Standalone Verification Code

Reproducibility and Scope Dossier

Article 77: Carbon traceable protocell construction
from substituted chrysene stocks

Charles D. Schaper, Ph.D.

Annals of the Chrysene Formalism
Volume 1, Number 4 · Companion manuscript opening page 1567
25 October 2026

Executed result: PASS

Runtime	Python standard library; no third-party packages
Fixed witness data	Four bundled files with pinned SHA-256 digests
Original finite audits	Five stock graphs; 48 pore graphs; 512 Hall cases
Prepared products	20 diagnostic paths, 668 edits; paired sulfur path, 174 edits
Simultaneous repertoire	19 instances, 1,504 actual atoms, 40 ports, 107 product carbons
Supplemental exact cases	3,888 origin transfers; 2,817 mask tests; 36 taut daughter geometries
Ordered count paths	Nine paths, 855 microsteps, 570 completed timestamps
Negative controls	21 supplied controls plus 19 additional controls
Numerical consistency	Four fresh RK4 runs; 802 frozen trajectory rows compared
Optimization / portability	Normal, `-O`, environment `-OO`, relocation, and listing reconstruction agree

Precisely limited certification

A pass verifies *finite formal witnesses and rational arithmetic under supplied model assumptions.* It does not certify every theorem of the manuscript, analytic ODE existence by simulation, chemical feasibility, a stock-to-prepared molecular path, measured peptide function, physical division, or a functioning protocell. Numerical trajectories and conditional division tests have separate, explicitly limited evidence levels.

1 Reproduction: a standalone document, a data-bearing executable package

This `.tex` file can be uploaded *alone* to a blank Overleaf project and compiled with pdfLaTeX. It embeds all five executable source listings and uses no external figures, `\input` files or listing files. PDF compilation does not run Python or Lean; shell escape is unnecessary.

Actual Python execution requires the accompanying source ZIP. Its `data/` directory contains the original catalogue, reservation fixture, product graph certificates and fixed trajectory CSV. The product JSON alone is approximately 544 KiB. These data are not silently recreated or claimed to be derivable from a carbon total. The single-file LaTeX document is therefore *compilation-independent*, not a data-independent runnable certificate.

```
python3 verify_article77.py
python3 -O verify_article77.py --output-dir results_optimized
PYTHONOPTIMIZE=2 python3 verify_article77.py --output-dir results_environment
```

Use Python 3.10 or later. The last command uses POSIX environment syntax; `python3 -OO` is the equivalent optimization mode. The selected output directory receives the aggregate JSON, runtime metadata, four component reports, fresh trajectories, and subprocess logs. Failed predicates raise exceptions and give a nonzero exit. Existing output files with the same names are replaced; the source and fixed inputs are read-only.

1.1 Optimization-safe execution

The supplied identity and dynamics programs contained seven and thirty-four Python `assert` statements, respectively. Those statements disappear under optimization. The bundled adapters expand all forty-one to explicit `if not ...: raise AssertionError(...)` branches. The original material and guard programs already use explicit checks. Only output routing and formatting are otherwise adapted; the original source bytes are retained under `provenance/`. The driver rejects any remaining `assert` node in its executed source set and propagates its optimization mode to every child process.

All five sources were reconstructed from this document's listings, compared byte-for-byte with the distributed executables, and executed on a relocated directory containing only those sources and the four data files. Normal, optimized, environment-optimized and relocated runs produced identical semantic JSON. An altered fixed-data file was rejected with the expected pinned-hash error in normal and optimized execution. These are actual test results, not archive-supplied success flags.

1.2 Source identity and supplied assumptions

The controlling source is the user's supplied Article 77 archive and its revised Article 75 product data. `input_manifest.json` records source-file byte hashes and the four original-to-adapted checker correspondences. It does not claim that publication metadata or layout adjustments preserve the original manuscript's hash. Original archive-relative paths are audit locators, not run-time dependencies on absent archives.

The driver embeds hashes for the four data files and four adapted auditors. Hashes detect substitution or accidental alteration; they are not digital signatures or proofs that a molecule or constitutive law exists physically. The expected formula tables, graph restrictions, rate constants, stoichiometric yields and physical-interface premises are *inputs to verification*, not discovered facts.

2 What is checked, and at what logical level

Layer	Recomputed result	Limit of the result
Stock and provenance	Five full neutral stock graphs; membrane precursors $C_{40}H_{31}O_8P$ and $C_{44}H_{39}O_{10}P$; terminal-complete pore $C_{162}H_{96}O_{75}P_{18}$; 810 distinct reserved carbon origins.	Constitution and disjoint reservations, not assembly chemistry or a sufficient cell inventory.
Origin transitions	Internal transfers, exports, old-origin returns and external-reserve entry; 3,888 exhaustive transitions of three fixed labels among six locations; 100 repeated export/return cycles.	Finite model tests support, but do not replace, the universal induction proof of provenance.

Layer	Recomputed result	Limit of the result
Allocation	All 512 three-demand/three-slot eligibility graphs agree with Hall's criterion; multi-resource collision and catalyst-gated stoichiometric counterexamples.	No chemical reachability follows from carbon sufficiency or a nonnegative net endpoint.
Prepared products	Exact replay of every primitive, intermediate capacities, source-specific extraction guards, full terminal components and target formulas. Independent weighted heavy-atom identities for 20 unary targets and two paired endpoints.	Prepared-input graph grammar only; no stock preparation, filled resident context, stereochemistry, yield or reaction path.
Joint inventory	Literal disjoint union of 18 nonsulfur unary endpoints and one two-copy sulfur endpoint: 1,504 actual atoms, 40 ports, 400 exposed carbons, 107 product carbons and 293 nonproduct carbons.	The 20 diagnostic unary paths plus the pair are not 21 simultaneous preparations. Full carrier contexts and spectator cores are additional.
Joining	An explicit Gly–Pro–Cys–Met graph word performs 12 unit edits, conserves all atoms and carbon, and yields $C_{15}H_{26}N_4O_5S_2$ plus three mapped waters.	Formal peptide connectivity, not spontaneous condensation or the functional product assigned to the ODE.
Amounts/counts	Three conserved stoichiometric rows; 60 rational projection cases; nine exact prescribed-time count paths, with production before deactivation at ties.	Finite stoichiometric approximation, not a physical stochastic process, kinetic derivation or stock-resolved atom map.
Operating law	Exact rational nominal and robust bounds, feedback sign checks, water boundary arithmetic, energy floors and four exact Donnan examples.	Arithmetic within the declared laws; the all-time analytic argument is not replaced by finite sampling.
Trajectories	Fresh feedback and ablation RK4 runs at two step sizes; exact half-period mode alignment; 802 rows compared with the fixed figure data.	Unvalidated floating-point consistency only, not a rigorous global numerical error enclosure.
Division/return	Exact normalized daughter capacity examples, integer assembly minima, finite renewal/branching budgets and scalar return-slope tests.	Consequences of conditional hypotheses, not an actual neck path, physical partition or return map for the operating witness.

The 810-carbon reservation witness and the ODE's 130 donor units, 30 carrier units and separately reserved scaffold are different constructions. No unprovided map identifying their materials is inferred. The preparation gate in the original material checker tests the presence of named fields, not the chemical validity of their contents. No completed preparation certificate is present, and its absence remains explicit.

3 Exact operating arithmetic versus numerical illustration

For the nominal assumed-law model, the arithmetic audit recomputes

$$\mu = \frac{2850}{3131}, \qquad \delta = \frac{3147}{6262} > 0, \qquad \frac{3/2}{\delta} = \frac{3131}{1049} < 2.985 < 4.$$

The corresponding stronger $p = 8/5$ calculation gives

$$\delta_{8/5} = \frac{87626}{203515} > 0, \qquad \frac{8/5}{\delta_{8/5}} = \frac{162812}{43813} < 3.717 < 4.$$

These exact inequalities are used in the manuscript's concavity, differential-comparison and no-downward-crossing proofs. Merely evaluating them does not formalize Carathéodory existence, uniqueness, compact continuation, or the continuous-time comparison principle.

The nominal bounds are $J \le 25$, $b \ge 30$, $c \ge 10$, $A \le 660$, $100 \le V \le 600$, and $E \ge 40$ on $[0, 4]$. The exact water upper-bracket value is $-5999/360000$. For the robust box, the audit verifies

$$\widehat{\mu} = \frac{717500}{799663}, \quad \widehat{\delta} = \frac{70245111}{159932600} > 0, \quad \frac{3/2}{\widehat{\delta}} = \frac{239898900}{70245111} < 3.416.$$

It also verifies $J < 26$, $A \le 669.688 < 670$, the strict water-boundary signs, an ablation active-amount ceiling below 1.026, and $E \ge 35.6 > 35$. Universal conclusions about all admissible measurable perturbations rely on the written regularity and comparison hypotheses, not on a finite parameter grid.

3.1 Fresh numerical results

The two step sizes are 0.001 and 0.0005. Every step lies within one constant-light mode, with light changes scheduled by an integer index. The fine-grid results are:

Model	First sampled $p \ge 1.5$	$p(4)$	$b(4)$
Feedback	1.6110	2.16439811	49.03043176
Matched ablation	Not attained	0.94265831	91.59136957

The first threshold crossing is a grid observation, not a rigorously localized event time. Step-size agreement and small conserved-ledger residuals can reveal implementation errors but cannot prove an error bound. The matched ablation deletes only the productive feedback term; supplies evolve separately from the same initial values under the remaining laws. No artificial forcing of equal depleted-reserve trajectories is used.

4 Conditional geometry and resource tests

The additional geometry audit does more than test $a + (1 - a) = 1$. For each pair of integer radii $1 \le r_1 \le r_2 \le 8$, put

$$\alpha = \frac{r_1^3}{r_1^3 + r_2^3}, \qquad a = \frac{r_1^2}{r_1^2 + r_2^2}, \qquad v^2 = \frac{(r_1^3 + r_2^3)^2}{(r_1^2 + r_2^2)^3}.$$

The physical sphere constants cancel from this reduced-volume expression. All 36 cases check the branch domain, conserved normalized area and volume, and both exact identities

$$v^2\frac{\alpha^2}{a^3} = 1, \qquad v^2\frac{(1-\alpha)^2}{(1-a)^3} = 1.$$

Three rational slack cases check symmetric-halving daughter capacity. This is an exact finite family, not a numerical root finder or a proof of the universal monotonicity/unique-root proposition.

For indivisible hardware, 84 viable integer allocations with parent count at most eight are checked. One pore cannot give one complete pore to each daughter, and half-pore allocations are rejected. Sixty-four finite renewal examples fund every chosen-lineage cycle and check the $2^N - 1$ binary-parent count. Eighty-one rational slope tests check

$$\left|\frac{1+G'}{2}\right| < 1 \quad \Longleftrightarrow \quad -3 < G' < 1.$$

A negative-slope counterexample rejects the insufficient criterion $G' < 1$ alone. These calculations do not certify the five physical flow–gate–path–partition–return requirements. In particular, the four-period operating witness has no supplied division or renewal reset.

5 Rejection controls and explicit assumptions

The 21 original controls reject representative duplicate admissions, free organic feeds, missing or substituted carbon, double allocations, unfilled port contexts, incorrect formulas, terminal-arm overuse, premature deactivation and illegal extraction edits. The 19 additional controls require the expected rejection reason rather than counting arbitrary exceptions as success:

1. truncated_arm_list: Stock locants/arms differ from declared catalogue.
2. missing_owned_origin: missing or hidden carbon.
3. invented_owned_origin: missing or hidden carbon.
4. unadmitted_transfer: unadmitted carbon movement.
5. noneligible_feed_CO2: carbon-bearing extra feed.
6. noneligible_feed_NaHCO3: carbon-bearing extra feed.
7. noneligible_feed_HCN: carbon-bearing extra feed.
8. noneligible_feed_C2H6O: carbon-bearing extra feed.
9. wrong_target_double_bond: Arg: weighted heavy-atom graph mismatch.
10. duplicate_product_bond: Duplicate bond records.
11. changed_atom_element: Atom identities or elements changed.
12. cross_copy_core_join: New or cross-copy core adjacency.
13. deactivation_annihilates_chain: Residue ledger is not conserved.
14. incorrect_processing_yield: Residue ledger is not conserved.
15. fractional_pore_daughters: Noninteger complete assembly.
16. missing_second_complete_pore: Daughter complete-assembly minimum.
17. double_used_pore: Partition conservation.
18. negative_slope_false_attraction: Not a contraction.
19. zero_transfer_margin: Strict margin required.

The source-model premises remain: the named neutral capacity convention and prepared graph grammar; finite donor packets and separately inventoried structural support; the stipulated joining

yields and retained-product feedback law; the stated switch, Donnan and water closures; the bounded regularity assumptions used in the analytic arguments; and, for every physical transfer or division claim, a single compatible molecular realization and all named path, sorting, resource and return conditions.

No assertion in this dossier supplies an original-stock atom map for the prepared backbone, resident adenosines or omitted contexts. There is no certification of L stereochemistry, tautomer/protonation selection, reaction feasibility, measured peptide activity, a fluid membrane, physical pore throughput, physical scission, hereditary information, an endogenous clock or indefinitely funded operation. The separate Lean supplement has its own exact theorem scope; neither supplement should be described as a complete machine verification of the entire manuscript.

6 Manifest and execution record

Driver SHA-256:

`04f7ca54fa3364762a5eca1b5fc1306cc2f56b2a596d19d29711122ff3ea7d34`

Semantic results SHA-256:

`bd9fec5ea60358a246679006a51483b7fbc637b7c57be6a1b79963f8e131773c`

The complete original-source inventory is supplied in `input_manifest.json`. A separate `published_manuscript_manifest.json` binds all 15 final published TeX source files (main, macros, journal style, references, nine sections and two appendices), without making them executable dependencies. Its SHA-256 is:

`98465cda01f385657fdde71449e4adeafb308ae8ece6e9302d82efafd0e43142`

The fixed runnable inputs are:

File	SHA-256
`auditors/verify_dynamics.py`	`b58e4359bb6c16034ce863058e552760c512caea4721b8b684ce6ea5e9bfada5`
`auditors/verify_materials.py`	`f7353ce4c54bf75c468c443375322351dc7dc2d22492360be7351dcc43631de3`
`auditors/verify_product_guards.py`	`139efbf60840ba82a417ff9932d1a4600ba6014b065dc9a252664a2d979afb78`
`auditors/verify_product_identities.py`	`f7fccf2a417d1a2994a77c37fe2f74b13b4ee1a0dc48dcd983dc99810f522398`
`data/article75_product_certificates.json`	`e014117195b2ec4444fcc73c9e6c62e14b34bdd238e50904beef87f37da0c903`
`data/input_catalog.json`	`9c3197f5d4a57a3172384882dd179804e0ad4523f75cf7c8bf11e334eaa85a95`
`data/material_fixture.json`	`6851a5d79b8f15c0d213138d4da01995e327951d2dc3549507aa85c0ce4523cc`
`data/operating_trajectories.csv`	`501d2aab60c7928678cfc34bb63b6e759dcd41bc17916bee599697fdd6a4524f`

The semantic result digest excludes runtime metadata so that different optimization modes can be compared directly. `runtime.json` records the Python version, current optimization flag and driver digest separately. `run_validation.json` records the executed reconstruction, relocation, mode agreement and rejection tests. The checks use no randomness, downloads or external source archive.

7 Complete executable sources

Save each listing under its displayed relative path, preserving the `auditors/` directory. Obtain the unchanged `data/` files from the supplied verification ZIP. The Python sources below are the same bytes used for the executed checks; the earlier sections explain their limited scope.

verify_article77.py

```python
#!/usr/bin/env python3
"""Article 77: finite exact certificates and separate RK4 illustrations.

Run with Python 3.10+ and its standard library. The four bundled adapters
consume the fixed supplied data; this driver adds independent finite checks.
Neither Python execution nor PDF compilation proves a physical protocell,
analytic ODE existence, a stock-to-prepared path, or actual cell division.
"""
from pathlib import Path
from fractions import Fraction as F
from collections import Counter
from copy import deepcopy
from itertools import product, combinations
import argparse
import ast
import csv
import hashlib
import importlib.util
import json
import math
import os
import subprocess
import sys

ROOT = Path(__file__).resolve().parent
EXPECTED_INPUTS = {
    'data/article75_product_certificates.json': 'e014117195b2ec4444fcc73c9e6c62e14b34bdd238e50904beef87f37da0c903',
    'data/input_catalog.json': '9c3197f5d4a57a3172384882dd179804e0ad4523f75cf7c8bf11e334eaa85a95',
    'data/material_fixture.json': '6851a5d79b8f15c0d213138d4da01995e327951d2dc3549507aa85c0ce4523cc',
    'data/operating_trajectories.csv': '501d2aab60c7928678cfc34bb63b6e759dcd41bc17916bee599697fdd6a4524f',
    'auditors/verify_dynamics.py': 'b58e4359bb6c16034ce863058e552760c512caea4721b8b684ce6ea5e9bfada5',
    'auditors/verify_materials.py': 'f7353ce4c54bf75c468c443375322351dc7dc2d22492360be7351dcc43631de3',
    'auditors/verify_product_guards.py': '139efbf60840ba82a417ff9932d1a4600ba6014b065dc9a252664a2d979afb78',
    'auditors/verify_product_identities.py': 'f7fccf2a417d1a2994a77c37fe2f74b13b4ee1a0dc48dcd983dc99810f522398',
}
NAMES = ('Ala Arg Asn Asp Cys Gln Glu Gly His Ile Leu Lys Met Phe '
         'Pro Ser Thr Trp Tyr Val').split()
NEGATIVES = []


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


def reject(name, action, expected_text):
    try:
        action()
    except (ValueError, AssertionError) as error:
        require(expected_text in str(error),
                name + ': wrong rejection reason: ' + str(error))
        NEGATIVES.append({'name': name, 'reason': str(error)})
        return
    raise ValueError('Invalid case accepted: ' + name)


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


```

```
def load_module(name):
    spec = importlib.util.spec_from_file_location(name, ROOT / 'auditors' / (name + '.py'))
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    return module


def validate_hashes():
    for name, expected in EXPECTED_INPUTS.items():
        require(sha(ROOT / name) == expected, 'Pinned input mismatch: ' + name)
    for path in [Path(__file__), *sorted((ROOT / 'auditors').glob('*.py'))]:
        require(not any(isinstance(n, ast.Assert) for n in ast.walk(ast.parse(path.read_text()))),
                'Optimization-removable assert: ' + path.name)
    return EXPECTED_INPUTS


def baseline_runs(out):
    env = dict(os.environ, A77_RESULTS_DIR=str(out))
    flags = ['-OO'] if sys.flags.optimize >= 2 else ['-O'] if sys.flags.optimize else []
    reports = {}
    for script, report in [('verify_materials', 'material_verification.json'),
                           ('verify_product_guards', 'product_guard_audit.json'),
                           ('verify_product_identities', 'product_identity_checks.json'),
                           ('verify_dynamics', 'dynamics_checks.json')]:
        args = [sys.executable, *flags, str(ROOT / 'auditors' / (script + '.py'))]
        if script == 'verify_dynamics':
            args += ['--output-dir', str(out)]
        p = subprocess.run(args, env=env, capture_output=True, text=True, cwd=ROOT)
        (out / (script + '.stdout.txt')).write_text(p.stdout)
        (out / (script + '.stderr.txt')).write_text(p.stderr)
        require(p.returncode == 0, script + ' failed: ' + p.stderr)
        reports[script] = json.loads((out / report).read_text())
    return reports


def validate_catalog(cat):
    expected = {'D2': ([6, 12], ['e', 'e']),
                'I2': ([3, 6, 12], ['e', 'e', 'e']),
                'T2': ([3, 6, 9, 12], ['e'] * 4),
                'T0': ([3, 6, 9, 12], ['h'] * 4),
                'Dm': ([6, 12], ['h', 'e'])}
    rows = cat['types']
    require(len(rows) == 5 and {r['id'] for r in rows} == set(expected),
            'Catalogue must contain exactly five distinct types')
    for row in rows:
        require((row['sites'], row['arms']) == expected[row['id']],
                'Stock locants/arms differ from declared catalogue')
        require(row['core_C'] == 18 and row['arm_C'] == 2 * row['arms'].count('e'),
                'Stock carbon metadata mismatch')


def material_extensions(m, catalog):
    # Exhaust all assignments of three distinct admitted labels to six locations,
    # then all single-label relocations. This is finite model testing, not induction.
    locations = ['free', 'bound', 'waste', 'recycle', 'outside', 'loss']
    labels = sorted(m.carbon_labels(catalog, {'batch': 'finite', 'parent': 'a', 'stock': 'Dm'}))[:3]
    cases = 0
    for state in product(locations, repeat=3):
        for atom, destination in product(labels, locations):
            ledger = m.Ledger(catalog)
            ledger.universe = set(labels)
            ledger.locations = dict(zip(labels, state))
            ledger.fresh = sum(x != 'outside' for x in state)
            before = dict(ledger.locations)
            ledger.move({atom}, destination)
            require(len(ledger.universe) == 3, 'Transfer changed the admitted universe')
            expected = dict(before, **{atom: destination})
            require(ledger.locations == expected, 'Ownership update mismatch')
            require(ledger.old_inward == int(before[atom] == 'outside' and destination != 'outside'),
                    'Old-origin crossing count mismatch')
            require(ledger.exported == int(before[atom] != 'outside' and destination == 'outside'),
                    'Export crossing count mismatch')
            cases += 1
    ledger = m.Ledger(catalog)
```

```
    admitted = ledger.admit({'batch': 'repeat', 'parent': 'a', 'stock': 'Dm'})
    one = {min(admitted)}
    for _ in range(100):
        ledger.move(one, 'outside'); ledger.move(one, 'free')
    require(len(ledger.universe) == 20 and ledger.old_inward == ledger.exported == 100,
            'Repeated crossing is not fresh admission')
    bad = deepcopy(ledger); del bad.locations[min(admitted)]
    reject('missing_owned_origin', bad.validate, 'missing or hidden carbon')
    bad = deepcopy(ledger); bad.locations['forged'] = 'free'
    reject('invented_owned_origin', bad.validate, 'missing or hidden carbon')
    reject('unadmitted_transfer', lambda: ledger.move({'forged'}, 'bound'), 'unadmitted carbon')
    for feed in ['CO2', 'NaHCO3', 'HCN', 'C2H6O']:
        reject('noneligible_feed_' + feed, lambda feed=feed: ledger.carbon_free(feed), 'carbon-bearing extra feed')
    return {'exhaustive_three_label_single_transfers': cases,
            'repeated_export_return_cycles': 100, 'admitted_count_after_cycles': 20}


SIDE = {
    'Ala': '', 'Arg': '5-4b 4b-10b 10b-Nn1 Nn1-11 11-Nn2 11-Nn3',
    'Asn': '5-4b 4b-On1 4b-Nn1', 'Asp': '5-4b 4b-On1 4b-On2',
    'Cys': '5-S1', 'Gln': '5-4b 4b-10b 10b-On1 10b-Nn1',
    'Glu': '5-4b 4b-4a 4a-On1 4a-On2', 'Gly': '',
    'His': '5-4b 4b-10b 10b-Nn1 Nn1-11 11-Nn2 Nn2-4b',
    'Ile': '5-6 5-4b 4b-10b', 'Leu': '5-4b 4b-4a 4b-10b',
    'Lys': '5-4b 4b-10b 10b-11 11-Nn1', 'Met': '5-4b 4b-S1 S1-10b',
    'Phe': '5-4b 4b-10b 10b-11 11-12 12-12a 12a-4a 4a-4b',
    'Pro': '5-4b 4b-10b 10b-Na', 'Ser': '5-On1', 'Thr': '5-On1 5-4b',
    'Trp': '5-4b 4b-10b 10b-Nn1 Nn1-12a 12a-4a 4a-4b 12a-1 1-2 2-3 3-4 4-4a',
    'Val': '5-6 5-4b'}
SIDE['Tyr'] = SIDE['Phe'] + ' 12-O12'
DOUBLE = {'Arg': '11-Nn2', 'Asn': '4b-On1', 'Asp': '4b-On1',
          'Gln': '10b-On1', 'Glu': '4a-On1', 'His': '4b-10b 11-Nn2',
          'Phe': '4b-10b 11-12 12a-4a', 'Tyr': '4b-10b 11-12 12a-4a',
          'Trp': '4b-10b 12a-4a 1-2 3-4'}
FORMULAS = dict(zip(NAMES, ('C3H7NO2 C6H14N4O2 C4H8N2O3 C4H7NO4 C3H7NO2S '
    'C5H10N2O3 C5H9NO4 C2H5NO2 C6H9N3O2 C6H13NO2 C6H13NO2 C6H14N2O2 '
    'C5H11NO2S C9H11NO2 C5H9NO2 C3H7NO3 C4H9NO3 C11H12N2O2 C9H11NO3 C5H11NO2').split()))


def weighted_identity(m, graph, name, prefix=''):
    atoms = graph['atoms']
    g = m.read_graph(graph)
    root = prefix + 'Ca'
    component = next(c for c in g.components() if root in c)
    expected = {tuple(sorted(x.split('-'))): 1
                for x in ('Ca-Cc Ca-Na Cc-Ox Cc-On0 ' + SIDE[name] +
                          ('' if name == 'Gly' else ' Ca-5')).split()}
    for x in ('Cc-Ox ' + DOUBLE.get(name, '')).split():
        expected[tuple(sorted(x.split('-')))] = 2
    actual = {}
    for (a, b), order in g.bonds.items():
        if a in component and b in component and atoms[a] != 'H' and atoms[b] != 'H':
            require(a.startswith(prefix) and b.startswith(prefix), 'Foreign product prefix')
            actual[tuple(sorted((a[len(prefix):], b[len(prefix):])))] = order
    require(actual == expected, name + ': weighted heavy-atom graph mismatch')
    require(g.count(component) == m.formula(FORMULAS[name]), name + ': formula mismatch')
    return component


def product_extensions(m, guards, data):
    require(set(data['rows']) == set(NAMES), 'Exactly twenty named target rows required')
    for name, row in data['rows'].items():
        weighted_identity(m, row['endpoint'], name)
    pair = data['paired_sulfur']
    for prefix, name in [('X3:', 'Cys'), ('X4:', 'Met')]:
        weighted_identity(m, pair['endpoint'], name, prefix)
    # Literal disjoint union of eighteen unary endpoints plus one paired endpoint.
    union = {}; products = set(); ports = 0
    instances = [(name, data['rows'][name], [('', name)]) for name in NAMES if name not in ('Cys', 'Met')]
    instances.append(('sulfur', pair, [('X3:', 'Cys'), ('X4:', 'Met')]))
    for instance, certificate, targets in instances:
        for atom, element in certificate['endpoint']['atoms'].items():
            key = (instance, atom)
```

```
            require(key not in union, 'Simultaneous atom collision')
            union[key] = element
            ports += int(element == '*')
        for prefix, name in targets:
            products |= {(instance, x) for x in weighted_identity(m, certificate['endpoint'], name, prefix)}
    require(len(union) - ports == 1504 and ports == 40, 'Actual-atom/port union count mismatch')
    all_carbon = {x for x, e in union.items() if e == 'C'}
    product_carbon = {x for x in products if union[x] == 'C'}
    require(len(all_carbon) == 400 and len(product_carbon) == 107, 'Simultaneous carbon mismatch')
    # Independently enumerate connected cycle subsets, not the guard's arc loop.
    cycle = guards.B6
    masks = {frozenset(), frozenset({'5', '4b', '10b'} | guards.A6)}
    for bits in product((0, 1), repeat=6):
        subset = {cycle[i] for i, b in enumerate(bits) if b}
        if subset and '4b' not in subset:
            continue
        if subset:
            reached = {next(iter(subset))}
            while True:
                new = reached | {cycle[(i + d) % 6] for i, x in enumerate(cycle)
                                 if x in reached for d in (-1, 1) if cycle[(i + d) % 6] in subset}
                if new == reached:
                    break
                reached = new
            if reached != subset:
                continue
        for optional in (set(), {'6'}):
            masks.add(frozenset({'5'} | optional | subset))
    # All subsets of the only ordinary support, plus all one-vertex additions.
    tested = set()
    support = sorted(set(cycle) | {'5', '6'})
    for bits in product((0, 1), repeat=len(support)):
        S = frozenset(x for x, b in zip(support, bits) if b)
        tested.add(S)
        for vertex in guards.PER:
            tested.add(S | {vertex})
    tested |= masks
    for S in tested:
        require(guards.mask_allowed(set(S)) == (S in masks), 'Independent mask characterization mismatch')
    bad = deepcopy(data['rows']['Arg']['endpoint'])
    for edge in bad['bonds']:
        if set(edge[:2]) == {'11', 'Nn2'}:
            edge[2] = 1
    reject('wrong_target_double_bond', lambda: weighted_identity(m, bad, 'Arg'), 'weighted heavy-atom graph')
    bad = deepcopy(data['rows']['Ala'])
    bad['initial']['bonds'].append(deepcopy(bad['initial']['bonds'][0]))
    reject('duplicate_product_bond', lambda: guards.replay(bad, {'': ({'5'}, {'6', '4b'})}), 'Duplicate bond')
    bad = deepcopy(data['rows']['Ala'])
    bad['endpoint']['atoms']['Ca'] = 'N'
    reject('changed_atom_element', lambda: guards.replay(bad, {'': ({'5'}, {'6', '4b'})}), 'Atom identities')
    bad = deepcopy(pair)
    bad['rewrite_steps'].insert(0, ['join', 'X3:5', 'X4:5'])
    reject('cross_copy_core_join', lambda: guards.replay(bad, {'X3:': ({'5'}, set()), 'X4:': ({'5', '4b', '10b'},
     set())}), 'cross-copy core')
    return {'weighted_target_graphs': 22, 'mask_cases': len(tested), 'allowed_masks': len(masks),
            'simultaneous_instances': 19, 'actual_atoms': len(union) - ports, 'ports': ports,
            'active_carbon': len(all_carbon), 'product_carbon': len(product_carbon),
            'nonproduct_carbon': len(all_carbon - product_carbon)}


def peptide_witness(m, data):
    sequence = ['Gly', 'Pro', 'Cys', 'Met']
    g = m.Graph()
    for i, name in enumerate(sequence):
        source = data['rows'][name]['endpoint']; old = m.read_graph(source)
        component = weighted_identity(m, source, name)
        label = lambda a: str(i) + ':' + a
        for a in component:
            g.atom(label(a), old.atoms[a])
        for (a, b), order in old.bonds.items():
            if a in component and b in component:
                g.bond(label(a), label(b), order)
    g.validate(True); initial_atoms = dict(g.atoms)
    def split(a, b):
```

```
        key = tuple(sorted((a, b)))
        require(g.bonds.get(key) == 1, 'Joining witness must split a single bond')
        del g.bonds[key]; g.validate()
    for i in range(len(sequence) - 1):
        c, o, n = str(i) + ':Cc', str(i) + ':On0', str(i + 1) + ':Na'
        hydrogens = sorted(b if a == n else a for (a, b) in g.bonds
                           if n in (a, b) and g.atoms[b if a == n else a] == 'H')
        require(bool(hydrogens), 'No available N-H for peptide joining')
        h = hydrogens[0]
        split(c, o); split(n, h)
        g.bond(c, n); g.validate(); g.bond(o, h); g.validate(True)
    require(g.atoms == initial_atoms, 'Peptide joining changed atom identities')
    components = g.components()
    peptide = next(c for c in components if '0:Ca' in c)
    waters = [c for c in components if c != peptide]
    require(len(waters) == 3 and all(g.count(c) == Counter(H=2, O=1) for c in waters),
            'Joining must produce three complete waters')
    require(g.count(peptide) == m.formula('C15H26N4O5S2'), 'Tetrapeptide formula mismatch')
    require({a for a in peptide if g.atoms[a] == 'C'} == {a for a, e in g.atoms.items() if e == 'C'},
            'Peptide lost carbon')
    return {'sequence': sequence, 'primitive_edits': 12, 'waters': 3,
            'peptide_formula': 'C15H26N4O5S2', 'chemical_feasibility_claimed': False}


def count_paths(m):
    observed = events = 0
    for scale, fraction in product((1, 7, 11), (F(0), F(1, 2), F(1))):
        schedule = {}
        for k in range(1, 10 * scale + 1):
            schedule.setdefault(F(2 * k, 5 * scale), []).append('production')
        if fraction:
            for k in range(1, int(10 * scale * fraction) + 1):
                schedule.setdefault(F(2 * k, 5 * scale * fraction), []).append('deactivation')
        B = W = 0
        for time, kinds in sorted(schedule.items()):
            # Production before deactivation at coincident thresholds.
            for kind in sorted(kinds, key=lambda x: x != 'production'):
                if kind == 'production':
                    require(130 * scale - 10 * B >= 10 and 30 * scale - 2 * B >= 2, 'Count reserve exhausted')
                    B += 1
                else:
                    require(B > W, 'Deactivation before available production')
                    W += 1
                counts = [130*scale-10*B, 30*scale-2*B, B-W, W, B, 2*B, 10*B, 2*B]
                b, c, p, w, l, h, r, z = counts
                require(min(counts) >= 0 and b + 4*(p+w+l)+h == 130*scale,
                        'Microstep residue ledger failure')
                require(b+r == 130*scale and c+z == 30*scale, 'Microstep reserve ledger failure')
                events += 1
            projection = m.count_projection(25*time, fraction*F(5, 2)*time, scale)
            require([F(v, scale) for v in counts] == list(projection.values()), 'Completed timestamp projection
     mismatch')
            observed += 1
        require(B+W <= 20*scale, 'Finite path exceeds proved event count')
    for scale in range(11, 101):
        require(F(8, 5)-F(1, scale) > F(3, 2), 'Finite-count functional margin')
    return {'paths': 9, 'ordered_microsteps': events, 'completed_timestamps': observed,
            'threshold_margin_scales_checked': 90,
            'scope': 'Exact prescribed-time stoichiometry; not physical kinetics or atom maps'}


def model_extensions(d):
    # Material terms of the vector field are evaluated exactly on a rational grid.
    cases = 0
    for b, c, p, s, light in product((F(30), F(130)), (F(10), F(30)),
                                     (F(0), F(3, 2), F(3)), (F(0), F(1)), (0, 1)):
        factor = b/(1+b)*c/(F(1,10)+c)*(F(19,20)+s/20)
        J = (10+20*p/(1+p))*factor
        vector = [-J, -J/5, J/10-p, p, J/10, J/5, J, J/5]
        rows = [[1,0,4,4,4,1,0,0], [1,0,0,0,0,0,1,0], [0,1,0,0,0,0,0,1]]
        for row in rows:
            require(sum(x*y for x,y in zip(row, vector)) == 0, 'Exact instantaneous material ledger')
        numeric = d.rhs([float(b),float(c),float(p),0,0,0,0,0,float(s),200,160],light,True)
        require(max(abs(float(x)-y) for x,y in zip(vector,numeric[:8])) < 1e-12, 'RHS implementation mismatch')
```

```
        require(20*factor/(1+p)**2 > 0 and J <= 25, 'Feedback sign or nominal flux ceiling')
        cases += 1
    def balanced(col):
        require(col[0]+4*(col[2]+col[3]+col[4])+col[5] == 0, 'Residue ledger is not conserved')
    reject('deactivation_annihilates_chain', lambda: balanced([0,0,-1,0,0,0,0,0]), 'not conserved')
    reject('incorrect_processing_yield', lambda: balanced([-10,-2,2,0,1,2,10,2]), 'not conserved')
    # Exact Donnan cases: integer Pythagorean triples give an exact square root.
    donnan = 0
    for Q, twice_csV, root in [(3,4,5),(5,12,13),(8,15,17),(7,24,25)]:
        V = F(10); cs = F(twice_csV,20)
        plus, minus = F(root+Q,2), F(root-Q,2)
        excess = F(root,10)-2*cs
        require(plus-minus == Q and plus*minus == cs*cs*V*V, 'Donnan closure identity')
        require(0 <= excess <= F(Q*Q)/(4*cs*V*V), 'Donnan rationalized bound')
        donnan += 1
    return {'exact_RHS_grid_cases': cases, 'exact_Donnan_cases': donnan,
            'source_rational_certificate': d.rational_checks(),
            'analytic_ODE_existence_machine_checked_here': False}


def conditional_extensions():
    geometry = 0
    for r1 in range(1, 9):
        for r2 in range(r1, 9):
            A = r1*r1+r2*r2; V = r1**3+r2**3
            alpha = F(r1**3,V); a = F(r1*r1,A)
            v2 = F(V*V,A**3)
            require(F(1,2) <= v2 < 1 and 0 < alpha <= F(1,2), 'Taut branch domain')
            require(a+(1-a) == 1 and alpha+(1-alpha) == 1, 'Normalized split not conserved')
            require(v2*alpha*alpha/a**3 == 1 and v2*(1-alpha)**2/(1-a)**3 == 1,
                    'Daughter spherical-capacity equality')
            geometry += 1
    for v2 in [F(1,100),F(1,3),F(49,100)]:
        require(v2*F(1,4)/F(1,8) == 2*v2 < 1, 'Slack symmetric daughter screen')
    def partition(pre, left, right, out, minimum=(1,1)):
        require(all(type(x) is int and x >= 0 for x in (pre,left,right,out)), 'Noninteger complete assembly')
        require(left+right+out == pre, 'Partition conservation')
        require(left >= minimum[0] and right >= minimum[1], 'Daughter complete-assembly minimum')
    allocations = 0
    for pre in range(9):
        possible = []
        for left in range(pre+1):
            for right in range(pre-left+1):
                if left >= 1 and right >= 1:
                    partition(pre,left,right,pre-left-right); possible.append((left,right))
                    allocations += 1
        require(bool(possible) == (pre >= 2), 'One-pore inheritance obstruction')
    reject('fractional_pore_daughters',lambda:partition(1,F(1,2),F(1,2),0),'Noninteger')
    reject('missing_second_complete_pore',lambda:partition(1,1,0,0),'complete-assembly minimum')
    reject('double_used_pore',lambda:partition(1,1,1,0),'Partition conservation')
    # Resource-aware induction examples, including costs for every binary parent.
    renewals = 0
    for N, cost, floor in product(range(1,9),range(1,5),(0,3)):
        reserve = N*cost+floor
        for j in range(N):
            require(reserve >= cost+floor,'Insufficient paid cycle reserve')
            reserve -= cost
        require(reserve == floor,'Renewal floor mismatch')
        require(sum(2**j for j in range(N)) == 2**N-1,'Binary parent budget identity')
        renewals += 1
    slope_cases = 0
    for numerator in range(-50,31):
        slope = F(numerator,10)
        require((abs((1+slope)/2) < 1) == (-3 < slope < 1),'Scalar return-slope criterion')
        slope_cases += 1
    reject('negative_slope_false_attraction',lambda:require(abs((1+F(-4))/2)<1,'Not a contraction'),'Not a
     contraction')
    # A concrete transfer-margin arithmetic witness, not its molecular antecedent.
    require(F(1,25)+F(1,20) < F(1,10),'Functional transfer margin')
    reject('zero_transfer_margin',lambda:require(F(1,20)+F(1,20)<F(1,10),'Strict margin required'),'Strict margin'
     )
    return {'exact_taut_geometry_cases':geometry,'exact_slack_geometry_cases':3,
            'integer_viable_partition_cases':allocations,'renewal_budget_cases':renewals,
            'scalar_return_slope_cases':slope_cases,
```

```
            'actual_scission_path_supplied':False,'actual_return_map_supplied':False,
            'scope':'Finite arithmetic consequences of conditional division and transfer hypotheses'}


def numerical_reproduction(out, baseline):
    with (ROOT/'data/operating_trajectories.csv').open() as f:
        frozen=list(csv.DictReader(f))
    with (out/'operating_trajectories.csv').open() as f:
        fresh=list(csv.DictReader(f))
    require(len(frozen)==len(fresh)==802,'Trajectory row count mismatch')
    error=0.0
    for a,b in zip(frozen,fresh):
        require(a.keys()==b.keys(),'Trajectory columns differ')
        for key in a:
            x,y=float(a[key]),float(b[key])
            require(math.isfinite(x) and math.isfinite(y),'Nonfinite trajectory value')
            error=max(error,abs(x-y))
            require(abs(x-y)<=1e-8*max(1,abs(x),abs(y)),'Frozen trajectory mismatch')
    runs=baseline['verify_dynamics']['illustrations']['runs']
    require(len(runs)==4,'Both feedback states and both resolutions required')
    return {'scope':'Unvalidated floating-point RK4; consistency only, no global error enclosure',
            'fresh_runs':runs,'compared_rows':802,'maximum_frozen_regeneration_difference':error}


def encode(value):
    if isinstance(value,F):
        return {'fraction':str(value),'decimal':float(value)}
    raise TypeError(type(value).__name__)


def main():
    parser=argparse.ArgumentParser(description=__doc__)
    parser.add_argument('--output-dir',type=Path,default=ROOT/'results')
    args=parser.parse_args();out=args.output_dir.resolve();out.mkdir(parents=True,exist_ok=True)
    hashes=validate_hashes()
    baseline=baseline_runs(out)
    m=load_module('verify_materials');guards=load_module('verify_product_guards');d=load_module('verify_dynamics')
    cat=json.loads((ROOT/'data/input_catalog.json').read_text());validate_catalog(cat)
    bad=deepcopy(cat);bad['types'][0]['arms'].pop()
    reject('truncated_arm_list',lambda:validate_catalog(bad),'locants/arms')
    catalog={r['id']:r for r in cat['types']}
    data=json.loads((ROOT/'data/article75_product_certificates.json').read_text())
    checks={'materials':material_extensions(m,catalog),
            'products':product_extensions(m,guards,data),
            'peptide_joining':peptide_witness(m,data),
            'count_paths':count_paths(m),
            'model_arithmetic':model_extensions(d),
            'conditional_interfaces':conditional_extensions(),
            'numerical_illustration':numerical_reproduction(out,baseline)}
    original_negative_count=(len(baseline['verify_materials']['checks']['negative_tests'])+
                             len(baseline['verify_product_guards']['rejection_tests']))
    require(original_negative_count==21,'Original negative-control coverage changed')
    payload={'status':'PASS','scope':'Partial executable certificate; not a proof of every manuscript theorem',
             'pinned_inputs':hashes,'checks':checks,'supplement_negative_controls':NEGATIVES,
             'original_negative_controls':original_negative_count,
             'not_certified':['analytic Caratheodory existence or universal invariance in a proof assistant',
                              'stock-to-prepared atom map and full port contexts',
                              'chemical feasibility, yields, stereochemistry, or measured functional response',
                              'physical membrane, pore transport, scission, sorting, or daughter return',
                              'infinite funded operation, genomic decoding, replication, or heredity']}
    encoded=json.dumps(payload,indent=2,sort_keys=True,default=encode)+'\n'
    (out/'results_article77.json').write_text(encoded)
    runtime={'python':sys.version,'optimization_level':sys.flags.optimize,
             'semantic_results_sha256':hashlib.sha256(encoded.encode()).hexdigest(),
             'verifier_sha256':sha(Path(__file__))}
    (out/'runtime.json').write_text(json.dumps(runtime,indent=2)+'\n')
    print(json.dumps({'status':'PASS','supplement_negative_controls':len(NEGATIVES),
                      'original_negative_controls':original_negative_count,
                      'semantic_results_sha256':runtime['semantic_results_sha256'],
                      'optimization_level':sys.flags.optimize},indent=2))


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

auditors/verify_dynamics.py

```python
# Publication adapter: fixed supplied checker, with asserts expanded to
# explicit branches (effective under -O), and configurable output directory.
"""Portable stdlib audit of the manuscript's reduced-model arithmetic.

Exact Fraction inequalities are the certificate checks. RK4 output is separately
labelled illustrative and is not used to prove existence, threshold crossing,
error bounds, physical realization, neck formation, or molecule-level kinetics.
No original source file is read or modified by this checker.
"""
from fractions import Fraction as F
import argparse
import csv
import json
import math
from pathlib import Path
import os

def rational_checks():
    mu = F(19, 20) * F(30, 31) * F(100, 101)
    delta = F(11, 5) * mu - F(3, 2)
    time = F(3, 2) / delta
    if not mu == F(2850, 3131):
        raise AssertionError('Certificate predicate failed at original line 21')
    if not delta == F(3147, 6262) > 0:
        raise AssertionError('Certificate predicate failed at original line 22')
    if not time == F(3131, 1049) < F(597, 200) < 4:
        raise AssertionError('Certificate predicate failed at original line 23')
    if not delta < mu:
        raise AssertionError('Certificate predicate failed at original line 24')
    terminal_delta = mu * (1 + 2 * F(8, 5) / (1 + F(8, 5))) - F(8, 5)
    terminal_time = F(8, 5) / terminal_delta
    if not terminal_delta == F(87626, 203515) > 0:
        raise AssertionError('Certificate predicate failed at original line 27')
    if not terminal_time == F(162812, 43813) < F(3717, 1000) < 4:
        raise AssertionError('Certificate predicate failed at original line 28')
    lower_water = F(20, 100) - F(1, 10)
    upper_water = F(50, 600) + F(1, 600 ** 2) - F(1, 10)
    global_upper_water = F(59, 600) + F(1, 600 ** 2) - F(1, 10)
    geo_squared = 36 * F(22, 7) * 600 ** 2 / F(500 ** 3)
    if not lower_water > 0 > upper_water:
        raise AssertionError('Certificate predicate failed at original line 33')
    if not upper_water == -F(5999, 360000):
        raise AssertionError('Certificate predicate failed at original line 34')
    if not global_upper_water == -F(599, 360000) < 0:
        raise AssertionError('Certificate predicate failed at original line 35')
    if not geo_squared < F(1, 3):
        raise AssertionError('Certificate predicate failed at original line 36')
    if not F(300, 660) > F(2, 5):
        raise AssertionError('Certificate predicate failed at original line 37')
    if not 160 - 4 * (25 + 2 + 3) == 40:
        raise AssertionError('Certificate predicate failed at original line 38')
    robust_coarse_flux = (F(101, 10) + F(101, 5)) * F(201, 200)
    if not robust_coarse_flux / 10 - F(99, 100) * F(31, 10) < 0:
        raise AssertionError('Certificate predicate failed at original line 40')
    robust_flux = (F(101, 10) + F(101, 5) * F(31, 41)) * F(201, 200)
    if not robust_flux == F(2091003, 82000) < 26:
        raise AssertionError('Certificate predicate failed at original line 42')
    robust_mu = F(189, 200) * F(25) / (25 + F(101, 100)) * F(41, 5) / (F(41, 5) + F(101, 1000))
    robust_delta = robust_mu * F(1089, 500) - F(303, 200)
    robust_time = F(3, 2) / robust_delta
    if not robust_mu == F(717500, 799663):
        raise AssertionError('Certificate predicate failed at original line 46')
    if not robust_delta == F(70245111, 159932600) > 0:
        raise AssertionError('Certificate predicate failed at original line 47')
    if not robust_delta < robust_mu * F(99, 100):
        raise AssertionError('Certificate predicate failed at original line 48')
    if not robust_time == F(239898900, 70245111) < F(427, 125):
        raise AssertionError('Certificate predicate failed at original line 49')
    robust_area = 501 + (F(41, 100) + F(101, 100) + F(101, 500)) * 104
    robust_N = F(201, 10) + F(3, 10) * 104
    robust_lower_water = F(199, 1000) - F(101, 1000) - F(1, 1000)
```

```
    robust_upper_water = robust_N / 600 + F(21, 10) ** 2 / (4 * F(99, 100) * 600 ** 2) - F(99, 1000) + F(1, 1000)
    robust_ablation = F(101, 10) * F(201, 200) / 10 / F(99, 100)
    if not robust_area == F(83711, 125) < 670:
        raise AssertionError('Certificate predicate failed at original line 55')
    if not robust_N == F(513, 10):
        raise AssertionError('Certificate predicate failed at original line 56')
    if not robust_lower_water == F(97, 1000) > 0 > robust_upper_water:
        raise AssertionError('Certificate predicate failed at original line 57')
    if not 36 * F(22, 7) * 600 ** 2 / F(499 ** 3) < F(1, 3):
        raise AssertionError('Certificate predicate failed at original line 58')
    if not F(300, 670) > F(2, 5):
        raise AssertionError('Certificate predicate failed at original line 59')
    if not robust_ablation < F(513, 500):
        raise AssertionError('Certificate predicate failed at original line 60')
    if not 160 - 4 * (26 + 2 + F(31, 10)) == F(178, 5) > 35:
        raise AssertionError('Certificate predicate failed at original line 61')
    processing = [-10, -2, 1, 0, 1, 2, 10, 2]
    deactivation = [0, 0, -1, 1, 0, 0, 0, 0]
    carbon_rows = [[1, 0, 4, 4, 4, 1, 0, 0], [1, 0, 0, 0, 0, 0, 1, 0], [0, 1, 0, 0, 0, 0, 0, 1]]
    for row in carbon_rows:
        for col in [processing, deactivation]:
            if not sum((x * y for x, y in zip(row, col))) == 0:
                raise AssertionError('Certificate predicate failed at original line 69')
    if not 4 * (1 + 1) + 2 == 10:
        raise AssertionError('Certificate predicate failed at original line 70')
    for area_share in [F(1, 2), F(1, 3), F(1, 10)]:
        if not area_share + (1 - area_share) == 1:
            raise AssertionError('Certificate predicate failed at original line 73')
    return {'status': 'PASS exact rational arithmetic only', 'nominal': {'mu': mu, 'bootstrap_speed_lower': delta,
      'bootstrap_time_upper': time, 'strict_margin_threshold': F(8, 5), 'strict_margin_time_upper': terminal_time,
      'strict_margin_speed_lower': terminal_delta, 'water_lower_drive_bound': lower_water, '
     water_upper_drive_bound': upper_water, 'global_water_upper_drive_bound': global_upper_water, '
     reduced_volume_squared_upper': geo_squared, 'energy_reserve_lower': 40}, 'robust': {'mu': robust_mu, '
     bootstrap_speed_lower': robust_delta, 'bootstrap_time_upper': robust_time, 'flux_upper': robust_flux, '
     area_upper': robust_area, 'neutral_particle_upper': robust_N, 'water_lower_drive_bound': robust_lower_water, '
     water_upper_drive_bound': robust_upper_water, 'ablation_active_upper': robust_ablation, 'energy_reserve_lower'
     : F(178, 5)}, 'stoichiometry': {'state_order': ['b', 'c', 'p', 'w', 'l', 'h', 'r', 'z'], '
     integer_processing_block': processing, 'deactivation': deactivation, 'conserved_rows': carbon_rows}, '
     not_certified': ['chemical kinetics or thermodynamic adequacy', 'atom-labelled donor graphs', 'stable
     membrane shape', 'molecular neck pathway', 'daughter sorting', 'a return map for the four-period witness']}
NAMES = ['b', 'c', 'p', 'w', 'l', 'h', 'r', 'z', 's', 'V', 'E']

def rhs(y, light, feedback):
    b, c, p, w, l, h, r, z, s, V, E = y
    J = (10 + (20 * p / (1 + p) if feedback else 0)) * b / (1 + b) * c / (0.1 + c) * (0.95 + 0.05 * s)
    A = 500 + 4 * (p + w) + r + z
    D = 4 / V ** 2 / (math.sqrt(4 + 4 / V ** 2) + 2)
    return [-J, -J / 5, J / 10 - p, p, J / 10, J / 5, J, J / 5, 2 * light * (1 - s) - s, A / 100 * ((20 + l + h) /
      V + D - 0.1), -(J + 2 * light * (1 - s) + p)]

def rk4(y, dt, light, feedback):
    a = rhs(y, light, feedback)
    b = rhs([v + dt * k / 2 for v, k in zip(y, a)], light, feedback)
    c = rhs([v + dt * k / 2 for v, k in zip(y, b)], light, feedback)
    d = rhs([v + dt * k for v, k in zip(y, c)], light, feedback)
    return [v + dt * (k + 2 * l + 2 * m + n) / 6 for v, k, l, m, n in zip(y, a, b, c, d)]

def simulate(steps_per_half, feedback, save_rows=False):
    dt = 0.5 / steps_per_half
    y = [130.0, 30.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 200.0, 160.0]
    rows = []
    first = None
    first16 = None
    residual = 0

    def row(t):
        b, c, p, w, l, h, r, z, s, V, E = y
        A = 500 + 4 * (p + w) + r + z
        return [t, int(feedback), int(round(t / dt) // steps_per_half % 2 == 0)] + y + [A, 20 + l + h, 130 - b, 6 *
      math.sqrt(math.pi) * V / A ** 1.5]
    if save_rows:
        rows.append(row(0))
    for i in range(8 * steps_per_half):
        light = 1 if i // steps_per_half % 2 == 0 else 0
```

```
        y = rk4(y, dt, light, feedback)
        t = (i + 1) * dt
        b, c, p, w, l, h, r, z, s, V, E = y
        if first is None and p >= 1.5:
            first = t
        if first16 is None and p >= 1.6:
            first16 = t
        residual = max(residual, abs(b + 4 * (p + w + l) + h - 130), abs(b + r - 130), abs(c + z - 30))
        if not (30 - 1e-09 <= b <= 130 and 10 - 1e-09 <= c <= 30):
            raise AssertionError('Certificate predicate failed at original line 130')
        if not (0 <= p <= 3 + 1e-09 and 0 <= s <= 1 + 1e-09 and (100 <= V <= 600) and (E >= 40 - 1e-09)):
            raise AssertionError('Certificate predicate failed at original line 131')
        if save_rows and (i + 1) % (steps_per_half // 50) == 0:
            rows.append(row(t))
    if feedback:
        if not first <= float(F(3131, 1049)) + dt:
            raise AssertionError('Certificate predicate failed at original line 134')
        if not y[2] >= 1.6:
            raise AssertionError('Certificate predicate failed at original line 135')
    elif not (first is None and y[2] < 1):
        raise AssertionError('Certificate predicate failed at original line 136')
    return ({'dt': dt, 'feedback': feedback, 'threshold_crossing_grid': first, 'margin_threshold_crossing_grid':
     first16, 'final': dict(zip(NAMES, y)), 'maximum_ledger_residual': residual}, rows)

def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('--output-dir', type=Path, default=Path(__file__).resolve().parent)
    args = parser.parse_args()
    args.output_dir.mkdir(parents=True, exist_ok=True)
    exact = rational_checks()
    reports = []
    allrows = []
    for feedback in [True, False]:
        for halfsteps in [500, 1000]:
            report, rows = simulate(halfsteps, feedback, save_rows=halfsteps == 1000)
            reports.append(report)
            allrows.extend(rows)
    for a, b in [(reports[0], reports[1]), (reports[2], reports[3])]:
        if not max((abs(a['final'][k] - b['final'][k]) for k in NAMES)) < 1e-07:
            raise AssertionError('Certificate predicate failed at original line 153')
    report = {'exact_certificate': exact, 'illustrations': {'status': 'Unvalidated floating-point RK4
     illustrations, not proof certificates', 'runs': reports}, 'method': 'Every RK4 step is wholly within one
     light mode; half-period boundaries align exactly by integer step index.'}

    def encode(x):
        if isinstance(x, F):
            return {'fraction': str(x), 'decimal': float(x)}
        raise TypeError(type(x).__name__)
    out = args.output_dir / 'dynamics_checks.json'
    out.write_text(json.dumps(report, indent=2, default=encode) + '\n')
    with (args.output_dir / 'operating_trajectories.csv').open('w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(['time', 'feedback', 'light'] + NAMES + ['A', 'N_free', 'S', 'reduced_volume'])
        writer.writerows(allrows)
    print(json.dumps({'status': 'PASS', 'exact_rational_checks': True, 'illustrative_runs': len(reports), '
     csv_rows': len(allrows), 'report': str(out), 'nominal_bootstrap_time_upper': str(F(3131, 1049)), '
     robust_bootstrap_time_upper': str(F(239898900, 70245111))}, indent=2))
if __name__ == '__main__':
    main()
```

auditors/verify__materials.py

```
# Publication adapter: fixed supplied checker, with asserts expanded to
# explicit branches (effective under -O), and configurable output directory.
"""Portable standard-library material checks, including rejecting mutations.

Checks constitutional graphs, immutable-carbon accounting, finite assignments,
source-certificate replay, and amount/count projection identities. It is not a
chemical, stereochemical, geometry, or stock-to-prepared-path validator.
"""
from pathlib import Path
```

```
import os
from collections import Counter, deque
from fractions import Fraction
from copy import deepcopy
from itertools import combinations, product
import argparse
import hashlib
import json
import re
HERE = Path(__file__).resolve().parents[1]
DATA = HERE / 'data'
RESULTS = Path(os.environ.get('A77_RESULTS_DIR', HERE / 'results'))
CORE = ['1', '2', '3', '4', '4a', '4b', '5', '6', '6a', '7', '8', '9', '10', '10a', '10b', '11', '12', '12a']
CAP = {'C': 4, 'H': 1, 'N': 3, 'O': 2, 'S': 2, 'P': 5, '*': 1}

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

def formula(text):
    tokens = re.findall('([A-Z][a-z]*)([0-9]*)', text)
    require(''.join((e + n for e, n in tokens)) == text, 'invalid formula')
    ans = Counter()
    for e, n in tokens:
        ans[e] += int(n or 1)
    return ans

def combine(*terms):
    ans = Counter()
    for coefficient, text in terms:
        for element, amount in formula(text).items():
            ans[element] += coefficient * amount
    return {e: n for e, n in ans.items() if n}

def origin(batch, parent, local):
    return json.dumps([batch, parent, local], separators=(',', ':'))

class Graph:

    def __init__(self):
        self.atoms = {}
        self.bonds = {}

    def atom(self, label, element):
        require(label not in self.atoms, 'duplicate atom origin')
        self.atoms[label] = element

    def bond(self, a, b, order=1):
        require(a in self.atoms and b in self.atoms and (a != b), 'invalid endpoint')
        key = tuple(sorted((a, b)))
        require(key not in self.bonds, 'duplicate bond specification')
        require(isinstance(order, int) and 1 <= order <= 3, 'invalid bond order')
        self.bonds[key] = order

    def degrees(self):
        d = Counter()
        for (a, b), v in self.bonds.items():
            d[a] += v
            d[b] += v
        return d

    def saturate(self):
        d = self.degrees()
        for a, e in list(self.atoms.items()):
            require(d[a] <= CAP[e], 'over-valent skeleton')
            b, p, l = json.loads(a)
            for i in range(CAP[e] - d[a]):
                h = origin(b, p, 'H@' + l + ':' + str(i))
                self.atom(h, 'H')
                self.bond(a, h)
        self.validate(True)

    def validate(self, closed=False):
        d = self.degrees()
```

```
        for (a, b), v in self.bonds.items():
            require(a in self.atoms and b in self.atoms and (a != b), 'invalid edge')
            require(isinstance(v, int) and 1 <= v <= 3, 'bad bond multiplicity')
        for a, e in self.atoms.items():
            require(e in CAP and d[a] <= CAP[e], 'valence violation')
            if closed:
                require(d[a] == CAP[e], 'unclosed valence')

    def count(self, labels=None):
        return Counter((self.atoms[a] for a in (self.atoms if labels is None else labels) if self.atoms[a] != '*'))


    def components(self):
        adj = {a: set() for a in self.atoms}
        for a, b in self.bonds:
            adj[a].add(b)
            adj[b].add(a)
        unseen = set(self.atoms)
        out = []
        while unseen:
            root = min(unseen)
            reached = {root}
            todo = [root]
            while todo:
                a = todo.pop()
                for b in adj[a] - reached:
                    reached.add(b)
                    todo.append(b)
            out.append(reached)
            unseen -= reached
        return out

    def serial(self):
        return {'atoms': self.atoms, 'bonds': [[a, b, v] for (a, b), v in sorted(self.bonds.items())]}

def add_stock(g, catalog, record):
    require(record['stock'] in catalog, 'ineligible carbon-bearing stock')
    row = catalog[record['stock']]
    batch, parent = (record['batch'], record['parent'])
    lab = lambda a: origin(batch, parent, a)
    for x in CORE:
        g.atom(lab('core:' + x), 'C')
    for i, x in enumerate(CORE):
        g.bond(lab('core:' + x), lab('core:' + CORE[(i + 1) % 18]), 2 if i % 2 == 0 else 1)
    for a, b in [('4a', '12a'), ('4b', '10b'), ('6a', '10a')]:
        g.bond(lab('core:' + a), lab('core:' + b))
    terminal = {}
    for site, arm in zip(row['sites'], row['arms']):
        key = 'arm:' + str(site) + ':'
        o = lab(key + 'o0')
        g.atom(o, 'O')
        g.bond(lab('core:' + str(site)), o)
        if arm == 'e':
            previous = o
            for name, e in [('e1', 'C'), ('e2', 'C'), ('o1', 'O')]:
                now = lab(key + name)
                g.atom(now, e)
                g.bond(previous, now)
                previous = now
            terminal[site] = previous
        else:
            require(arm == 'h', 'unregistered arm')
            terminal[site] = o
    return terminal

def add_phosphate(g, batch, parent, oxygens):
    require(len(oxygens) in (1, 2), 'phosphate ester arity')
    require(len(set(oxygens)) == len(oxygens), 'repeated ester atom')
    for a in oxygens:
        require(g.atoms.get(a) == 'O' and g.degrees()[a] == 1, 'spent hydroxyl arm')
    p = origin(batch, parent, 'P')
    g.atom(p, 'P')
    ox = origin(batch, parent, 'oxo')
    g.atom(ox, 'O')
```

```
    g.bond(p, ox, 2)
    for a in oxygens:
        g.bond(p, a)
    for j in range(3 - len(oxygens)):
        o = origin(batch, parent, 'OH:' + str(j))
        g.atom(o, 'O')
        g.bond(p, o)

def pore_graph(catalog, m, q, p):
    require(all((isinstance(v, int) for v in (m, q, p))), 'integer pore dimensions required')
    require(m >= 3 and q >= 1 and (0 <= p <= 2 * m), 'invalid pore dimensions')
    g = Graph()
    arms = {}
    for i in range(m):
        for j in range(q):
            arms[i, j] = add_stock(g, catalog, {'batch': 'pore', 'parent': f'T_{i}_{j}', 'stock': 'T0'})
    k = 0

    def bridge(atoms):
        nonlocal k
        add_phosphate(g, 'pore', 'phosphate_' + str(k), atoms)
        k += 1
    for i in range(m):
        for j in range(q - 1):
            bridge([arms[i, j][12], arms[i, j + 1][6]])
    for i in range(m):
        for j in range(q):
            bridge([arms[i, j][9], arms[(i + 1) % m, j][3]])
    terminals = [arms[i, q - 1][12] for i in range(m)] + [arms[i, 0][6] for i in range(m)]
    for a in terminals[:p]:
        bridge([a])
    g.saturate()
    require(len(g.components()) == 1, 'pore scaffold disconnected')
    b = m * (2 * q - 1)
    require(g.count() == Counter(C=18 * m * q, H=12 * m * q - b + p, O=4 * m * q + 2 * b + 3 * p, P=b + p), 'pore
     graph formula')
    return g

def membrane_graph(catalog, stock):
    g = Graph()
    a = add_stock(g, catalog, {'batch': 'membrane', 'parent': 'a', 'stock': stock})
    b = add_stock(g, catalog, {'batch': 'membrane', 'parent': 'b', 'stock': stock})
    add_phosphate(g, 'membrane', 'bridge', [a[12], b[12]])
    g.saturate()
    require(len(g.components()) == 1, 'membrane precursor disconnected')
    return g

def carbon_labels(catalog, record):
    g = Graph()
    add_stock(g, catalog, record)
    return {a for a, e in g.atoms.items() if e == 'C'}

class Ledger:

    def __init__(self, catalog):
        self.catalog = catalog
        self.universe = set()
        self.locations = {}
        self.fresh = 0
        self.admitted = 0
        self.old_inward = 0
        self.exported = 0

    def admit(self, record, location='free'):
        new = carbon_labels(self.catalog, record)
        require(not self.universe & new, 'duplicate stock admission')
        self.universe |= new
        require(location in {'free', 'outside'}, 'admission location')
        self.locations.update({a: location for a in new})
        self.admitted += len(new)
        if location == 'free':
            self.fresh += len(new)
        self.validate()
        return new
```

```
    def move(self, labels, destination):
        require(destination in {'free', 'bound', 'waste', 'recycle', 'outside', 'loss'}, 'unknown compartment')
        require(labels <= self.universe, 'unadmitted carbon movement')
        for a in labels:
            old = self.locations[a]
            if old == 'outside' and destination != 'outside':
                self.old_inward += 1
            if old != 'outside' and destination == 'outside':
                self.exported += 1
            self.locations[a] = destination
        self.validate()

    def validate(self):
        require(set(self.locations) == self.universe, 'missing or hidden carbon')
        inside = sum((v != 'outside' for v in self.locations.values()))
        require(inside == self.fresh + self.old_inward - self.exported, 'open carbon balance violated')

    def carbon_free(self, feed):
        require(formula(feed).get('C', 0) == 0, 'carbon-bearing extra feed')

    def rewrite(self, old_atoms, new_atoms):
        require(old_atoms == new_atoms, 'atom deletion, recreation or origin swap')
        require(old_atoms <= self.universe, 'unadmitted rewrite support')

    def allocate(self, groups):
        used = set()
        for support in groups:
            require(support <= self.universe, 'unadmitted allocation')
            require(not used & support, 'double allocation')
            used |= support
        return used

def unit_matching(eligibility):
    match = {}

    def augment(j, seen):
        for u in eligibility[j]:
            if u in seen:
                continue
            seen.add(u)
            if u not in match or augment(match[u], seen):
                match[u] = j
                return True
        return False
    for j in eligibility:
        if not augment(j, set()):
            return None
    return {j: u for u, j in match.items()}

def hall_holds(eligibility):
    demands = list(eligibility)
    for size in range(len(demands) + 1):
        for js in combinations(demands, size):
            neighbors = set().union(*(set(eligibility[j]) for j in js))
            if len(neighbors) < len(js):
                return False
    return True

def packet_choice(alternatives):
    for choices in product(*alternatives):
        seen = set()
        valid = True
        for h in choices:
            if seen & set(h):
                valid = False
                break
            seen |= set(h)
        if valid:
            return choices
    return None

def read_graph(record):
    g = Graph()
```

```
    g.atoms = dict(record['atoms'])
    for a, b, v in record['bonds']:
        g.bond(a, b, v)
    return g

def replay(row, name=None):
    initial = read_graph(row['initial'])
    final = read_graph(row['endpoint'])
    require(initial.atoms == final.atoms, 'source certificate changed atoms')
    initial.validate(True)
    final.validate(True)
    g = deepcopy(initial)
    for op, a, b in row['rewrite_steps']:
        require(a in g.atoms and b in g.atoms and (a != b), 'invalid rewrite vertices')
        key = tuple(sorted((a, b)))
        if op == 'split':
            require(g.bonds.get(key, 0) > 0, 'split absent bond')
            g.bonds[key] -= 1
            if g.bonds[key] == 0:
                del g.bonds[key]
        elif op == 'join':
            g.bonds[key] = g.bonds.get(key, 0) + 1
        else:
            raise ValueError('unknown graph primitive')
        g.validate()
    g.validate(True)
    require(g.bonds == final.bonds, 'wrong terminal graph')
    if name is not None:
        vertices = set(row['product_vertices'])
        require(vertices in g.components(), 'target not an entire molecular component')
        require(g.count(vertices) == formula(row['product_formula']), 'wrong target formula')
        carbons = {a for a in vertices if g.atoms[a] == 'C'}
        require(carbons == set(row['source_carbon_retained']) | {'Ca', 'Cc'}, 'target carbon map')
        require(all((row['persistent_origin'][a] == 'framework' for a in row['source_carbon_retained'])), 'core
     origin tag')
        require(all((row['persistent_origin'][a] == 'arm_derived_backbone' for a in ('Ca', 'Cc'))), 'backbone
     origin tag')
        require(row['stereo_proved_by_graph_rules'] is False, 'unsupported stereochemical claim')
        return (len(carbons), len(row['source_carbon_retained']), len(row['rewrite_steps']))
    return len(row['rewrite_steps'])

def require_complete_preparation(preparation):
    require(preparation.get('stock_atom_map') is not None, 'missing original stock atom map')
    require(preparation.get('complete_context_graphs') is not None, 'unresolved carbon-bearing ports')
    require(preparation.get('preparation_steps') is not None, 'missing preparation word')

def full_noninterference(left, right):
    return not set(left['writes']) & (set(right['reads']) | set(right['writes'])) and (not set(right['writes']) &
     set(left['reads']))

def count_projection(S, w, M):
    require(isinstance(M, int) and M > 0, 'scale must be positive integer')
    require(0 <= w <= S / 10 and 0 <= S <= 130, 'invalid projection domain')
    B = M * S // 10
    W = M * w // 1
    require(W <= B, 'deactivation exceeds produced chains')
    return {'b': Fraction(130) - Fraction(10 * B, M), 'c': Fraction(30) - Fraction(2 * B, M), 'p': Fraction(B - W,
      M), 'w': Fraction(W, M), 'l': Fraction(B, M), 'h': Fraction(2 * B, M), 'r': Fraction(10 * B, M), 'z':
     Fraction(2 * B, M)}

def rejection_tests(catalog, products):
    bad = {}

    def rejected(name, fn):
        try:
            fn()
        except (ValueError, KeyError, AssertionError):
            bad[name] = True
        else:
            raise AssertionError('mutation accepted: ' + name)
    stock = {'batch': 'a', 'parent': 'one', 'stock': 'D2'}
    l = Ledger(catalog)
    aa = l.admit(stock)
    rejected('duplicate_stock_copy', lambda: l.admit(stock))
```

```
    rejected('free_adenosine_import', lambda: l.admit({'batch': 'a', 'parent': 'foreign', 'stock': 'adenosine'}))
    rejected('organic_extra_feed', lambda: l.carbon_free('C2H6O'))
    one = {min(aa)}
    rejected('discarded_carbon', lambda: l.rewrite(aa, aa - one))
    rejected('same_count_unqualified_carbon_swap', lambda: l.rewrite(aa, aa - one | {'unadmitted-carbon'}))
    rejected('double_allocation', lambda: l.allocate([aa, one]))
    rejected('hidden_port_carbon', lambda: require_complete_preparation({'stock_atom_map': {}, 'preparation_steps': []}))
    rejected('origin_class_as_stock_map', lambda: require_complete_preparation({'origin_tag': 'arm_derived_backbone'}))
    broken = deepcopy(products['rows']['Ala'])
    del broken['endpoint']['atoms']['Ca']
    rejected('deleted_Ala_backbone_atom', lambda: replay(broken, 'Ala'))
    broken = deepcopy(products['rows']['Ala'])
    broken['product_formula'] = 'C4H7NO2'
    rejected('wrong_Ala_formula', lambda: replay(broken, 'Ala'))
    rejected('pore_terminal_overuse', lambda: pore_graph(catalog, 3, 3, 7))
    rejected('D2_dimer_claimed_as_40C', lambda: require(membrane_graph(catalog, 'D2').count()['C'] == 40, 'four unused-arm carbons omitted'))
    rejected('terminal_complete_pore_claimed_as_15P', lambda: require(pore_graph(catalog, 3, 3, 3).count()['P'] == 15, 'distal phosphate omitted'))
    rejected('deactivation_before_production', lambda: count_projection(Fraction(1), Fraction(1), 10))
    return bad

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--output', type=Path, default=RESULTS / 'material_verification.json')
    args = parser.parse_args()
    cat = json.loads((DATA / 'input_catalog.json').read_text())
    catalog = {r['id']: r for r in cat['types']}
    fixture = json.loads((DATA / 'material_fixture.json').read_text())
    products = json.loads((DATA / 'article75_product_certificates.json').read_text())
    checks = {}
    stock_counts = {}
    for name, row in catalog.items():
        ne = row['arms'].count('e')
        nh = row['arms'].count('h')
        expected = Counter(C=18 + 2 * ne, H=12 + 4 * ne, O=nh + 2 * ne)
        require(formula(row['formula']) == expected, 'catalog formula')
        g = Graph()
        add_stock(g, catalog, {'batch': 'stock', 'parent': name, 'stock': name})
        g.saturate()
        require(g.count() == expected, 'full stock atom graph formula')
        require(len(g.components()) == 1, 'stock disconnected')
        stock_counts[name] = dict(g.count())
    checks['five_full_stock_graphs'] = stock_counts
    p = pore_graph(catalog, 3, 3, 3)
    p0 = pore_graph(catalog, 3, 3, 0)
    require(p.count() == formula('C162H96O75P18'), 'completed pore formula')
    require(p0.count() == formula('C162H93O66P15'), 'internal pore formula')
    for m in range(3, 7):
        for q in range(1, 5):
            for pn in (0, m, 2 * m):
                pore_graph(catalog, m, q, pn)
    checks['pore_graphs_checked'] = 48
    checks['pore_internal_formula'] = dict(p0.count())
    checks['pore_completed_formula'] = dict(p.count())
    dm = membrane_graph(catalog, 'Dm')
    d2 = membrane_graph(catalog, 'D2')
    require(dm.count() == formula('C40H31O8P'), 'Dm membrane formula')
    require(d2.count() == formula('C44H39O10P'), 'D2 membrane formula')
    require(combine((1, 'C40H31O8P'), (26, 'H2')) == combine((1, 'C40H79O6P'), (2, 'H2O')), 'illustrative lipid balance')
    checks['membrane_Dm_and_D2_formulas'] = {'Dm': dict(dm.count()), 'D2': dict(d2.count()), 'D2_extra_C': 4}
    ledger = Ledger(catalog)
    groups = {}
    totals = Counter()
    for rec in fixture['stock_witness']:
        labels = ledger.admit(rec)
        groups.setdefault(rec['role'], set()).update(labels)
        totals[rec['stock']] += 1
    ledger.allocate(list(groups.values()))
    require(len(ledger.universe) == 810, 'witness carbon total')
    require({k: len(v) for k, v in groups.items()} == fixture['role_carbon'], 'role carbon totals')
```

```
    checks['nonempty_stock_reservation'] = {'carbon': 810, 'stock_counts': dict(totals), 'role_carbon': fixture['role_carbon'], 'scope': 'Explicit stock graphs and disjoint reservations, not a completed cell or preparation path.'}
    l = Ledger(catalog)
    d = l.admit({'batch': 'trace0', 'parent': 'donor', 'stock': 'D2'})
    t = l.admit({'batch': 'trace0', 'parent': 'poreunit', 'stock': 'T0'})
    l.carbon_free('H2O')
    l.carbon_free('NH3')
    l.carbon_free('H3PO4')
    selected = {origin('trace0', 'donor', 'core:5'), origin('trace0', 'donor', 'arm:6:e1'), origin('trace0', 'donor', 'arm:6:e2')}
    l.rewrite(d, d)
    l.move(selected, 'bound')
    l.move(d - selected, 'recycle')
    outgoing = set(sorted(d - selected)[:4])
    l.move(outgoing, 'outside')
    l.admit({'batch': 'trace1', 'parent': 'newstock', 'stock': 'Dm'})
    returned = set(sorted(outgoing)[:2])
    l.move(returned, 'recycle')
    l.move(returned, 'free')
    l.validate()
    reserve = l.admit({'batch': 'trace0', 'parent': 'outside_reserve', 'stock': 'I2'}, location='outside')
    first_entry = set(sorted(reserve)[:5])
    l.move(first_entry, 'free')
    l.validate()
    checks['carbon_trace'] = {'fresh_admitted_inside': l.fresh, 'outward_crossings': l.exported, 'old_origin_inward_crossings': l.old_inward, 'inside': sum((x != 'outside' for x in l.locations.values())), 'admitted_once': len(l.universe), 'note': 'Old-origin inward crossings include five first entries from an admitted external reserve and two genuine returns; returns are optional.'}
    for mask in range(1 << 9):
        elig = {i: [j for j in range(3) if mask & 1 << 3 * i + j] for i in range(3)}
        require((unit_matching(elig) is not None) == hall_holds(elig), 'matching/Hall equivalence')
    require(packet_choice([[{'a', 'b'}], [{'a', 'b'}]]) is None, 'multi-resource packet collision')
    require(packet_choice([[{'a', 'b'}], [{'c'}]]) is not None, 'disjoint packet witness')
    checks['allocation'] = {'exhaustive_unit_slot_graphs': 512, 'general_packet_collision_rejected': True}
    left = {'reads': ['guard'], 'writes': ['x']}
    right = {'reads': [], 'writes': ['guard']}
    require(not full_noninterference(left, right), 'read-write guard conflict missed')
    require(full_noninterference({'reads': ['a'], 'writes': ['a']}, {'reads': ['b'], 'writes': ['b']}), 'disjoint supports')
    checks['full_guard_noninterference'] = True
    initial = Counter(A=1, B=0, C=0)
    reactions = [(Counter(A=1, C=1), Counter(B=1, C=1)), (Counter(B=1), Counter(C=1))]
    require(not any((all((initial[x] >= v for x, v in lhs.items())) for lhs, rhs in reactions)), 'counterexample unexpectedly enabled')
    net = initial.copy()
    for lhs, rhs in reactions:
        for x in set(lhs) | set(rhs):
            net[x] += rhs[x] - lhs[x]
    require(all((v >= 0 for v in net.values())) and net['C'] == 1, 'net endpoint')
    checks['net_stoichiometry_not_reachability'] = True
    targetC = coreC = steps = 0
    for name, row in products['rows'].items():
        a, b, c = replay(row, name)
        targetC += a
        coreC += b
        steps += c
    require(len(products['rows']) == 20 and (targetC, coreC) == (107, 67), 'repertoire carbon sum')
    pair_steps = replay(products['paired_sulfur'])
    pair_atoms = Counter((e for e in products['paired_sulfur']['initial']['atoms'].values() if e != '*'))
    require(pair_atoms == formula('C40H90N8O14S2'), 'shared sulfur packet')
    require(18 * 75 + sum(pair_atoms.values()) == 1504, 'exposed module atom count')
    checks['Article75'] = {'diagnostic_rows': 20, 'diagnostic_primitive_steps': steps, 'joint_sulfur_steps': pair_steps, 'target_C': 107, 'core_C': 67, 'backbone_C': 40, 'active_module_C': 400, 'nonproduct_active_C': 293, 'exposed_actual_atoms': 1504, 'full_stock_preparation_verified': False, 'all_source_extraction_guards_independently_verified': False, 'stereochemistry_verified': False}
    for cm, cr, cc in product((2, 3, 11), (0, 7, 18), (0, 10, 24)):
        require(10 * (cm + cr) + 2 * cc == 8 * cm + 2 * cm + 10 * cr + 2 * cc, 'integer block carbon')
    tested = 0
    for M in (1, 7, 11, 100):
        for S in (Fraction(0), Fraction(1, 3), Fraction(75, 2), Fraction(100), Fraction(1299, 10)):
            for fraction in (Fraction(0), Fraction(1, 3), Fraction(1)):
                w = fraction * S / 10
                out = count_projection(S, w, M)
```

```
                continuous = {'b': 130 - S, 'c': 30 - S / 5, 'p': S / 10 - w, 'w': w, 'l': S / 10, 'h': S / 5, 'r': S, 'z': S / 5}
                for key in continuous:
                    bound = 10 if key in ('b', 'r') else 2 if key in ('c', 'h', 'z') else 1
                    require(abs(out[key] - continuous[key]) < Fraction(bound, M), 'projection error bound')
                require(out['b'] + 4 * (out['p'] + out['w'] + out['l']) + out['h'] == 130, 'residue carbon coefficient')
                require(out['b'] + out['r'] == 130 and out['c'] + out['z'] == 30, 'remnant/carrier coefficients')
                tested += 1
    checks['finite_count_projection'] = {'exact_rational_cases': tested, 'integer_block_carbon': True, 'scope': 'Prescribed-time stoichiometric approximation only; no atom map or kinetics inferred.'}
    checks['negative_tests'] = rejection_tests(catalog, products)
    checks['provenance_scope'] = {'chemical_reachability': False, 'spatial_pore_realization': False, 'stereoselective_path': False, 'gene_directed_synthesis': False, 'incoming_stock_manufacture_in_scope': False, 'nonempty_stock_graph_and_allocation': True, 'finite_state_requires_bounded_total_universe': True}
    hashes = {p.name: hashlib.sha256(p.read_bytes()).hexdigest() for p in sorted(DATA.glob('*.json'))}
    report = {'status': 'PASS', 'checks': checks, 'data_sha256': hashes}
    args.output.parent.mkdir(parents=True, exist_ok=True)
    args.output.write_text(json.dumps(report, indent=2) + '\n')
    print(json.dumps(report, indent=2))
if __name__ == '__main__':
    main()
```

auditors/verify__product__guards.py

```
# Publication adapter: fixed supplied checker, with asserts expanded to
# explicit branches (effective under -O), and configurable output directory.
"""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 os
import copy
import json
ROOT = Path(__file__).resolve().parents[1]
RESULTS = Path(os.environ.get('A77_RESULTS_DIR', ROOT / 'results'))
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')
    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 / 'data/article75_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']}
    (RESULTS / '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()
```

auditors/verify__product__identities.py

```
# Publication adapter: fixed supplied checker, with asserts expanded to
# explicit branches (effective under -O), and configurable output directory.
"""Independent target constitution check against the manuscript's explicit table."""
import json, hashlib, collections
from pathlib import Path
import os
P = Path(__file__).resolve().parents[1]
RESULTS = Path(os.environ.get('A77_RESULTS_DIR', P / 'results'))
f = P / 'data/article75_product_certificates.json'
data = json.loads(f.read_text())

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

def emap(g):
    return {edge(a, b): o for a, b, o in g['bonds']}

def component(g, root):
    adj = {a: set() for a in g['atoms']}
    for a, b, o in g['bonds']:
        adj[a].add(b)
        adj[b].add(a)
    seen = {root}
    stack = [root]
    while stack:
        for v in adj[stack.pop()] - seen:
            seen.add(v)
            stack.append(v)
    return seen
base = [('Ca', 'Cc'), ('Ca', 'Na'), ('Cc', 'Ox'), ('Cc', 'On0')]
side = {'Ala': [], 'Arg': [('5', '4b'), ('4b', '10b'), ('10b', 'Nn1'), ('Nn1', '11'), ('11', 'Nn2'), ('11', 'Nn3')
    ], 'Asn': [('5', '4b'), ('4b', 'On1'), ('4b', 'Nn1')], 'Asp': [('5', '4b'), ('4b', 'On1'), ('4b', 'On2')], '
    Cys': [('5', 'S1')], 'Gln': [('5', '4b'), ('4b', '10b'), ('10b', 'On1'), ('10b', 'Nn1')], 'Glu': [('5', '4b'),
     ('4b', '4a'), ('4a', 'On1'), ('4a', 'On2')], 'Gly': [], 'His': [('5', '4b'), ('4b', '10b'), ('10b', 'Nn1'),
    ('Nn1', '11'), ('11', 'Nn2'), ('Nn2', '4b')], 'Ile': [('5', '6'), ('5', '4b'), ('4b', '10b')], 'Leu': [('5', '
    4b'), ('4b', '4a'), ('4b', '10b')], 'Lys': [('5', '4b'), ('4b', '10b'), ('10b', '11'), ('11', 'Nn1')], 'Met':
     [('5', '4b'), ('4b', 'S1'), ('S1', '10b')], 'Phe': [('5', '4b'), ('4b', '10b'), ('10b', '11'), ('11', '12'),
     ('12', '12a'), ('12a', '4a'), ('4a', '4b')], 'Pro': [('5', '4b'), ('4b', '10b'), ('10b', 'Na')], 'Ser': [('5'
    , 'On1')], 'Thr': [('5', 'On1'), ('5', '4b')], 'Trp': [('5', '4b'), ('4b', '10b'), ('10b', 'Nn1'), ('Nn1', '
    12a'), ('12a', '4a'), ('4a', '4b'), ('12a', '1'), ('1', '2'), ('2', '3'), ('3', '4'), ('4', '4a')], 'Val': [('
    5', '6'), ('5', '4b')]}
side['Tyr'] = side['Phe'] + [('12', 'O12')]
formulas = {'Ala': (3, 7, 1, 2, 0), 'Arg': (6, 14, 4, 2, 0), 'Asn': (4, 8, 2, 3, 0), 'Asp': (4, 7, 1, 4, 0), 'Cys':
     (3, 7, 1, 2, 1), 'Gln': (5, 10, 2, 3, 0), 'Glu': (5, 9, 1, 4, 0), 'Gly': (2, 5, 1, 2, 0), 'His': (6, 9, 3, 2,
     0), 'Ile': (6, 13, 1, 2, 0), 'Leu': (6, 13, 1, 2, 0), 'Lys': (6, 14, 2, 2, 0), 'Met': (5, 11, 1, 2, 1), 'Phe'
    : (9, 11, 1, 2, 0), 'Pro': (5, 9, 1, 2, 0), 'Ser': (3, 7, 1, 3, 0), 'Thr': (4, 9, 1, 3, 0), 'Trp': (11, 12, 2,
     2, 0), 'Tyr': (9, 11, 1, 3, 0), 'Val': (5, 11, 1, 2, 0)}
report = {}
for name, row in data['rows'].items():
    g = row['endpoint']
    atoms = g['atoms']
    comp = component(g, 'Ca')
    heavy = {edge(a, b) for a, b, o in g['bonds'] if a in comp and b in comp and (atoms[a] != 'H') and (atoms[b]
     != 'H')}
    expected = {edge(a, b) for a, b in base + side[name] + ([] if name == 'Gly' else [('Ca', '5')])}
    if not heavy == expected:
        raise AssertionError((name, heavy ^ expected))
    formula = collections.Counter((atoms[a] for a in comp))
    if not tuple((formula[k] for k in ['C', 'H', 'N', 'O', 'S'])) == formulas[name]:
        raise AssertionError((name, formula))
    if not emap(g)[edge('Cc', 'Ox')] == 2:
        raise AssertionError('Certificate predicate failed at original line 44')
    report[name] = {'formula': dict(sorted(formula.items())), 'heavy_adjacency_matches_table': True}
if not sum((v[0] for v in formulas.values())) == 107:
    raise AssertionError('Certificate predicate failed at original line 46')
pair = data['paired_sulfur']
b = emap(pair['initial'])
bridge = {edge('X3:5', 'X3:S1'), edge('X3:S1', 'X4:S1'), edge('X4:S1', 'X4:4b')}
bridge_seen = []
for i, (kind, a, c) in enumerate(pair['rewrite_steps'], 1):
    e = edge(a, c)
```

```
    b[e] = b.get(e, 0) + (1 if kind == 'join' else -1)
    if not b[e]:
        del b[e]
    if bridge <= b.keys():
        bridge_seen.append(i)
if not (bridge_seen and 8 in bridge_seen):
    raise AssertionError('Certificate predicate failed at original line 53')
for prefix, name in [('X3:', 'Cys'), ('X4:', 'Met')]:
    g = pair['endpoint']
    comp = component(g, prefix + 'Ca')
    ats = g['atoms']
    edges = {edge(a[len(prefix):], b[len(prefix):]) for a, b, o in g['bonds'] if a in comp and b in comp and (ats[
     a] != 'H') and (ats[b] != 'H')}
    expected = {edge(a, b) for a, b in base + side[name] + [('Ca', '5')]}
    if not edges == expected:
        raise AssertionError('Certificate predicate failed at original line 58')
    if not not any((a.startswith('X4:' if prefix == 'X3:' else 'X3:') and ats[a] == 'C' for a in comp)):
        raise AssertionError('Certificate predicate failed at original line 59')
summary = {'status': 'PASS', 'scope': 'Constitutional heavy adjacency and neutral formulas only; no stereochemical
      or chemical mechanism assertion.', 'input_sha256': hashlib.sha256(f.read_bytes()).hexdigest(), 'targets':
     report, 'product_carbon_total': 107, 'paired_ordered_bridge_after_steps': bridge_seen, '
     paired_terminal_Cys_Met_independent_identity_check': True}
(RESULTS / 'product_identity_checks.json').write_text(json.dumps(summary, indent=2) + '\n')
print('PASS: 20 table identities, neutral formulas, 107-carbon total and paired sulfur bridge/endpoints
     independently agree.')
```