ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Formal Verification and Deterministic State Machine Standard

Executable specifications, invariant verification, deterministic transitions, model checking, and traceable implementation refinement.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0007

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Formal Verification and Deterministic State Machine Standard

DOCUMENT ID
MYTHOS-SWE-0007

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

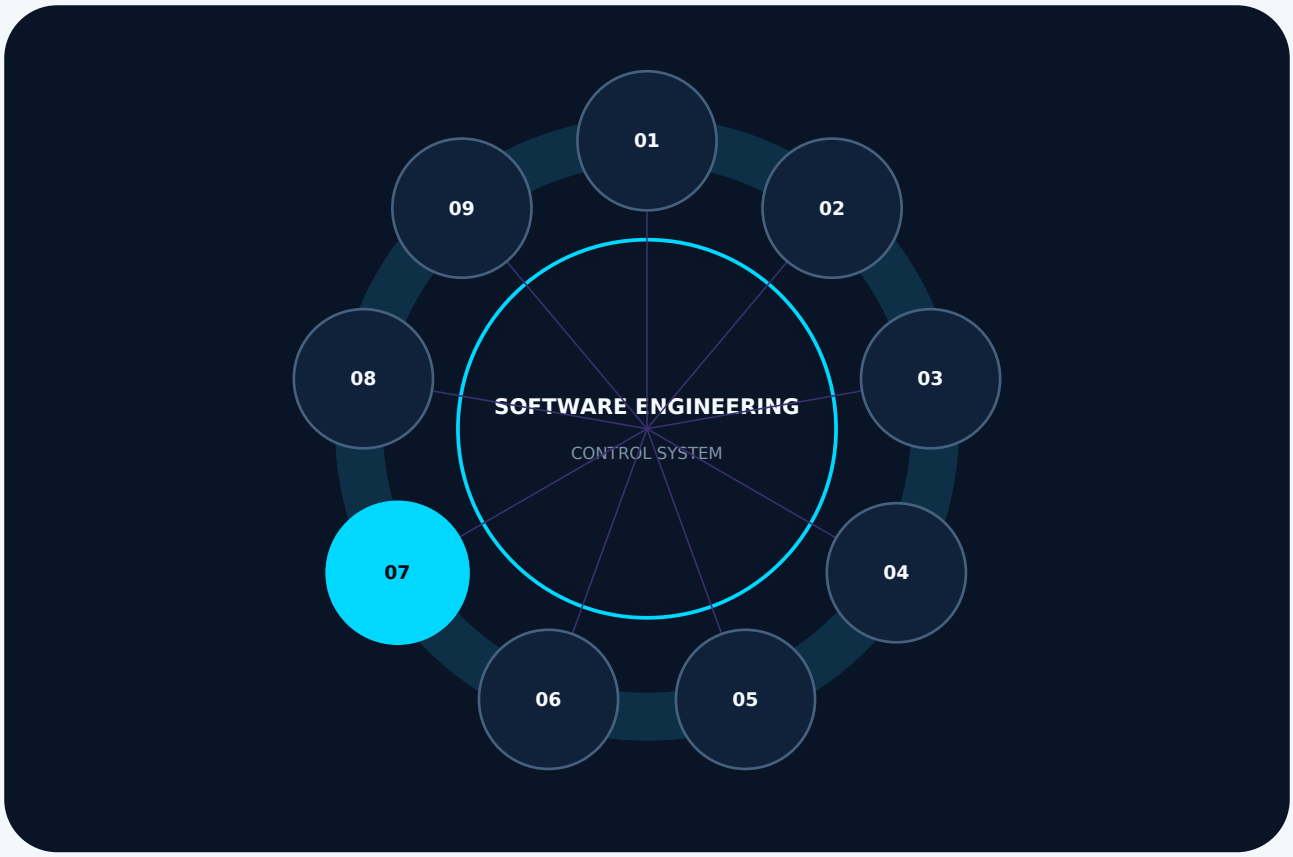
SCHEDULED REVIEW
2027-01-24

16	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0007 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Formal Verification and Deterministic State Machine Standard

The validation of critical state machines has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

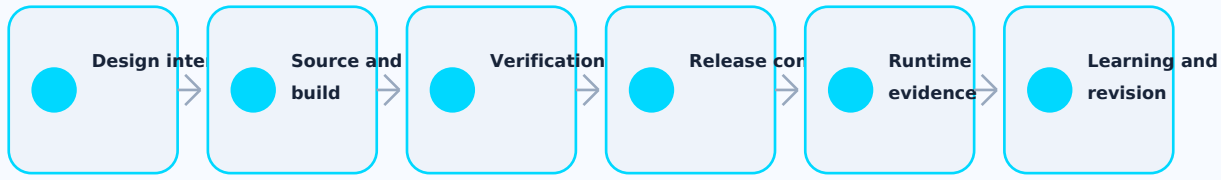
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Safety and Invariant Verification (Weight: 25%)	1
Liveness and Temporal Logic (Weight: 20%)	2
Specification-to-Code Alignment (Weight: 20%)	2
State Machine Architecture (Weight: 15%)	2
Tooling and CI Integration (Weight: 10%)	1
Operational Lifecycle (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	5

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Safety and Invariant Verification (Weight: 25%)	Deadlock Freedom
2	Liveness and Temporal Logic (Weight: 20%)	Termination Guarantees
3	Liveness and Temporal Logic (Weight: 20%)	Fairness and Starvation
4	Specification-to-Code Alignment (Weight: 20%)	Refinement Mapping
5	Specification-to-Code Alignment (Weight: 20%)	Conformance Testing
6	State Machine Architecture (Weight: 15%)	Determinism
7	State Machine Architecture (Weight: 15%)	State Space Bounding
8	Tooling and CI Integration (Weight: 10%)	Specification in CI
9	Operational Lifecycle (Weight: 10%)	Specification Maintenance
10	Suite Composition	MFVS-Safety
11	Suite Composition	MFVS-Liveness
12	Additional Evaluation Dimension	Verification Dimensions
13	Additional Evaluation Dimension	The Verification Doctrine
14	Additional Evaluation Dimension	Scoring Model
15	Additional Evaluation Dimension	Suite Composition
16	Additional Evaluation Dimension	Execution Protocol

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The validation of critical state machines has failed.

Organizations rely on unit tests, integration tests, and property-based testing to verify the correctness of distributed systems and agentic execution planes. Testing, however, is sampling. It proves the presence of bugs, not their absence. For critical state machines—where a deadlock stalls a million-dollar GPU training job, or a livelock drains a cloud budget—testing is fundamentally insufficient.

This standard exists because:

1. Testing is Not Proof: A test suite verifies a specific execution path. Formal verification (TLA+, Apalache) proves mathematically that *no* execution path violates an invariant.
2. Concurrency Bugs are Invisible to Tests: Race conditions, deadlocks, and livelocks are Heisenbugs that rarely reproduce in test environments. Formal verification exhaustively explores the state space, proving deadlock freedom.
3. State Machines are the Core of Agents: An AI agent is a state machine. If the state machine allows invalid transitions (e.g., "Executing" to "Completed" without "Verifying"), the agent will eventually take that invalid path in production.
4. Determinism is Not Optional: In governed AI and infrastructure, the same inputs MUST produce the same state transitions. Non-deterministic state machines are untestable, unverifiable, and ungovernable.

This document establishes the Mythos Formal Verification Standard (MFVS), a comprehensive, evidence-based methodology for evaluating the use of formal specification, model checking, and deterministic state machine design in production systems.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Formal specification languages (TLA+, PlusCal, Alloy)
1. Model checkers (TLC, Apalache, NuSMV)
1. Safety properties (invariants, deadlock freedom, state constraints)

1. Liveness properties (termination, fairness, eventual consistency)
1. Deterministic state machine design (finite state machines, hierarchical state machines)
1. Refinement mapping (specification to implementation alignment)
1. State space optimization and bounded model checking

Out of Scope

1. General software design principles (covered in MYTHOS-SWE-0001)
1. Specific language engineering (covered in MYTHOS-SWE-0002 through 0004)
1. General testing methodologies (unit, integration, property-based)

Audience

1. Principal engineers and architects designing critical state machines
1. Security teams evaluating invariant enforcement
1. Platform engineers building agentic execution planes
1. AI governance, risk, and compliance teams
1. Standards bodies seeking a reference formal verification methodology

Definitions and Taxonomy

Verification Dimensions

Dimension	Definition
Safety Property	A property that asserts "nothing bad ever happens." E.g., the system never enters an invalid state, two agents never hold the same exclusive lock.
Liveness Property	A property that asserts "something good eventually happens." E.g., the system eventually terminates, a requested resource is eventually granted.
Invariant	A predicate that must be true in every reachable state of the system.
Model Checking	An automated technique for verifying finite-state concurrent systems by exhaustively exploring the state space.
Refinement Mapping	A formal proof that an implementation (low-level specification) correctly implements an abstract specification (high-level specification).
State Explosion	The exponential growth of the state space that makes exhaustive model checking computationally intractable.

The Verification Doctrine

> Testing shows the presence of bugs. Formal verification proves their absence. A deadlock is an outage. A livelock is a budget fire. If it is critical, it must be proven.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; no formal verification, ad-hoc state machines
1	Basic state diagrams but no formal specification
2	Functional TLA+ specs but not checked in CI or unmaintained
3	Solid production performance; TLA+/Apalache in CI, safety properties proven
4	Strong performance; liveness proven, refinement mapping, bounded state
5	Best-in-class; sets the standard for mathematically proven systems

Weighting

Category	Weight	Rationale
Safety & Invariant Verification	25%	Invalid states cause outages and breaches
Liveness & Temporal Logic	20%	Livelocks and starvation cause resource exhaustion
Specification-to-Code Alignment	20%	An unimplemented spec is a fiction
State Machine Architecture	15%	Non-determinism makes verification impossible
Tooling & CI Integration	10%	Specs must be continuously verified
Operational Lifecycle	10%	Specs must evolve with the system

Total: 100%

A system MUST score at least 3.0 in Safety and Specification-to-Code Alignment to be considered for production use in critical paths.

Evaluation Categories

Safety and Invariant Verification (Weight: 25%)

Invariant Definition

Are critical safety properties formally defined and proven?

Score	Criteria
0	No formal invariants; safety assumed via testing
1	Invariants documented in natural language
2	Invariants defined in TLA+ but not model checked
3	Invariants model checked (TLC/Apalache) for bounded state space
4	Invariants proven for unbounded state space using inductive invariants

Score	Criteria
5	Perfect safety: formally proven inductive invariants, zero invariant violations possible, mathematical proof of deadlock freedom

Deadlock Freedom

Is the system mathematically proven to be deadlock-free?

Score	Criteria
0	Deadlocks discovered in production
1	Deadlocks tested via chaos engineering
2	TLA+ model checked for deadlocks on small state space
3	Apalache bounded model checking for deadlocks
4	Proof of deadlock freedom via inductive invariant
5	Perfect proof: formally verified deadlock freedom, zero circular waits, mathematically proven resource allocation

---

Liveness and Temporal Logic (Weight: 20%)

Termination Guarantees

Can the system prove that critical operations eventually terminate?

Score	Criteria
0	No termination guarantees; processes hang indefinitely
1	Timeouts used as a proxy for liveness
2	Liveness properties defined in TLA+ (e.g., `<>[]Terminated`)
3	Liveness properties model checked with fairness assumptions
4	Proven termination under weak fairness
5	Perfect liveness: formally proven termination, zero starvation, proven progress under adversarial conditions

Fairness and Starvation

Does the system guarantee that all processes make progress?

Score	Criteria
0	No fairness guarantees; starvation observed
1	Round-robin scheduling in implementation (not proven)
2	Strong fairness assumed but not proven
3	Weak fairness proven in specification
4	Strong fairness proven for critical resources
5	Perfect fairness: formally proven no starvation, equitable resource distribution mathematically guaranteed

---

Specification-to-Code Alignment (Weight: 20%)

Refinement Mapping

Is there a formal mapping between the abstract specification and the implementation?

Score	Criteria
0	Specification and implementation unrelated
1	Specification written after implementation as documentation
2	Specification drives design, but no formal mapping
3	Informal refinement mapping documented
4	Formal refinement mapping defined; implementation proven to refine specification
5	Perfect alignment: formally verified refinement, automated conformance testing, zero specification-implementation drift

Conformance Testing

Is the implementation continuously tested against the specification?

Score	Criteria
0	No conformance testing
1	Manual verification during code review
2	Property-based tests derived from specification
3	Model-based testing generating tests from TLA+ specs
4	Runtime monitoring verifying implementation matches spec
5	Perfect conformance: continuous runtime verification, automated trace validation, zero unobservable state deviations

State Machine Architecture (Weight: 15%)

Determinism

Are state machines strictly deterministic?

Score	Criteria
0	Non-deterministic state transitions; behavior depends on timing
1	Mostly deterministic but with implicit non-determinism (e.g., hash map iteration)
2	Explicit determinism in design but not enforced
3	Deterministic state transitions enforced by type system (e.g., Rust type-state)
4	Formally verified determinism in specification
5	Perfect determinism: mathematically proven deterministic transitions, zero non-deterministic behavior, reproducible execution

State Space Bounding

Is the state space bounded and manageable?

Score	Criteria
0	Unbounded state space; model checking impossible
1	State space bounded by constants but too large for exhaustive checking
2	State space abstracted for model checking
3	Symmetry reduction and abstraction techniques applied
4	Bounded model checking (Apalache) handles realistic state space
5	Perfect bounding: state space fully explored, formally verified abstraction, zero state explosion

Tooling and CI Integration (Weight: 10%)

Specification in CI

Is formal verification a mandatory gate in the CI/CD pipeline?

Score	Criteria
0	Specifications run locally, if at all
1	Specifications run on commit, but failures are warnings
2	Specifications run in CI; failures block merge
3	Apalache bounded model checking in CI; fast feedback loop
4	Incremental verification; only changed modules re-verified
5	Perfect integration: formally verified gates, cryptographic signing of verification results, zero unverified merges

Operational Lifecycle (Weight: 10%)

Specification Maintenance

Are specifications maintained with the same rigor as code?

Score	Criteria
0	Specifications abandoned after initial design
1	Specifications updated manually, sporadically
2	Specifications version-controlled alongside code
3	Specification changes required for any state machine modification
4	Automated drift detection between spec and implementation
5	Perfect maintenance: formally verified spec-code parity, automated consistency checks, zero specification rot

The Mythos Formal Verification Suite (MFVS)

Traditional benchmarks measure code coverage. The MFVS evaluates mathematical proof, liveness, and specification alignment.

Suite Composition

MFVS-Safety

1. Invariant Violation: 100+ tests injecting adversarial states into the model checker.
1. Deadlock Injection: 50+ tests simulating resource contention to verify deadlock freedom proofs.

MFVS-Liveness

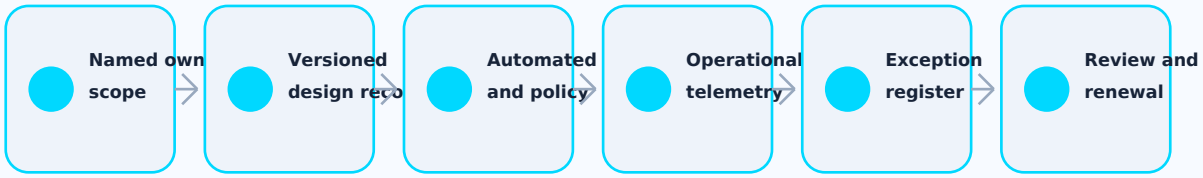
1. Starvation Simulation: 50+ tests simulating high-load to verify fairness properties.
1. Termination: 50+ tests verifying all operations reach a terminal state.

Execution Protocol

ASSURANCE AND ADOPTION

Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

Assurance maturity spectrum

0	1	2	3	4	5
Absent	Ad hoc	Repeatable	Controlled	Measured	Adaptive

Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.