1
            rules=('FORM','GATE','START')+(('STEP',) if elong else ())+('OBSERVE','RELEASE','HALT')
            for rule in rules:state=transition(state,rule);transitions+=1
        word=('Met','Ala') if elong else ('Met',)
        check(state.done==(word,)*n,'Every tested reset segment is Met-first')
        check(not state.remaining and state.used==tuple(range(n)),'Exactly n packets, no next unused preload')
        check(len(state.consumed)==n*(2 if elong else 1),'Each emitted product has one unique token')
        rejection('reuse_packet_n='+str(n)+'_'+stage+'_'+str(elong),lambda:transition(state,'RESET',packet=0))
        traces+=1;segments+=n
    # Mandatory certificate flags are model predicates, not physical observations.
    s=State(remaining=(0,));s=transition(s,'RESET',packet=0);s=transition(s,'FORM');s=transition(s,'GATE')
    rejection('wrong_constitution_at_start',lambda:transition(junction['replace'](s,outid='Ala'),'START'))
    return {'additional_complete_traces':traces,'successful_segments':segments,
            'transition_executions':transitions,'packet_range':[1,20],
            'endpoint_stages':['standard','mitochondrial'],'tapes':['AUG UAA','AUG GCU UAA']}

def verify_interfaces(root,products,reg,reach,junction_data,junction_report):
    names=dict(zip('Ala Cys Asp Glu Phe Gly His Ile Lys Leu Met Asn Pro Gln Arg Ser Thr Val Trp Tyr'.split(),'
     ACDEFGHIKLMNPQRSTVWY'))
    check(set(products['rows'])==set(names),'Exactly the twenty declared product names')
    check(set(reg['amino_acids'])==set(names.values()),'Shared residue alphabet')
    check(reg['standard_code']==reach['standard'],'Registry and routing standard table agree')
    reference=load(root,'data/ncbi_reference_tables.json')
    words=[''.join(c) for c in product(reference['base_order'],repeat=3)]
    for row in reference['tables'].values():
        check(len(row['aas'])==64,'64 assignments in fixed reference snapshot')
        check(dict(zip(words,row['aas']))==reach[row['name']],'All entries equal fixed NCBI snapshot')
    for name,projection in junction_data['identity_projections'].items():
        a,b=heavy_graph(products['rows'][name])
        check(a==projection['atoms'],'Junction/product atom identity: '+name)
        check(b=={key(u,v):n for u,v,n in projection['bonds']},'Junction/product bond identity: '+name)
        check(products['rows'][name]['product_formula']==projection['product_formula'],'Junction/product formula')

    check(junction_data['paired_sulfur']==products['paired_sulfur'],'Identical full paired sulfur path, not just
     endpoint')
    check(junction_report['no_packet_reuse'] and junction_report['no_emission_before_start'],'Controller report
     invariant flags')
    differences=sorted(c for c in words if reach['standard'][c]!=reach['mitochondrial'][c])
    check(differences==['AGA','AGG','AUA','UGA'],'Exactly four endpoint assignment differences')
    return {'tables_compared':2,'assignments_per_table':64,'endpoint_differences':differences,
            'identity_projections_compared':3,'paired_path_exact_equality':True,
            'reference_scope':'Bundled NCBI snapshot, not a live database query'}

def verify_assertion_transform():
    events=[]
    def record(value,line):
        events.append(('predicate',bool(value)))
        return bool(value)
    def message():
        events.append(('message',True))
        return 'deliberate failure'
    source='assert True, message()\nassert False, message()\n'
    tree=ast.fix_missing_locations(KeepAssertions().visit(ast.parse(source)))
```

```
    rejection('optimized_assert_transform',lambda:exec(compile(tree,'transform-selftest','exec',optimize=2),
             {'__record_assert__':record,'message':message}))
    check(events==[('predicate',True),('predicate',False),('message',True)],
          'Assertion transform preserves predicate tests and lazy failure messages')


def main():
    verify_assertion_transform()
    parser=argparse.ArgumentParser(description=__doc__)
    parser.add_argument('--output',type=Path,default=Path('results_article75.json'))
    args=parser.parse_args()
    with tempfile.TemporaryDirectory(prefix='article75-verification-') as directory:
        root=Path(directory)
        for name,source in SOURCES.items():
            p=root/name;p.parent.mkdir(parents=True,exist_ok=True);p.write_text(source)
        for name,data in FIXTURES.items():write_json(root/name,data)
        for name in SOURCES:
            check(digest(root/name)==EXPECTED_SHA256[name],'Bundled source bytes agree with controlling original: '+name)
        for name in FIXTURES:
            check(digest(root/name)==EXPECTED_SHA256[name],'Fixture serialization agrees with controlling original: '+name)
        registry,_=run_source(root,'certificates/registry/verify_registry.py',('--write',))
        product_ns,_=run_source(root,'certificates/products/build_product_certificates.py')
        products=load(root,'certificates/products/product_certificates.json')
        junction_data=dict(JUNCTION_METADATA)
        junction_data['paired_sulfur']=products['paired_sulfur']
        write_json(root/'certificates/junction/junction_data.json',junction_data,sort_keys=True)
        routing,_=run_source(root,'certificates/reachability/verify_reachability.py')
        junction,_=run_source(root,'certificates/junction/verify_junction.py')
        guards,_=run_source(root,'scripts/validate_product_guards.py')
        # The supplied combined entrypoint runs its own child programs normally.
        # Its require predicates are instrumented; child assertion counts are not.
        saved_optimization=os.environ.get('PYTHONOPTIMIZE')
        os.environ['PYTHONOPTIMIZE']='0'
        try:
            combined,_=run_source(root,'verify_manuscript.py')
        finally:
            if saved_optimization is None:os.environ.pop('PYTHONOPTIMIZE',None)
            else:os.environ['PYTHONOPTIMIZE']=saved_optimization
        reg=load(root,'certificates/registry/registry_data.json')
        reach=load(root,'certificates/reachability/reachability_checks.json')
        jreport=load(root,'certificates/junction/junction_checks.json')
        hash_checks={}
        for name,expected in EXPECTED_SHA256.items():
            observed=digest(root/name)
            check(observed==expected,'Regenerated artifact SHA-256 matches controlling original: '+name)
            hash_checks[name]=observed
        interfaces=verify_interfaces(root,products,reg,reach,junction_data,jreport)
        identity=verify_products(products,guards)
        hall=hall_exhaustion(reg)
        composition=verify_composition(products)
        controller=verify_controller(junction)
        audit=load(root,'data/product_guard_audit.json')
        check(all(audit['rejection_tests'].values()),'All original seven extraction-guard controls reject')
        check(reach['control_breakpoints']==['0','1/2','1'],'Complete affine zero/equality breakpoint set')
        check(reach['distinct_routing_assignments_at_u0']==19508428800,'Exact locked routing freedom')
        report={
          'status':'PASS','scope':'Finite formal graph, registry, routing and controller verification',
          'runner_version':1,'minimum_python':'3.10','standard_library_only':True,
          'assertion_transformation':'Explicit predicate evaluation and AssertionError; original assert messages remain lazy',
          'instrumented_source_predicates':{k:dict(sorted(v.items())) for k,v in sorted(MODULE_COUNTS.items())},
          'additional_predicate_evaluations':COUNTS['additional_predicate_evaluations'],
          'additional_negative_controls':NEGATIVE,
          'original_guard_negative_controls':audit['rejection_tests'],
          'registry':{'classes':len(reg['classes']),'addressed':64,'reference_only':36,'common_refinement':reg['minimal_refinement']['cell_count'],'all_letter_bijections':len(reg['all24']),'orbit_enumeration_N':[1,6]},
          'exhaustive_hall_subsets':hall,'product_identities':identity,'resource_composition':composition,
          'routing':{'seed_demands':reach['seed_demand_count'],'seed_cells':reach['seed_cell_count'],
                     'matching_count':reach['matching_count'],'spare_triples':reach['spare_triples'],
                     'breakpoints':reach['control_breakpoints'],'distinct_routing_assignments_at_u0':reach['distinct_routing_assignments_at_u0']},
          'junction':{k:jreport[k] for k in ('marker_frame_conjugacy_checks','hexamers_checked','controller_case_count','accepted_cases','paired_graph_rewrite_steps','paired_graph_atom_count','
```

```
     paired_bridge_stage_steps')},
          'additional_controller':controller,'shared_interfaces':interfaces,
          'verified_artifact_sha256':hash_checks,
          'not_certified':['precursor/carrier atom map','chemical feasibility or rates','stereoselectivity','full
      external scaffold grafting','physical decoding, release, start, halt, reset','target-independent or unique
     code selection','all infinite or continuous claims in the manuscript','independent live NCBI verification'],
        }
    write_json(args.output,report,sort_keys=True)
    print('PASS: all original modules and shared interfaces reproduced exactly.')
    print('PASS: 400 rooted graph comparisons; 3 x 1,048,576 Hall subsets.')
    print('PASS: 19-module, 1,504-atom repertoire interleaving; 840 reset segments.')
    print('Additional explicit negative controls rejected:',len(NEGATIVE))
    print('Wrote',args.output)

if __name__=='__main__':main()
```