ContinuumPort

A Protocol for Enforced Execution Validity

Gh. Rotaru (Giorgio Roth) · Independent Researcher
Version 2.0 · Status: Preprint · April 2026
https://continuumport.com/

ABSTRACT

Most systems can execute. Very few can prove that execution remains valid over time. ContinuumPort defines a structural protocol for execution continuity and enforces that all state transitions preserve declared invariants — across models, sessions, and environments. It does not decide what is correct. It makes invalid execution impossible. Regen Engine is the reference implementation. 651 tests passed. 0 invariant violations.

1. Problem

Execution in distributed and multi-model systems is not inherently stable. Failure does not occur because systems are incorrect at initialization. It occurs because they cannot guarantee that execution remains valid across time, environment change, or model substitution.

Common failure modes observed in practice:

- **Partial state application** — a transition commits partially, leaving the system in an inconsistent intermediate state.
- **Non-deterministic outcomes** — identical inputs produce divergent results across executions or environments.
- **Hidden state mutation** — side effects occur outside declared control boundaries. • **Simulation/execution divergence** — a transition that appears valid in simulation produces different behaviour on commit.

These are not edge cases. They are systemic properties of unconstrained execution.

2. Why Existing Approaches Fail

Current approaches to execution safety rely on behavioural validation, post-hoc monitoring, and policy-based guardrails. These methods detect issues after execution has occurred. They depend on interpretation. They can be bypassed. They observe invalid execution — they do not prevent it. The distinction is structural, not technical.

3. Core Principle

Execution must be constrained structurally, not behaviourally. Behavioural validation is insufficient because it evaluates outcomes, not admissibility.

A transition MUST NOT be executed unless it can be proven to preserve all declared invariants. If proof fails, execution is rejected — not logged, not flagged, rejected.

4. Execution Model

ContinuumPort defines execution as a sequence of state transitions under declared constraints. Every transition passes through two mandatory phases.

build()

Constructs a PreparedRequest: a fully declared, inspectable description of the intended transition. No execution occurs at this stage. The request is a value, not an action.

commit()

Evaluates the PreparedRequest against all declared invariants before any state change is applied. If any invariant is violated, commit is rejected and no partial effect persists.

This separation is the enforcement mechanism. There is no implicit execution path. There is no shortcut from intent to effect. All execution MUST pass through this separation. No execution path exists outside this two-phase model.

5. System State

A persistent execution system is represented as:

```
Σ = (D, A, Auth)
```

• **D** — Declarative state: explicit knowledge relevant to current tasks.

• **A** — Adaptive context: information accumulated through interaction or environment. • **Auth** — Execution authority: the AuthorityContext that determines which transitions are permitted.

Auth is not optional and not implicit. Every transition is evaluated against a declared AuthorityContext. The system has no fallback that permits execution outside this model.

The output of a valid transition is a StateCapsule:

```
C = (I, K, S, A_next)
```

Where I is intent, K is known state, S is the semantic result, and A_next is the authority context for the next transition.

6. Invariants

Every transition must satisfy four invariants before commit is permitted. These invariants are mandatory and non-negotiable.

Atomicity

The transition either commits fully or does not commit at all. No partial application persists.

Determinism

Identical inputs under identical authority context produce identical outputs. Non-deterministic execution is a violation, not a warning.

Isolation

No state mutation occurs outside the declared boundary of the transition. Hidden side effects are structurally impossible, not merely discouraged.

Semantic Alignment

Simulation and execution are equivalent:

```
simulate(state, action) ≡ execute(state, action)
```

Divergence between what a transition is declared to do and what it does on commit is an invariant violation. Execution halts.

7. Authority Model

Execution authority is explicit. There is no ambient permission.

Every transition is evaluated against an AuthorityContext that declares what the transition is permitted to do. The system distinguishes between:

• **A_untrusted** — the declared intent of the caller, before validation.

• **A'** — the validated authority context under which execution is permitted.

No execution path exists that bypasses this validation. This is formally demonstrated through adversarial test suites designed to find any such path. None was found.

The TOCTOU (time-of-check/time-of-use) problem is addressed at the architectural level: the PreparedRequest captures state at check time, and commit operates on that captured state — not on re-queried live state.

8. Enforcement in Practice

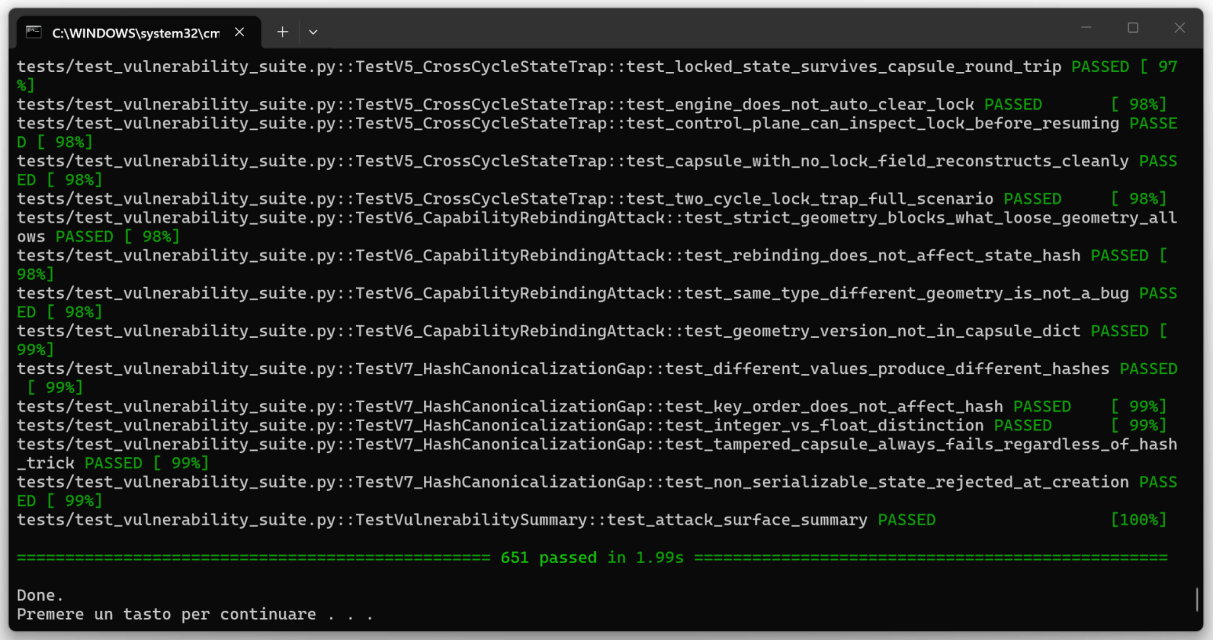
The enforcement model was validated adversarially. Test implementations were constructed that appear structurally correct but violate individual invariants. Each class of violation was introduced deliberately. Failures are mandatory and cannot be bypassed.

Violation	Description
Atomicity	Partial state persists after a failed commit

Non-determinism	Transition output varies across identical inputs
Snapshot isolation breach	Commit operates on re-queried state, not captured state
Simulation / execution mismatch	Declared behaviour diverges from committed behaviour

All violations were detected. None resulted in a committed invalid state.

651 tests passed · 0 invariant violations



9. Verification Model

Validation is invariant-based, not behaviour-based.

Behavioural testing asks whether the system does what is expected. Invariant-based testing asks whether the system is capable of violating a structural property — and then proves that it is not. A system that passes behavioural tests but cannot pass invariant-based adversarial testing has not demonstrated execution validity.

The tests are the arbiter. Not the implementation. No alternative validation method is considered sufficient.

10. What This Makes Impossible

Under the ContinuumPort enforcement model, the following are structurally impossible — not runtime-checked, structurally impossible:

- partial execution
- non-deterministic outcomes
- state mutation outside declared authority
- simulation / execution divergence
- invalid transitions committing

These properties hold across models, sessions, and environments because they are properties of the execution structure, not of any specific implementation.

11. Scope

ContinuumPort enforces correctness of execution under declared

constraints. It does not guarantee:

- correctness of intent
- correctness of the declared constraints themselves
- consistency of external side effects not under system control

Undeclared risks are not blocked. The protocol enforces what has been declared. What has not been declared is outside its scope.

12. What ContinuumPort Is Not

Not a library. Not a policy layer. Not a monitoring system. Not a behavioural guardrail. It is an

execution validity protocol. The distinction is that it operates before execution, not after it.

13. Access

The public repository demonstrates enforcement behaviour. The compliance layer — adapter interface, adversarial suite, and conformance tests — is required to verify external systems and is available upon request.

If your system executes but cannot prove validity, it will fail under ContinuumPort.

access@continuumport.com

Conclusion

Execution without proof of validity leads to drift, inconsistency, and undetectable failure.
ContinuumPort defines a different model: execution is permitted only when validity is preserved.

The system does not assume. It enforces.

Nothing outside the invariants is allowed to execute. Execution without proof is not restricted. It is disallowed.

Repository: https://github.com/giorgioroth/ContinuumPort · Site: https://continuumport.com/