Comprehensive Test Suite for Cognitive Architecture

Tests all components: beliefs, safety, verification, integration

"""

import sys

import time

from typing import List, Tuple

import math

# Import core components

# Note: In actual deployment, use proper imports

# from philosophical\_agi.core import \*

# from philosophical\_agi.safety import \*

# ============================================================================

# TEST UTILITIES

# ============================================================================

class TestResult:

"""Test result container"""

def \_\_init\_\_(self, name: str, passed: bool, message: str, duration: float = 0.0):

self.name = name

self.passed = passed

self.message = message

self.duration = duration

def \_\_repr\_\_(self):

status = "✓ PASS" if self.passed else "✗ FAIL"

return f"{status} | {self.name} ({self.duration\*1000:.1f}ms)\n {self.message}"

class TestSuite:

"""Test suite manager"""

def \_\_init\_\_(self, name: str):

self.name = name

self.results: List[TestResult] = []

self.start\_time = 0.0

def run\_test(self, test\_func, test\_name: str):

"""Run a single test and record result"""

print(f" Running: {test\_name}...", end=" ")

start = time.time()

try:

test\_func()

duration = time.time() - start

result = TestResult(test\_name, True, "Test passed successfully", duration)

print("✓")

except AssertionError as e:

duration = time.time() - start

result = TestResult(test\_name, False, str(e), duration)

print("✗")

except Exception as e:

duration = time.time() - start

result = TestResult(test\_name, False, f"Exception: {str(e)}", duration)

print("✗")

self.results.append(result)

return result

def print\_summary(self):

"""Print test summary"""

passed = sum(1 for r in self.results if r.passed)

total = len(self.results)

total\_time = sum(r.duration for r in self.results)

print("\n" + "="\*70)

print(f"TEST SUITE: {self.name}")

print("="\*70)

for result in self.results:

print(result)

print("\n" + "-"\*70)

print(f"Results: {passed}/{total} tests passed ({passed/total\*100:.1f}%)")

print(f"Total time: {total\_time\*1000:.1f}ms")

if passed == total:

print("✅ ALL TESTS PASSED!")

else:

print(f"❌ {total - passed} TEST(S) FAILED")

print("="\*70 + "\n")

return passed == total

# ============================================================================

# BELIEF SYSTEM TESTS

# ============================================================================

def test\_belief\_creation():

"""Test creating beliefs"""

from philosophical\_agi.core.belief\_system import DeSeBeliefSystem

beliefs = DeSeBeliefSystem("test\_agent")

# Test de dicto belief

beliefs.add\_de\_dicto\_belief("sky is blue")

assert "sky is blue" in beliefs.de\_dicto\_beliefs, "De dicto belief not added"

# Test de se belief

belief = beliefs.add\_de\_se\_belief("location", (1.0, 2.0, 0.0))

assert "SELF\_location" in belief, "De se belief format incorrect"

assert len(beliefs.de\_se\_beliefs) > 0, "De se belief not added"

def test\_belief\_confidence():

"""Test belief confidence tracking"""

from philosophical\_agi.core.belief\_system import DeSeBeliefSystem

beliefs = DeSeBeliefSystem("test\_agent")

beliefs.add\_de\_se\_belief("state", "active", confidence=0.85)

# Check confidence is stored

belief\_str = "SELF\_state=active"

assert belief\_str in beliefs.belief\_confidence, "Confidence not tracked"

assert abs(beliefs.belief\_confidence[belief\_str] - 0.85) < 0.01, "Confidence value incorrect"

def test\_propositional\_attitudes():

"""Test Chalmers' propositional attitudes"""

from philosophical\_agi.core.belief\_system import DeSeBeliefSystem

from philosophical\_agi.core.types import AttitudeType

beliefs = DeSeBeliefSystem("test\_agent")

# Add different attitude types

beliefs.add\_de\_dicto\_belief("goal exists")

beliefs.add\_desire("reach target", urgency=0.9)

beliefs.add\_de\_se\_belief("moving", True)

# Check attitudes were created

assert len(beliefs.attitudes) >= 3, "Not all attitudes created"

# Check attitude types

attitude\_types = {att.attitude\_type for att in beliefs.attitudes}

assert AttitudeType.BELIEF in attitude\_types, "Belief attitude missing"

assert AttitudeType.DESIRE in attitude\_types, "Desire attitude missing"

def test\_centered\_world\_update():

"""Test belief updates from centered worlds"""

from philosophical\_agi.core.belief\_system import DeSeBeliefSystem

from philosophical\_agi.core.types import CenteredWorld

beliefs = DeSeBeliefSystem("test\_agent")

# Create centered world

world = CenteredWorld("env", "test\_agent", 1)

# Update with observation

observation = {

'self\_location': (5.0, 5.0, 0.0),

'self\_state': 'active',

'threat\_level': 0.8

}

beliefs.update\_from\_centered\_world(world, observation)

# Check beliefs were created

assert len(beliefs.de\_se\_beliefs) > 0, "No de se beliefs created"

assert any('location' in b for b in beliefs.de\_se\_beliefs), "Location belief missing"

assert any('in\_danger' in b for b in beliefs.de\_se\_beliefs), "Danger belief missing (threat > 0.7)"

def test\_belief\_query():

"""Test querying beliefs"""

from philosophical\_agi.core.belief\_system import DeSeBeliefSystem

beliefs = DeSeBeliefSystem("test\_agent")

beliefs.add\_de\_se\_belief("location", (1, 2, 3))

beliefs.add\_de\_se\_belief("state", "moving")

beliefs.add\_de\_dicto\_belief("weather is sunny")

# Query beliefs

location\_beliefs = beliefs.query\_belief("location")

assert len(location\_beliefs) > 0, "Query returned no results"

assert any("location" in b for b in location\_beliefs), "Location belief not found"

# ============================================================================

# SAFETY VERIFICATION TESTS

# ============================================================================

def test\_safe\_action\_verification():

"""Test verification of safe actions"""

from philosophical\_agi.safety.verifier import FormalSafetyVerifier

from philosophical\_agi.core.types import AgentState, Action, WorkspaceBounds

verifier = FormalSafetyVerifier()

# Create safe state and action

state = AgentState(

position=(5.0, 5.0, 0.0),

workspace\_bounds=WorkspaceBounds(x\_min=0, x\_max=10, y\_min=0, y\_max=10)

)

action = Action(

action\_type="MOVE",

target\_position=(6.0, 6.0, 0.0)

)

is\_safe, msg, proof = verifier.verify\_action\_safety(state, action)

assert is\_safe, f"Safe action rejected: {msg}"

assert 'verified\_position' in proof, "Proof missing verified position"

def test\_out\_of\_bounds\_detection():

"""Test detection of out-of-bounds actions"""

from philosophical\_agi.safety.verifier import FormalSafetyVerifier

from philosophical\_agi.core.types import AgentState, Action, WorkspaceBounds

verifier = FormalSafetyVerifier()

state = AgentState(

position=(5.0, 5.0, 0.0),

workspace\_bounds=WorkspaceBounds(x\_min=0, x\_max=10, y\_min=0, y\_max=10)

)

# Out of bounds action

action = Action(

action\_type="MOVE",

target\_position=(15.0, 5.0, 0.0) # Beyond x\_max

)

is\_safe, msg, proof = verifier.verify\_action\_safety(state, action)

assert not is\_safe, "Out of bounds action not detected"

def test\_human\_proximity\_safety():

"""Test human proximity constraints"""

from philosophical\_agi.safety.verifier import FormalSafetyVerifier

from philosophical\_agi.core.types import AgentState, Action, WorkspaceBounds

verifier = FormalSafetyVerifier()

# State with human nearby

state = AgentState(

position=(5.0, 5.0, 0.0),

workspace\_bounds=WorkspaceBounds(x\_min=0, x\_max=10, y\_min=0, y\_max=10),

human\_positions=[(6.0, 6.0, 0.0)] # Human at (6, 6)

)

# Action too close to human (< 1.0m)

action = Action(

action\_type="MOVE",

target\_position=(6.5, 6.0, 0.0) # 0.5m from human

)

is\_safe, msg, proof = verifier.verify\_action\_safety(state, action)

assert not is\_safe, "Human proximity violation not detected"

assert proof['humans\_nearby'] == 1, "Human count incorrect"

def test\_obstacle\_avoidance():

"""Test obstacle avoidance"""

from philosophical\_agi.safety.verifier import FormalSafetyVerifier

from philosophical\_agi.core.types import AgentState, Action, WorkspaceBounds

verifier = FormalSafetyVerifier()

state = AgentState(

position=(5.0, 5.0, 0.0),

workspace\_bounds=WorkspaceBounds(x\_min=0, x\_max=10, y\_min=0, y\_max=10),

obstacles=[

{'position': (6.0, 6.0, 0.0), 'radius': 1.0}

]

)

# Action colliding with obstacle

action = Action(

action\_type="MOVE",

target\_position=(6.2, 6.0, 0.0) # Inside obstacle radius

)

is\_safe, msg, proof = verifier.verify\_action\_safety(state, action)

assert not is\_safe, "Obstacle collision not detected"

def test\_multiple\_constraints():

"""Test verification with multiple constraints"""

from philosophical\_agi.safety.verifier import FormalSafetyVerifier

from philosophical\_agi.core.types import AgentState, Action, WorkspaceBounds

verifier = FormalSafetyVerifier()

# Complex environment

state = AgentState(

position=(5.0, 5.0, 0.0),

workspace\_bounds=WorkspaceBounds(x\_min=0, x\_max=10, y\_min=0, y\_max=10),

human\_positions=[(3.0, 3.0, 0.0), (8.0, 8.0, 0.0)],

obstacles=[{'position': (5.0, 8.0, 0.0), 'radius': 0.5}]

)

# Safe action avoiding all constraints

action = Action(

action\_type="MOVE",

target\_position=(5.0, 6.0, 0.0)

)

is\_safe, msg, proof = verifier.verify\_action\_safety(state, action)

assert is\_safe, f"Safe action in complex environment rejected: {msg}"

assert proof['humans\_nearby'] == 2, "Human count incorrect"

assert proof['obstacles\_checked'] == 1, "Obstacle count incorrect"

# ============================================================================

# RUNTIME MONITORING TESTS

# ============================================================================

def test\_safety\_monitor\_passthrough():

"""Test that safe actions pass through monitor"""

from philosophical\_agi.safety.monitor import RuntimeSafetyMonitor

from philosophical\_agi.core.types import AgentState, Action, WorkspaceBounds

monitor = RuntimeSafetyMonitor()

state = AgentState(

position=(5.0, 5.0, 0.0),

workspace\_bounds=WorkspaceBounds(x\_min=0, x\_max=10, y\_min=0, y\_max=10)

)

action = Action(

action\_type="MOVE",

target\_position=(6.0, 6.0, 0.0)

)

safe\_action, info = monitor.monitor\_action(state, action)

assert not info['modified'], "Safe action was modified"

assert safe\_action.target\_position == action.target\_position, "Action was changed"

def test\_safety\_monitor\_intervention():

"""Test that unsafe actions trigger intervention"""

from philosophical\_agi.safety.monitor import RuntimeSafetyMonitor

from philosophical\_agi.core.types import AgentState, Action, WorkspaceBounds

monitor = RuntimeSafetyMonitor()

state = AgentState(

position=(5.0, 5.0, 0.0),

workspace\_bounds=WorkspaceBounds(x\_min=0, x\_max=10, y\_min=0, y\_max=10)

)

# Unsafe action

action = Action(

action\_type="MOVE",

target\_position=(15.0, 5.0, 0.0) # Out of bounds

)

safe\_action, info = monitor.monitor\_action(state, action)

assert info['modified'], "Unsafe action not modified"

assert safe\_action.action\_type == "EMERGENCY\_STOP", "Fallback not triggered"

def test\_nan\_detection():

"""Test detection of NaN values"""

from philosophical\_agi.safety.monitor import RuntimeSafetyMonitor

from philosophical\_agi.core.types import AgentState, Action

import numpy as np

monitor = RuntimeSafetyMonitor()

state = AgentState()

# Action with NaN

action = Action(

action\_type="MOVE",

target\_position=(np.nan, 5.0, 0.0)

)

safe\_action, info = monitor.monitor\_action(state, action)

assert info['modified'], "NaN not detected"

def test\_violation\_counting():

"""Test that violations are counted"""

from philosophical\_agi.safety.monitor import RuntimeSafetyMonitor

from philosophical\_agi.core.types import AgentState, Action, WorkspaceBounds

monitor = RuntimeSafetyMonitor()

state = AgentState(

workspace\_bounds=WorkspaceBounds(x\_min=0, x\_max=10)

)

# Trigger multiple violations

for i in range(3):

action = Action(

action\_type="MOVE",

target\_position=(15.0, 5.0, 0.0)

)

monitor.monitor\_action(state, action)

assert monitor.violations == 3, f"Violation count incorrect: {monitor.violations}"

# ============================================================================

# AGENT INTEGRATION TESTS

# ============================================================================

def test\_agent\_initialization():

"""Test agent initialization"""

from philosophical\_agi.core.agent import CognitiveAgent

agent = CognitiveAgent("test\_agent")

assert agent.id == "test\_agent", "Agent ID incorrect"

assert agent.time == 0, "Initial time should be 0"

assert agent.beliefs is not None, "Beliefs not initialized"

assert agent.safety\_monitor is not None, "Safety monitor not initialized"

def test\_cognitive\_cycle\_execution():

"""Test complete cognitive cycle"""

from philosophical\_agi.core.agent import CognitiveAgent

agent = CognitiveAgent("test\_agent")

observation = {

'agent\_position': (5.0, 5.0, 0.0),

'target': (8.0, 8.0, 0.0),

'threat\_level': 0.3

}

action, info = agent.cognitive\_cycle(observation)

assert agent.time == 1, "Time not incremented"

assert action is not None, "No action returned"

assert 'safety\_monitoring' in info, "Safety monitoring not in info"

assert info['beliefs\_updated'] > 0, "Beliefs not updated"

def test\_memory\_accumulation():

"""Test that agent accumulates episodic memory"""

from philosophical\_agi.core.agent import CognitiveAgent

agent = CognitiveAgent("test\_agent")

# Run multiple cycles

for i in range(5):

observation = {

'agent\_position': (5.0 + i\*0.1, 5.0 + i\*0.1, 0.0),

'target': (8.0, 8.0, 0.0)

}

agent.cognitive\_cycle(observation)

assert len(agent.episode\_memory) == 5, "Episode memory not accumulating"

assert agent.time == 5, "Time tracking incorrect"

def test\_danger\_response():

"""Test agent response to danger"""

from philosophical\_agi.core.agent import CognitiveAgent

agent = CognitiveAgent("test\_agent")

# High threat observation

observation = {

'agent\_position': (5.0, 5.0, 0.0),

'threat\_level': 0.9 # High threat

}

action, info = agent.cognitive\_cycle(observation)

# Check that danger belief was created

danger\_beliefs = agent.beliefs.query\_belief("in\_danger")

assert len(danger\_beliefs) > 0, "Danger belief not created"

# Check that escape action was planned

assert action.action\_type in ["ESCAPE", "EMERGENCY\_STOP"], "Escape behavior not triggered"

def test\_thought\_log\_generation():

"""Test thought log generation"""

from philosophical\_agi.core.agent import CognitiveAgent

agent = CognitiveAgent("test\_agent")

# Run a cycle

observation = {

'agent\_position': (5.0, 5.0, 0.0),

'target': (8.0, 8.0, 0.0)

}

agent.cognitive\_cycle(observation)

# Get thought log

log = agent.get\_thought\_log()

assert 'agent\_id' in log, "Agent ID missing from log"

assert 'attitudes' in log, "Attitudes missing from log"

assert 'de\_se\_beliefs' in log, "De se beliefs missing from log"

assert 'safety\_stats' in log, "Safety stats missing from log"

# ============================================================================

# PERFORMANCE TESTS

# ============================================================================

def test\_verification\_performance():

"""Test that verification is reasonably fast"""

from philosophical\_agi.safety.verifier import FormalSafetyVerifier

from philosophical\_agi.core.types import AgentState, Action, WorkspaceBounds

verifier = FormalSafetyVerifier()

state = AgentState(

position=(5.0, 5.0, 0.0),

workspace\_bounds=WorkspaceBounds(x\_min=0, x\_max=10, y\_min=0, y\_max=10),

human\_positions=[(3.0, 3.0, 0.0), (8.0, 8.0, 0.0)]

)

action = Action(

action\_type="MOVE",

target\_position=(6.0, 6.0, 0.0)

)

# Time verification

start = time.time()

for \_ in range(100):

verifier.verify\_action\_safety(state, action)

duration = time.time() - start

avg\_time = duration / 100

assert avg\_time < 0.01, f"Verification too slow: {avg\_time\*1000:.2f}ms per verification"

def test\_cognitive\_cycle\_performance():

"""Test cognitive cycle performance"""

from philosophical\_agi.core.agent import CognitiveAgent

agent = CognitiveAgent("test\_agent")

observation = {

'agent\_position': (5.0, 5.0, 0.0),

'target': (8.0, 8.0, 0.0)

}

# Time multiple cycles

start = time.time()

for \_ in range(50):

agent.cognitive\_cycle(observation)

duration = time.time() - start

avg\_time = duration / 50

assert avg\_time < 0.02, f"Cognitive cycle too slow: {avg\_time\*1000:.2f}ms per cycle"

# ============================================================================

# RUN ALL TESTS

# ============================================================================

def run\_all\_tests():

"""Run complete test suite"""

print("\n" + "="\*70)

print("PHILOSOPHICAL COGNITIVE ARCHITECTURE - TEST SUITE")

print("="\*70 + "\n")

all\_passed = True

# Belief System Tests

suite = TestSuite("Belief System Tests")

suite.run\_test(test\_belief\_creation, "Belief creation")

suite.run\_test(test\_belief\_confidence, "Belief confidence tracking")

suite.run\_test(test\_propositional\_attitudes, "Propositional attitudes")

suite.run\_test(test\_centered\_world\_update, "Centered world updates")

suite.run\_test(test\_belief\_query, "Belief querying")

all\_passed &= suite.print\_summary()

# Safety Verification Tests

suite = TestSuite("Safety Verification Tests")

suite.run\_test(test\_safe\_action\_verification, "Safe action verification")

suite.run\_test(test\_out\_of\_bounds\_detection, "Out of bounds detection")

suite.run\_test(test\_human\_proximity\_safety, "Human proximity safety")

suite.run\_test(test\_obstacle\_avoidance, "Obstacle avoidance")

suite.run\_test(test\_multiple\_constraints, "Multiple constraints")

all\_passed &= suite.print\_summary()

# Runtime Monitoring Tests

suite = TestSuite("Runtime Monitoring Tests")

suite.run\_test(test\_safety\_monitor\_passthrough, "Safe action passthrough")

suite.run\_test(test\_safety\_monitor\_intervention, "Unsafe action intervention")

suite.run\_test(test\_nan\_detection, "NaN detection")

suite.run\_test(test\_violation\_counting, "Violation counting")

all\_passed &= suite.print\_summary()

# Agent Integration Tests

suite = TestSuite("Agent Integration Tests")

suite.run\_test(test\_agent\_initialization, "Agent initialization")

suite.run\_test(test\_cognitive\_cycle\_execution, "Cognitive cycle execution")

suite.run\_test(test\_memory\_accumulation, "Memory accumulation")

suite.run\_test(test\_danger\_response, "Danger response")

suite.run\_test(test\_thought\_log\_generation, "Thought log generation")

all\_passed &= suite.print\_summary()

# Performance Tests

suite = TestSuite("Performance Tests")

suite.run\_test(test\_verification\_performance, "Verification performance")

suite.run\_test(test\_cognitive\_cycle\_performance, "Cognitive cycle performance")

all\_passed &= suite.print\_summary()

# Final summary

print("\n" + "="\*70)

if all\_passed:

print("🎉 ALL TEST SUITES PASSED!")

else:

print("❌ SOME TEST SUITES FAILED")

print("="\*70 + "\n")

return all\_passed

if \_\_name\_\_ == "\_\_main\_\_":

# Add parent directory to path for imports

import os

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

success = run\_all\_tests()

sys.exit(0 if success else 1)