A Hybrid Framework for Low-Overhead Observability and Runtime Verification in Concurrent Multi-Agent Systems

Tiberiu-Alexandru Bădoi, Andrei Olaru
Faculty of Automatic Control and Computers
National University of Science and Technology Politehnica Bucharest

badoitiberiualexandru@gmail.com andrei.olaru@upb.ro

Evolution and Challenges of Modern MAS

Architectural Evolution

Stochastic Reasoning: Transition from rule-based execution toward Deep Reinforcement Learning (DRL) and localized Large Language Models (LLMs).

High Concurrency: Decoupled agent networks executing asynchronously across parallel thread pools for modularity and fault isolation.

Non-Deterministic Behavior: Decisions vary dynamically across runs, complicating reproducibility and traditional deterministic testing routines.

The Quality Assurance Dilemma

Modified Cognitive Agents: Updating neural weights, heuristics, or decision logic requires formal safety invariant re-validation.

Full-Scale Re-simulation Bottleneck: Rerunning all concurrent live entities incurs prohibitive processing time and hardware resource costs.

Static Replay Failure: Pre-recorded historical mock environments desynchronize as soon as an agent deviates (Cognitive Drift).

Core Bottlenecks & Problem Statement (1)

1. Probe Effect

Instrumentation overhead perturbs asynchronous thread schedules, masking or introducing critical concurrency race conditions.

2. Cognitive Drift

Candidate agents making novel decisions cause surrounding static mock environments to fail to adapt, corrupting global context.

3. Telemetry Ingestion

Massive, unstructured multi-agent log streams incur high storage/CPU overhead and resist manual runtime diagnosis.

4. Formal Intractability

Exhaustive state exploration (model checking) is mathematically intractable for large, asynchronous agent clusters.

Research Question (RQ)

How can a concurrent multi-agent system verify the runtime behavioral compliance of an altered cognitive agent against formal protocol specifications within a historical environment, without inducing state desynchronization or incurring full-scale simulation overhead?

Core Bottlenecks & Problem Statement (2)

Research Question (RQ)

How can a concurrent multi-agent system verify the runtime behavioral compliance of an altered cognitive agent against formal protocol specifications within a historical environment, without inducing state desynchronization or incurring full-scale simulation overhead?

- **We want to *replay* most of the agents in the multi-agent system**
- **One agent has its behavior *altered***
- **How can we handled the *divergence* from the recorded behaviour while *live-simulating* only a fraction of the agents, while *mock-simulating* the other agents based on recorded data**

Three-Tier Hybrid Observability & Verification Framework (1)

Layer 1: Observability & Ingestion

Packet Interception: Monitors message waves at the messaging middleware shard layer without invasive bytecode rewrites.

Lock-Free Buffers: Dispatches events asynchronously through lock-free ring buffers (ConcurrentLinkedQueue).

Zero-Cost Facade: Short-circuits telemetry ingestion via a zero-cost no-op facade (NullRecorder) when logging is inactive.

Layer 2: Hybrid Re-Simulation

Unified Virtual Grid: Coordinates dual-mode execution partitions between active live threads and lightweight playback shells.

Cascade Orchestrator: Monitors behavioral deltas against historical logs in real-time.

State Rehydration: Synchronizes and promotes candidate node dependencies on demand via state bridges.

Layer 3: Verification Kernel

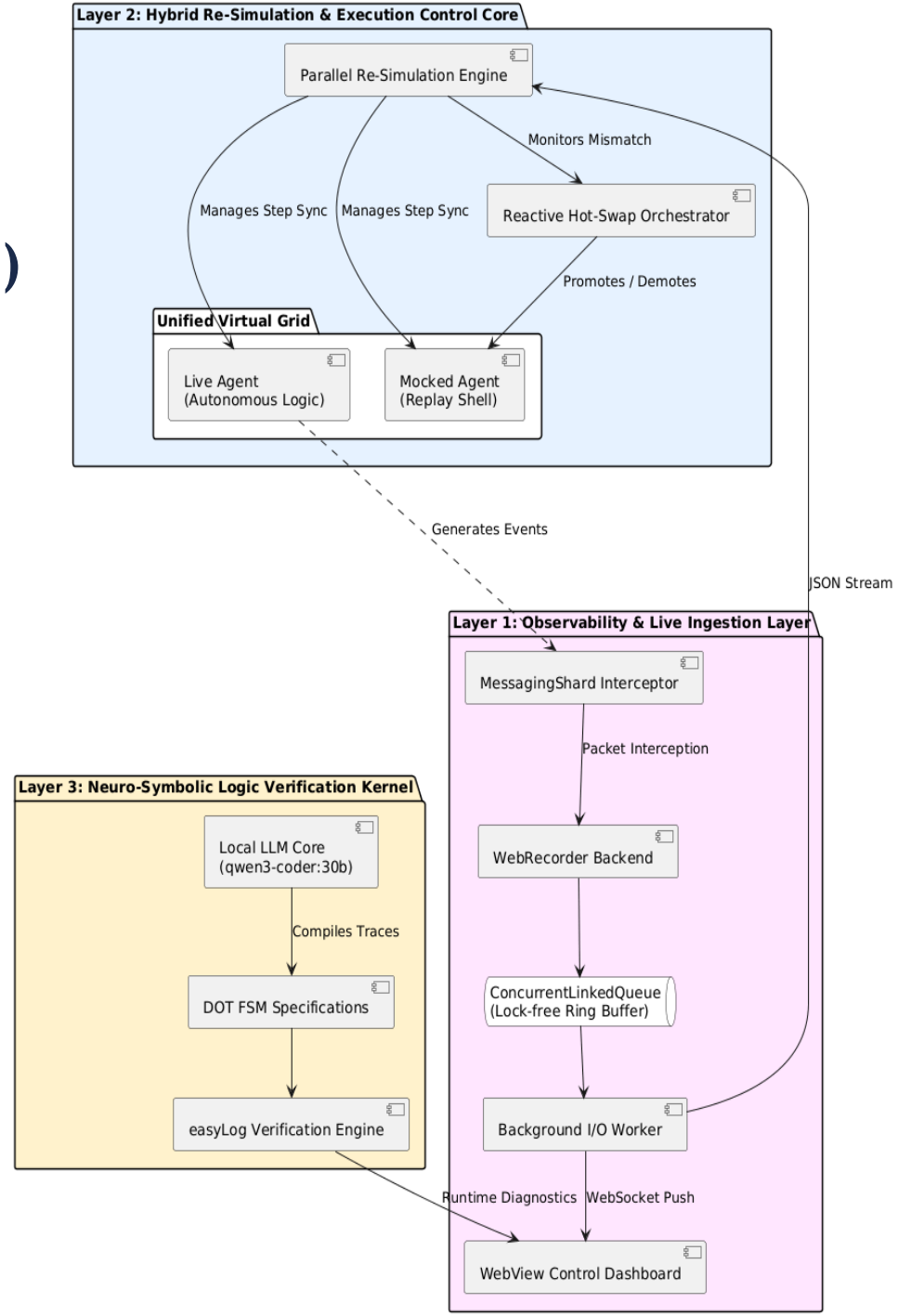
Local LLM Core: Uses localized Qwen3-Coder (30B) to compile raw interaction baseline logs into Graphviz DOT FSM specifications.

Automata Parser: The easyLog engine strictly validates runtime execution traces against formal state machines.

Zero Hallucinations: Separates AI model extraction from deterministic, absolute runtime trace verification.

SYSTEM ARCHITECTURE

Three-Tier Hybrid Observability & Verification Framework (2)



Formal Protocol Compliance Checking

Automata Formalization

Deterministic FSM Model: Formalized as $\mathcal{M} = (S, \Sigma, \delta, s_0, F)$

State Space (S): $S = \{\text{IDLE}, \text{EVALUATING}, \text{BID_SUBMITTED}, \text{MATCHED}, \text{ERROR}\}$

Alphabet (Σ): Valid interaction payloads (bids, asks, price bounds, timestamps).

Transition Function (δ): $\delta: S \times \Sigma \to S$ maps the active state and observed payload to the next state.

Verification Scope: Runtime Verification (RV) over observed live traces rather than exhaustive static model checking.

Safety Invariant Guarantees

Safety Property Guarantee:

$$\forall t, \quad \delta\left(s(t), E_{\text{actual}}(a_{\text{target}}, t)\right) \in S \setminus \{\text{ERROR}\}$$

`Protocol Compliance:` **Guarantees that an altered candidate agent never submits out-of-order bids or violates market clearing rules.**

Boundary Enforcement: Detects bid values exceeding price caps or settlement transactions arriving after scheduled deadlines.

State Containment: Flags protocol deviations at runtime without interrupting the host simulation container.

Mitigating Drift via Adaptive Execution Bubbles

The Cognitive Drift Paradox

Desynchronization Point: Occurs when candidate agent a_target deviates from historical logs while surrounded by static mocks.

Causality Breakdown: Mock agents continue replaying obsolete responses, invalidating the multi-agent decision space.

The Core Dilemma: Full-scale resimulation is computationally intractable; static trace replay produces invalid execution traces.

Execution Bubble Formalization

System Graph: $G = (A, M), \quad A = \{a_1, a_2, \dots, a_n\}, \quad M \subset A \times A$, where A represents agent threads and M represents active messaging channels.

Execution Mode: Each agent has state $\sigma_i(t) \in \{\text{Mocked}, \text{Live}\}$.

Initialization (t = 0): Only the candidate agent is live ($\sigma_{\text{target}}(0) = \text{Live}, \quad \forall a_i \neq \text{Mocked}$); all other agents begin as Mocked replay shells.

Execution Bubble B(t): The minimal live subset $\mathcal{B}(t) = \{a_i \in A \mid \sigma_i(t) = \text{Live}\}$, dynamically scaled to preserve CPU resources.

Dynamic Promotion and Boundary Expansion Protocol

Step 1: Divergence Detection

The telemetry layer checks active transactions and flags an execution delta whenever a live agent deviates from historical records:

Condition: $\exists a_i \in \mathcal{B}(t) \quad \textbf{s.t.} \quad E_{\text{actual}}(a_i, t) \neq E_{\text{hist}}(a_i, t)$

Step 2: Neighborhood Extraction

Queries the interaction topology graph G to identify all immediate upstream and downstream communication dependencies:

Dependency Set: $\mathcal{N}(a_i) = \{a_j \in A \mid (a_i, a_j) \in M \ \vee \ (a_j, a_i) \in M\}$

Step 3: Dynamic Promotion (Hot-Swap)

For each dependent neighbor in the mock state, loads cognitive logic and rehydrates state via a state bridge:

Action: $\forall a_j \in \mathcal{N}(a_i)$ **with** $\sigma_j(t) = \textbf{Mocked}$: $\sigma_j(t) \leftarrow \textbf{Live}, \quad \mathcal{B}(t) \leftarrow \mathcal{B}(t) \cup \{a_j\}$ **(dynamically expanding the execution bubble)**

Step 4: Recursive Boundary Expansion

If newly promoted neighbors generate subsequent divergent actions, promotion cascades recursively along connected communication edges.

Step 5: Mass Demotion & Convergence

When agent interactions realign with baseline logs ($\forall a_i \in \mathcal{B}(t+k), \quad E_{\text{actual}}(a_i, t+k) = E_{\text{hist}}(a_i, t+k)$), the bubble collapses back to the minimal single-agent footprint:

Action: Mass Demotion $\sigma_j(t+k) \leftarrow \textbf{Mocked}, \quad \mathcal{B}(t+k) \rightarrow \{a_{\textbf{target}}\}$

Lock-Free Ring Buffers & Thread Decoupling

Thread-Safe Observability Layer

Lock-Free Dispatch: Employs ConcurrentLinkedQueue buffers to ensure sub-microsecond non-blocking event dispatch.

Decoupled I/O Pipeline: JSON serialization and persistent disk writes are handed off entirely to background daemon threads.

Probe Effect Mitigation: Prevents lock contention, priority inversion, and CPU context-switch thrashing along critical execution paths.

Architectural Ingestion Mechanics

NullRecorder Facade: When monitoring is toggled off, logging invocations short-circuit to a zero-cost no-op facade.

Dynamic Interceptor: Messaging shard interceptors hook into incoming and outgoing message waves at the network boundary.

Live WebSocket Push: Streams runtime diagnostic events and FSM state mutations directly to an external WebView control dashboard.

Decentralized Transactive Energy Market Benchmark

Benchmark Testbed & Target Platform

Host Platform (FLASH-MAS): Lightweight, actor-like Java multi-agent framework featuring decoupled communication shards.

Workload Threshold: Stress-tested across 10,000 completed multi-agent market transactions under unbounded CPU loops.

Zonal Coordinators

MarketAgent Instances: 3 regional market coordinator nodes managing continuous double-auction clearing houses.

Prosumer Entities

ProsumerAgent Instances: 99 concurrent autonomous entities running Reinforcement Learning (RL) bidding policies across 3 distinct zones.

Total Scale

102 Concurrent Entities: Continuous peer-to-peer and microgrid trading interactions generating dense execution message waves.

Empirical Telemetry Overhead (10,000 Transactions)

Is monitoring bringing overhead?

Configuration Profile (10,000 Trades)	Duration (ms)	Avg JVM CPU Load	Relative Overhead
Test 1: Logging DISABLED (All Live Baseline)	11,771 ms	6.40%	**Baseline (0.0%)**
Test 2: Logging ENABLED (Asynchronous I/O Active)	13,997 ms	9.34%	**+18.9% (Δ = 2,226 ms)**

Performance Analysis & Observations

Bounded Temporal Overhead: Active telemetry incurs an 18.9% execution time penalty under fully saturated, unbounded CPU loop stress testing.

Slight CPU Load Increase: CPU load rises by 2.94% due to lock-free serialization and background I/O operations.

Absence of Probe Effect: Agent scheduling, event ordering, and transaction sequencing are preserved because critical worker threads are never blocked.

Partitioned Scalability & Hot-Swap Efficiency

Is hybrid simulation effective?

Workload Partition Profile (10,000 Trades)	Duration (ms)	Avg CPU Load	Resource Compression / Impact
Test 5: Full Structural Emulation (All Mocked Baseline)	1,537 ms	17.21%	**Throughput floor (pure message dispatch)**
Test 3: Partial Emulation Profile (66/99 Mocked Nodes)	6,178 ms	7.63%	**47.5% runtime reduction vs. All-Live**
Test 4: Dynamic Mutation Sequence (Hot-Swap Injected)	11,571 ms	6.43%	**Seamless mid-run expansion; 0 drops/deadlocks**

Key Findings

Near-Linear Compute Compression: Restricting live execution to 1/3 of agents compresses runtime by almost 50% without loss of fidelity.

Zero-Interruption Hot-Swap: In Test 4, 66 agents were promoted on demand mid-run with zero dropped messages or deadlocks.

Summary of Contributions & Research Horizons

Core Contributions

Hybrid Architecture: Platform-agnostic framework combining lock-free telemetry capture with neuro-symbolic FSM runtime verification.

Resolution of Cognitive Drift: Formalized Reactive Hot-Swap by Cascade Propagation to adaptively expand execution bubbles around divergent nodes.

Empirically Validated Efficiency: Demonstrated 18.9% temporal overhead and up to 47.5% compute time compression in a 102-agent continuous auction environment.

Future Directions & Discussion

Semantic Log Embeddings: Integrating vector embeddings to enable automated clustering of behavioral divergence root causes.

Theoretical Demotion Bounds: Formalizing mathematical contraction bounds for automated bubble demotion.

Cross-Platform Validation: Deploying the hybrid observability framework across heterogeneous industrial actor frameworks.

Thank you!