Neuro-Symbolic Resilience Engine for Critical Infrastructure

Design, Architecture, and Evolutionary Implementation

PhD Candidate Name

January 18, 2026

Abstract

This document details the design and implementation of a Neuro-Symbolic Artificial Intelligence system aimed at monitoring critical infrastructure resilience. The system integrates sub-symbolic (neural/statistical) data processing with symbolic (ontological) reasoning to detect cascading failures in power grids. The architecture combines a Python-based Semantic Engine (using Owlready2 and SWRL rules) with a real-time Angular visualization dashboard. Special emphasis is placed on the evolutionary development of the system, addressing challenges such as state persistence, race conditions ("Zombie Connections"), and hierarchical logic enforcement to ensure high-fidelity alerting.

Contents

Chapter 1

Introduction

1.1 Problem Statement

Critical Infrastructure (CI) systems, such as power grids and hospitals, are increasingly interdependent. Traditional monitoring systems rely on linear thresholds (e.g., ”Temperature ¿ 90°C”). However, these systems fail to capture the semantic context of a failure, such as redundancy loss or cascading effects.

1.2 Proposed Solution

We propose a **Neuro-Symbolic Resilience Engine** that fuses:

- **The Neuro/Data Layer:** Handles real-time raw sensor data (Temperature, Load, Fuel) via Python.
- **The Symbolic/Logic Layer:** Uses an OWL Ontology and SWRL rules to reason about dependencies, redundancy, and propagation of failure.

Chapter 2

System Architecture

The system follows a decoupled Client-Server architecture communicating via WebSockets.

2.1 The Semantic Backend

The backend is built with **Python (FastAPI + Socket.IO)**. It hosts the `resilience.owl` ontology and the HermiT reasoner. It acts as the "Brain," converting raw data into semantic states (e.g., inferring *GridUnstable* from *FailedNode* and *OverloadedNode*).

2.2 The Visualization Frontend

The frontend is an **Angular** application utilizing:

- **Leaflet.js:** For geospatial representation of nodes.
- **NgRx/Services:** For state management and real-time streams.
- **Alert Banners:** For immediate heads-up notifications.

Chapter 3

Ontology Engineering

The core of the symbolic reasoning is the Ontology.

3.1 TBox: Classes and Properties

We defined the following hierarchy:

- **Classes:** `InfrastructureNode`, `PowerSubstation`, `CriticalAsset`, `BackupGenerator`.
- **Semantic Tags:** `FailedNode`, `OverloadedNode`, `LowFuelGenerator`.
- **Inferred States:** `GridUnstable` (Warning), `TotalBlackout` (Critical).

3.2 ABox: The Topology

The implemented scenario models a simplified Athens topology:

1. **Syntagma (Substation):** The main power source.
2. **Omonia (Substation):** The redundant (backup) power source.
3. **Evangelismos (Hospital):** The critical asset.
4. **Generator:** The hospital's last line of defense.

3.3 SWRL Rules (The PhD Logic)

We implemented Cascading Failure logic using Semantic Web Rule Language (SWRL):

Rule 1: Loss of Redundancy (Warning)

$$\begin{aligned} supplies(?p1, ?a) \wedge is\_redundant\_to(?p2, ?p1) \wedge \\ FailedNode(?p1) \wedge OverloadedNode(?p2) \rightarrow GridUnstable(?a) \end{aligned} \tag{3.1}$$

Rule 2: Total Collapse (Critical)

$$\begin{aligned} GridUnstable(?a) \wedge has\_backup(?a, ?g) \wedge \\ LowFuelGenerator(?g) \rightarrow TotalBlackout(?a) \end{aligned} \tag{3.2}$$

Chapter 4

Evolutionary Implementation

This chapter describes the iterative process and the sophisticated solutions developed to solve critical synchronization bugs.

4.1 Phase 1: The Initial Prototype

The initial version simply mapped sensor values to ontology classes. However, it suffered from "Memory Leaks" in the logic layer. The ontology retained failure states even after the simulation restarted.

4.2 Phase 2: State Persistence Solution (Brain Wipe)

To ensure every simulation run is independent, we implemented a **Hard Reset** mechanism. Upon every client connection, the server performs a "Brain Wipe":

```python
def hard_reset_ontology():
    print("WIPING MEMORY...")
    # Forcefully reset instances to their base classes
    syntagma.is_a = [PowerSubstation]
    omonia.is_a = [PowerSubstation]
    gen_evangelismos.is_a = [BackupGenerator]
    evangelismos.is_a = [CriticalAsset]
    try:
        with onto: sync_reasoner(infer_property_values=True)
    except: pass
```

Listing 4.1: The Hard Reset Function

4.3 Phase 3: The "Zombie Connection" Problem

We observed a race condition where refreshing the browser caused two parallel sessions (the old "Zombie" and the new "Active") to conflict. The old session would send "Critical" data while the new one sent "Safe" data, causing flickering alerts.

Solution: The Zombie Killer Pattern
We introduced a `Session ID (sid)` check. The server now tracks the `active_sid` and

discards packets from any other source.

```python
active_sid = None

@sio.event
async def connect(sid, environ):
    global active_sid
    active_sid = sid # Set the new captain
    hard_reset_ontology()

@sio.event
async def sensor_update(sid, data):
    # Ignore packets from old browser sessions
    if sid != active_sid:
        return
    # Process legitimate data...
```

Listing 4.2: Zombie Packet Filtering

4.4 Phase 4: Strict Hierarchical Logic

The Reasoner occasionally produced ”False Positives” due to inference latency. For example, declaring a Blackout while the generator still had fuel.

Solution: The Neuro-Gatekeeper
We implemented a strict logic layer in Python that overrides the Semantic Reasoner if physical constraints are not met.

```python
# Default to ontology inference
if TotalBlackout in evangelismos.is_a: final_status = "CRITICAL"

# STRICT SAFETY CHECK
# Even if Reasoner says Critical, if Fuel > 20, downgrade to Warning.
if final_status == "CRITICAL" and current_fuel > 20:
    print("Safety Override: Fuel is OK.")
    final_status = "WARNING"
```

Listing 4.3: Strict Logic Enforcement

Chapter 5

Simulation Results

The final system demonstrates a clear Cascading Failure scenario over a 20-second timeline.

5.1 Scenario Timeline

- **T+0s to T+5s (Normal Operation):** All nodes are green. Syntagma Temp: 45°C.
- **T+6s (Local Failure):** Syntagma overheats (95°C). The system remains stable due to Redundancy (Omonia). *No Alert issued.*
- **T+12s (Grid Loss):** Omonia overloads (95%). Redundancy is lost.
  - **Inference:** `GridUnstable`
  - **UI Action:** Orange Warning Banner: ”Grid Lost. Running on Backup.”
- **T+20s (Total Blackout):** Generator runs out of fuel (15%).
  - **Inference:** `TotalBlackout`
  - **UI Action:** Red Pulsing Banner: ”TOTAL BLACKOUT! Generator Dead.”

5.2 User Interface

The frontend visualizes this via:

1. **Live Dashboard Cards:** Showing real-time values and turning red/orange upon threshold breach.
2. **Geospatial Map:** Leaflet markers updating color dynamically based on node status.
3. **Notification System:** An animated top-bar alerting the user of semantic inferences.

Chapter 6

Conclusion

We have successfully developed a robust Neuro-Symbolic engine capable of identifying complex failure modes that simple threshold-based systems would miss. By combining the speed of Python/Angular with the reasoning depth of OWL/SWRL, and addressing real-world concurrency issues (Zombie connections), the system represents a PhD-level approach to Cyber-Physical System resilience.