Executable Certification

Standalone Verification-Code Dossier

Charles D. Schaper, Ph.D.
Chief Editor, Annals of the Chrysene Formalism
editor@chrysene.com

Companion to Article 101, "Executable Certification: Verification Code and Machine-Checked Proof as Accretive Layers in Orthogonal Review." The article occupies pages 1955–1965 in Volume 1, Number 5 (November 2026). This supplement has its own page numbering.

1 Scope and interpretation

The domains are an explicit editorial policy. The finite [A] category is assigned bounded exhaustive evaluation; the universal-discrete [A] category is assigned symbolic proof. Arithmetic [D] statements are checked relative to stated constants. The tags are supplied data, not an automatic classifier of arbitrary prose. The non-nesting result is conditional on this declared policy and its two witness classes. It is not a computability separation between programming languages or proof assistants.

The checks certify the formulas and finite objects represented below. They do not establish that a manuscript's extraction or classification is faithful. They do not certify molecular realization, chemical feasibility, statistical independence, or empirical truth.

Historical claims about Articles 41, 62 and 64 remain source reports: their reported audit counts, authorship independence, past compilation failures, 108-check harness, 23-theorem artifact and mutation outcomes are not rerun here. The pool ratio near 66, the 143-second replication threshold, the membrane tilt window and depth/footprint values require source constants, models or evidence not contained in this editorial. Numerical comparisons below verify only the explicitly stated arithmetic, not those derivations. The perimeter model is the abstract cycle on 18 numbered vertices; these vertex IDs are not a molecular locant assignment or a full chrysene graph.

2 Verification coverage

The baseline executes 5,905 assertions, all passing:

- 5,120 tests of lifted-domain conjunction and empty-domain vacuity, across all 64 subsets of six policy kinds and all 64 verdict assignments;
- 8 domain membership, overlap and non-nesting checks;
- 112 gate checks across all 32 five-gate verdict assignments;
- 496 perimeter checks: 18 single cuts, all 153 two-edge cuts, connectivity, path degrees and component-size/edge-count conditions;
- 33 sampled capacity identities, for $m = 0, \ldots, 32$;
- 128 exact-rational checks of linear dissipation scaling under $\omega = kv$, with positive normalized parameters;
- 8 arithmetic witnesses covering phase/period, angles, tether subtraction and reported coverage fractions.

Python uses arbitrary-precision integers and exact fractions throughout. The capacity sampling does not prove the all-natural statement; the Lean companion supplies that theorem. The energy checks use normalized parameters to test the stated algebra, not measured physical constants.

3 Execution and mutation sensitivity

Run `python3 verification_A101.py`. A failed predicate produces a nonzero process exit status. The source collects Boolean checks and enforces their conjunction through its exit code; it does not rely on Python's optional `assert` statements for the manuscript checks. Every mutation begins from the passing baseline and changes one declared input or model coefficient while keeping the claim to be checked fixed. A mutation pass is not accepted as evidence of sensitivity. To reproduce a mutation, append `-mutation NAME`. To rerun this layer and its mutation tests, use `python3 run_python_checks.py`. The formal companion supplies `run_lean_checks.py`, taking the path to the Lean executable.

Eight expected-invalid Python mutations and five Lean mutations are included. Their exact failure counts are shown in the runner transcript below; individual raw logs are included in the package. Sensitivity to these selected mutations is not completeness against all possible errors.

```
Lean (version 4.24.0, x86_64-unknown-linux-gnu, commit 797c613eb9b6d4ec95db23e3e00af9ac6657f24b, Release)
Baseline Python: PASS; baseline Lean: PASS; audited theorems: 31
Python pair_count: 1 failed checks/theorems; exit 1
Python half_period: 1 failed checks/theorems; exit 1
Python bearing_sign: 2 failed checks/theorems; exit 1
Python tether_extension: 1 failed checks/theorems; exit 1
Python coverage_numerator: 1 failed checks/theorems; exit 1
Python unqualified_A_domain: 2 failed checks/theorems; exit 1
Python capacity_base: 32 failed checks/theorems; exit 1
Python energy_square: 96 failed checks/theorems; exit 1
Lean unqualified_A_domain: 3 failed checks/theorems; exit 1
Lean capacity_base: 1 failed checks/theorems; exit 1
Lean cycle_vertices: 2 failed checks/theorems; exit 1
Lean half_period: 1 failed checks/theorems; exit 1
Lean tether_extension: 1 failed checks/theorems; exit 1
All 13 expected-invalid mutations rejected. Exit status: 0
```

4 Exact checked source

```python
#!/usr/bin/env python3
"""Article 101: scoped, dependency-free reproducible verification.
Uses only the Python standard library. Checks identities and the declared
editorial domain policy, not historical audit events or physical realization.
"""
from fractions import Fraction as F
from itertools import combinations, product
from collections import Counter
import argparse
import sys

KINDS = ('finite_A', 'universal_A', 'arithmetic_D', 'other_D', 'C', 'E')
BASE = dict(nodes=18, pair_claim=153, active_hours=12, duty=F(1, 2),
            period_claim=86400, bearing=F(1339, 10), acute_claim=F(461, 10),
            contour=F(114, 10), extension=F(38, 10), loss_claim=F(76, 10),
            explicit=48, total=63, combined=55, rounded_percent=F(873, 10),
            broad_V=False, power_base=4, energy_power=1)
MUTATIONS = {
    'pair_count': dict(pair_claim=154),
    'half_period': dict(period_claim=43200),
    'bearing_sign': dict(bearing=-F(1339, 10)),
    'tether_extension': dict(extension=F(48, 10)),
    'coverage_numerator': dict(combined=54),
    'unqualified_A_domain': dict(broad_V=True),
    'capacity_base': dict(power_base=3),
    'energy_square': dict(energy_power=2),
}

def dom_l(kind):
    return kind in ('finite_A', 'universal_A')

def dom_v(kind, data):
    return kind in ('finite_A', 'arithmetic_D') or (
        data['broad_V'] and kind == 'universal_A')

def components(n, removed):
    adj = [set() for _ in range(n)]
    for i in range(n):
        if i not in removed:
            j = (i + 1) % n
            adj[i].add(j)
            adj[j].add(i)
    unseen = set(range(n))
    parts = []
    while unseen:
        seed = min(unseen)
        seen, todo = {seed}, [seed]
        while todo:
            for j in adj[todo.pop()]:
                if j not in seen:
                    seen.add(j)
                    todo.append(j)
        unseen -= seen
        parts.append(seen)
    return parts, adj

def run(data):
    checks = []
    def check(group, label, truth):
        checks.append((group, label, bool(truth)))
    # Every finite model manuscript, plus all pass/fail assignments on its claims.
    for kind in KINDS:
        check('domains', 'intersection_' + kind,
              (dom_v(kind, data) and dom_l(kind)) == (kind == 'finite_A'))
    check('domains', 'L_not_subset_V', any(dom_l(k) and not dom_v(k, data) for k in KINDS))
    check('domains', 'V_not_subset_L', any(dom_v(k, data) and not dom_l(k) for k in KINDS))
    for mask in product((False, True), repeat=6):
        manuscript = [k for k, present in zip(KINDS, mask) if present]
        for verdicts in product((False, True), repeat=6):
            passed = dict(zip(KINDS, verdicts))
            lifted_v = all(passed[k] for k in manuscript if dom_v(k, data))
            lifted_l = all(passed[k] for k in manuscript if dom_l(k))
            direct = all(passed[k] for k in manuscript if dom_v(k, data) or dom_l(k))
            check('lift', 'conjunction_equivalence', (lifted_v and lifted_l) == direct)
            if not any(dom_v(k, data) for k in manuscript):
                check('lift', 'empty_domain_vacuity', lifted_v)
    for gates in product((False, True), repeat=5):
        extended = all(gates)
        check('gates', 'non_substitution', not extended or all(gates[:3]))
```

```
        for i in range(5):
            if not gates[i]:
                check('gates', 'failed_gate_rejects', not extended)
    n = data['nodes']
    for i in range(n):
        parts, adj = components(n, {i})
        degrees = sorted(map(len, adj))
        check('cycle', 'single_connected', len(parts) == 1)
        check('cycle', 'single_path_degrees', degrees == [1, 1] + [2] * (n - 2))
    pairs = list(combinations(range(n), 2))
    check('cycle', 'pair_count', len(pairs) == data['pair_claim'])
    for i, j in pairs:
        parts, adj = components(n, {i, j})
        check('cycle', 'two_components', len(parts) == 2)
        check('cycle', 'two_partition_sizes', sorted(map(len, parts)) == sorted([j-i, n-j+i]))
        check('cycle', 'two_paths_or_singletons', all(sum(len(adj[v]) for v in p) == 2*(len(p)-1) for p in parts))
    # Finite tests supplement, and do not replace, the all-m Lean theorem.
    for m in range(33):
        check('capacity', 'm_' + str(m), data['power_base']**m == 2**m * 2**m)
    # Normalize fixed factors to positive rational values; omega = k*v.
    for N, zeta, k, r, length in product((F(1), F(2)), repeat=5):
        for v in (F(1, 10), F(1), F(10), F(100)):
            energy = N*zeta*(k*v*r)**2*(length/v)
            proposed = N*zeta*(k*r)**2*length*v**data['energy_power']
            check('scaling', 'linear_energy', energy == proposed)
    active = data['active_hours'] * 3600
    check('arithmetic', 'active_seconds', active == 43200)
    check('arithmetic', 'full_period', F(active)/data['duty'] == data['period_claim'])
    check('arithmetic', 'supplementary_angle', 180-data['bearing'] == data['acute_claim'])
    check('arithmetic', 'perpendicular_bearing', data['bearing']-90 == F(439, 10))
    check('arithmetic', 'window_width', 90-46 == 44)
    check('arithmetic', 'tether_subtraction', data['contour']-data['extension'] == data['loss_claim'])
    check('arithmetic', 'coverage_rounding', abs(F(100*data['combined'], data['total'])-data['rounded_percent']) < F(1, 20))
    check('arithmetic', 'explicit_coverage', F(data['explicit'], data['total']) == F(16, 21))
    return checks

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--mutation', choices=MUTATIONS)
    args = parser.parse_args()
    data = dict(BASE)
    if args.mutation:
        data.update(MUTATIONS[args.mutation])
    checks = run(data)
    failed = [row for row in checks if not row[2]]
    print('Article 101 verification | Python ' + sys.version.split()[0])
    print('Run: ' + (args.mutation or 'baseline'))
    print('Checks by group: ' + ', '.join(f'{g}={n}' for g, n in sorted(Counter(x[0] for x in checks).items())))
    print(f'Checks: {len(checks)}; passed: {len(checks)-len(failed)}; failed: {len(failed)}')
    for group, label, _ in failed:
        print('FAIL ' + group + '/' + label)
    print('Exit status: ' + ('1' if failed else '0'))
    return 1 if failed else 0

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

5 Verbatim baseline output

```
Article 101 verification | Python 3.12.14
Run: baseline
Checks by group: arithmetic=8, capacity=33, cycle=496, domains=8, gates=112, lift=5120, scaling=128
Checks: 5905; passed: 5905; failed: 0
Exit status: 0
```

Baseline process exit status: 0.

6 Reproducibility and byte identity

The source block above is inserted programmatically from the same UTF-8 file that produced the log. Extraction of the block is compared byte-for-byte with that file during the dossier build. The `extract_dossiers.py` utility reproduces that check. Compile this document directly with pdfLaTeX; no local font map or companion source file is needed for its LaTeX build. The executable source and logs are also distributed as plain files for convenience.

Source SHA-256:

```
4b56f5ad4adb7fa5e440e901666d064e092cf097e9f48f09191686da416ee66d
```