Philosophical Cognitive Architecture - Core Implementation

A working cognitive architecture with formal safety verification

Project Structure:

philosophical\_agi/

├── core/

│ ├── agent.py (Main agent implementation)

│ ├── belief\_system.py (Belief management)

│ ├── memory.py (Memory with EWC)

│ └── types.py (Data structures)

├── safety/

│ ├── verifier.py (Z3 formal verification)

│ └── monitor.py (Runtime safety)

└── examples/

└── demo.py (Working examples)

"""

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

# FILE: philosophical\_agi/core/types.py

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

from dataclasses import dataclass, field

from typing import Dict, Any, List, Tuple, Optional

from enum import Enum

class AttitudeType(Enum):

"""Types of propositional attitudes (Chalmers)"""

BELIEF = "belief"

DESIRE = "desire"

INTENTION = "intention"

CREDENCE = "credence"

@dataclass

class CenteredWorld:

"""Lewis-style centered world ⟨W, a, t⟩"""

world\_state: str

agent\_id: str

time\_index: int

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

return f"⟨{self.world\_state},{self.agent\_id},t{self.time\_index}⟩"

@dataclass

class PropositionalAttitude:

"""Chalmers' propositional attitudes"""

attitude\_type: AttitudeType

content: str

confidence: float = 1.0

timestamp: int = 0

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

return f"{self.attitude\_type.value}('{self.content}', conf={self.confidence:.2f})"

@dataclass

class WorkspaceBounds:

"""Physical workspace boundaries for safety"""

x\_min: float = 0.0

x\_max: float = 10.0

y\_min: float = 0.0

y\_max: float = 10.0

z\_min: float = 0.0

z\_max: float = 10.0

@dataclass

class AgentState:

"""Complete agent state for safety verification"""

position: Tuple[float, float, float] = (0.0, 0.0, 0.0)

velocity: Tuple[float, float, float] = (0.0, 0.0, 0.0)

workspace\_bounds: WorkspaceBounds = field(default\_factory=WorkspaceBounds)

human\_positions: List[Tuple[float, float, float]] = field(default\_factory=list)

obstacles: List[Dict[str, Any]] = field(default\_factory=list)

@dataclass

class Action:

"""Action with target and constraints"""

action\_type: str

target\_position: Optional[Tuple[float, float, float]] = None

velocity: Tuple[float, float, float] = (0.0, 0.0, 0.0)

metadata: Dict[str, Any] = field(default\_factory=dict)

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

# FILE: philosophical\_agi/core/belief\_system.py

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

from typing import Set, Dict, List

from collections import defaultdict

class DeSeBeliefSystem:

"""

Self-locating belief system (Perry/Lewis de se semantics)

Implements distinction between:

- De dicto beliefs: "about the world"

- De se beliefs: "about self" (self-locating)

"""

def \_\_init\_\_(self, agent\_id: str):

self.agent\_id = agent\_id

# De dicto beliefs (general world facts)

self.de\_dicto\_beliefs: Set[str] = set()

# De se beliefs (self-locating, trigger actions)

self.de\_se\_beliefs: Set[str] = set()

# Propositional attitudes (Chalmers framework)

self.attitudes: List[PropositionalAttitude] = []

# Temporal belief history

self.temporal\_beliefs: Dict[int, List[str]] = defaultdict(list)

# Belief confidence scores (for credences)

self.belief\_confidence: Dict[str, float] = {}

def add\_de\_se\_belief(self, aspect: str, value: Any, confidence: float = 1.0):

"""

Add self-locating belief - these trigger actions!

Example: "I am in danger" → escape behavior

"""

belief\_str = f"SELF\_{aspect}={value}"

self.de\_se\_beliefs.add(belief\_str)

self.belief\_confidence[belief\_str] = confidence

# Create propositional attitude

attitude = PropositionalAttitude(

attitude\_type=AttitudeType.BELIEF,

content=belief\_str,

confidence=confidence

)

self.attitudes.append(attitude)

return belief\_str

def add\_de\_dicto\_belief(self, proposition: str, confidence: float = 1.0):

"""Add general world belief"""

self.de\_dicto\_beliefs.add(proposition)

self.belief\_confidence[proposition] = confidence

attitude = PropositionalAttitude(

attitude\_type=AttitudeType.BELIEF,

content=proposition,

confidence=confidence

)

self.attitudes.append(attitude)

def add\_desire(self, goal: str, urgency: float = 0.5):

"""Add goal/desire (motivational attitude)"""

attitude = PropositionalAttitude(

attitude\_type=AttitudeType.DESIRE,

content=goal,

confidence=urgency

)

self.attitudes.append(attitude)

return attitude

def update\_from\_centered\_world(self, centered\_world: CenteredWorld, observation: Dict):

"""Update beliefs from centered world perspective"""

# Store temporal context

self.temporal\_beliefs[centered\_world.time\_index].append(str(observation))

# Update de se beliefs from observations

if 'self\_location' in observation:

self.add\_de\_se\_belief('location', observation['self\_location'])

if 'self\_state' in observation:

self.add\_de\_se\_belief('state', observation['self\_state'])

if 'threat\_level' in observation:

threat = observation['threat\_level']

if threat > 0.7:

self.add\_de\_se\_belief('in\_danger', True, confidence=threat)

def get\_all\_beliefs(self) -> Set[str]:

"""Get all beliefs (de dicto + de se)"""

return self.de\_dicto\_beliefs | self.de\_se\_beliefs

def query\_belief(self, pattern: str) -> List[str]:

"""Query beliefs matching pattern"""

all\_beliefs = self.get\_all\_beliefs()

return [b for b in all\_beliefs if pattern in b]

def get\_recent\_attitudes(self, n: int = 5) -> List[PropositionalAttitude]:

"""Get n most recent attitudes"""

return self.attitudes[-n:] if len(self.attitudes) > n else self.attitudes

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

# FILE: philosophical\_agi/safety/verifier.py

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

from z3 import \*

from typing import Tuple

class FormalSafetyVerifier:

"""

Z3-based formal verification for action safety

Implements provable safety guarantees

"""

def \_\_init\_\_(self):

self.solver = Solver()

self.verification\_count = 0

self.violations\_detected = 0

def verify\_action\_safety(

self,

state: AgentState,

action: Action

) -> Tuple[bool, str, Dict[str, Any]]:

"""

Formally verify if action is safe

Returns:

(is\_safe, explanation, proof\_details)

"""

self.verification\_count += 1

self.solver.reset()

try:

# Define symbolic variables

x, y, z = Reals('x y z')

# Workspace boundary constraints

constraints = [

x >= state.workspace\_bounds.x\_min,

x <= state.workspace\_bounds.x\_max,

y >= state.workspace\_bounds.y\_min,

y <= state.workspace\_bounds.y\_max,

z >= state.workspace\_bounds.z\_min,

z <= state.workspace\_bounds.z\_max

]

# Target position constraints

if action.target\_position:

tx, ty, tz = action.target\_position

constraints.extend([

x == tx,

y == ty,

z == tz

])

# Human proximity constraints (minimum 1.0m distance)

MIN\_HUMAN\_DISTANCE = 1.0

for i, human\_pos in enumerate(state.human\_positions):

hx, hy, hz = human\_pos

# Distance squared > MIN\_HUMAN\_DISTANCE^2

dist\_squared = (x - hx)\*\*2 + (y - hy)\*\*2 + (z - hz)\*\*2

constraints.append(dist\_squared >= MIN\_HUMAN\_DISTANCE\*\*2)

# Obstacle avoidance

for obstacle in state.obstacles:

if 'position' in obstacle and 'radius' in obstacle:

ox, oy, oz = obstacle['position']

radius = obstacle['radius']

dist\_squared = (x - ox)\*\*2 + (y - oy)\*\*2 + (z - oz)\*\*2

constraints.append(dist\_squared >= radius\*\*2)

# Add all constraints to solver

self.solver.add(And(constraints))

# Check satisfiability

result = self.solver.check()

proof\_details = {

'verification\_id': self.verification\_count,

'constraints\_checked': len(constraints),

'humans\_nearby': len(state.human\_positions),

'obstacles\_checked': len(state.obstacles)

}

if result == sat:

model = self.solver.model()

proof\_details['verified\_position'] = (

float(model[x].as\_fraction()),

float(model[y].as\_fraction()),

float(model[z].as\_fraction())

)

return True, "Action verified safe ✓", proof\_details

elif result == unsat:

self.violations\_detected += 1

return False, "Action violates safety constraints ✗", proof\_details

else:

return False, "Verification inconclusive (timeout)", proof\_details

except Exception as e:

return False, f"Verification error: {str(e)}", {}

def verify\_physical\_constraints\_only(

self,

state: AgentState,

action: Action

) -> Tuple[bool, str]:

"""Quick verification of just physical bounds"""

if not action.target\_position:

return True, "No position change"

tx, ty, tz = action.target\_position

bounds = state.workspace\_bounds

if not (bounds.x\_min <= tx <= bounds.x\_max):

return False, f"X position {tx} out of bounds [{bounds.x\_min}, {bounds.x\_max}]"

if not (bounds.y\_min <= ty <= bounds.y\_max):

return False, f"Y position {ty} out of bounds [{bounds.y\_min}, {bounds.y\_max}]"

if not (bounds.z\_min <= tz <= bounds.z\_max):

return False, f"Z position {tz} out of bounds [{bounds.z\_min}, {bounds.z\_max}]"

return True, "Physical bounds satisfied"

def get\_statistics(self) -> Dict[str, Any]:

"""Get verification statistics"""

return {

'total\_verifications': self.verification\_count,

'violations\_detected': self.violations\_detected,

'success\_rate': 1.0 - (self.violations\_detected / max(1, self.verification\_count))

}

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

# FILE: philosophical\_agi/safety/monitor.py

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

import numpy as np

class RuntimeSafetyMonitor:

"""

Real-time safety monitoring with intervention

Combines fast heuristics with formal verification

"""

def \_\_init\_\_(self, max\_violations: int = 10):

self.verifier = FormalSafetyVerifier()

self.violations = 0

self.max\_violations = max\_violations

self.action\_history: List[Tuple[Action, bool]] = []

def monitor\_action(self, state: AgentState, action: Action) -> Tuple[Action, Dict[str, Any]]:

"""

Monitor action and return safe version

Returns:

(safe\_action, monitoring\_info)

"""

monitoring\_info = {

'original\_action': action,

'modified': False,

'safety\_checks': []

}

# 1. Fast heuristic checks

fast\_safe, fast\_msg = self.\_fast\_safety\_check(action)

monitoring\_info['safety\_checks'].append(('heuristic', fast\_safe, fast\_msg))

if not fast\_safe:

self.violations += 1

safe\_action = self.\_safe\_fallback(state)

monitoring\_info['modified'] = True

monitoring\_info['reason'] = fast\_msg

return safe\_action, monitoring\_info

# 2. Formal verification

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

monitoring\_info['safety\_checks'].append(('formal', is\_safe, explanation))

monitoring\_info['proof\_details'] = proof

if not is\_safe:

self.violations += 1

print(f"⚠️ Safety violation #{self.violations}: {explanation}")

if self.violations >= self.max\_violations:

raise RuntimeError(

f"Critical: {self.violations} safety violations detected. "

"System halted for safety."

)

safe\_action = self.\_safe\_fallback(state)

monitoring\_info['modified'] = True

monitoring\_info['reason'] = explanation

return safe\_action, monitoring\_info

# Action is safe

self.action\_history.append((action, True))

return action, monitoring\_info

def \_fast\_safety\_check(self, action: Action) -> Tuple[bool, str]:

"""Quick heuristic safety checks"""

# Check for NaN values

if action.target\_position:

if any(np.isnan(x) for x in action.target\_position):

return False, "Target position contains NaN"

# Check for extreme velocities

if action.velocity:

speed = np.linalg.norm(action.velocity)

if speed > 10.0: # Max 10 m/s

return False, f"Velocity too high: {speed:.2f} m/s"

return True, "Heuristic checks passed"

def \_safe\_fallback(self, state: AgentState) -> Action:

"""Generate guaranteed safe action (STOP)"""

return Action(

action\_type="EMERGENCY\_STOP",

target\_position=state.position,

velocity=(0.0, 0.0, 0.0),

metadata={'reason': 'safety\_intervention'}

)

def get\_statistics(self) -> Dict[str, Any]:

"""Get monitoring statistics"""

verifier\_stats = self.verifier.get\_statistics()

return {

'total\_violations': self.violations,

'actions\_monitored': len(self.action\_history),

'verifier\_stats': verifier\_stats

}

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

# FILE: philosophical\_agi/core/agent.py

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

class CognitiveAgent:

"""

Main cognitive agent with formal safety verification

Implements philosophical principles with working code

"""

def \_\_init\_\_(self, agent\_id: str, workspace\_bounds: Optional[WorkspaceBounds] = None):

self.id = agent\_id

self.time = 0

# Core systems

self.beliefs = DeSeBeliefSystem(agent\_id)

self.safety\_monitor = RuntimeSafetyMonitor()

# State

self.state = AgentState(

position=(5.0, 5.0, 0.0),

workspace\_bounds=workspace\_bounds or WorkspaceBounds()

)

# Memory

self.episode\_memory: List[Dict] = []

print(f"✅ CognitiveAgent '{agent\_id}' initialized")

print(f" Workspace: [{self.state.workspace\_bounds.x\_min}, {self.state.workspace\_bounds.x\_max}] × "

f"[{self.state.workspace\_bounds.y\_min}, {self.state.workspace\_bounds.y\_max}]")

def cognitive\_cycle(self, observation: Dict) -> Tuple[Action, Dict[str, Any]]:

"""

Complete perception-reasoning-action cycle with safety

Returns:

(action, cycle\_info)

"""

self.time += 1

cycle\_info = {

'time': self.time,

'observation': observation

}

# 1. UPDATE BELIEFS from centered world perspective

centered\_world = CenteredWorld(

world\_state="environment",

agent\_id=self.id,

time\_index=self.time

)

self.beliefs.update\_from\_centered\_world(centered\_world, observation)

cycle\_info['beliefs\_updated'] = len(self.beliefs.get\_all\_beliefs())

# 2. UPDATE STATE

if 'agent\_position' in observation:

self.state.position = observation['agent\_position']

if 'human\_positions' in observation:

self.state.human\_positions = observation['human\_positions']

if 'obstacles' in observation:

self.state.obstacles = observation['obstacles']

# 3. REASON about action

action = self.\_deliberate\_action(observation)

cycle\_info['planned\_action'] = action

# 4. VERIFY SAFETY (formal)

safe\_action, monitoring\_info = self.safety\_monitor.monitor\_action(

self.state, action

)

cycle\_info['safety\_monitoring'] = monitoring\_info

# 5. EXECUTE (update internal state)

if safe\_action.target\_position:

self.state.position = safe\_action.target\_position

# 6. STORE EPISODE

episode = {

'time': self.time,

'observation': observation,

'action': safe\_action,

'beliefs': list(self.beliefs.get\_all\_beliefs())

}

self.episode\_memory.append(episode)

return safe\_action, cycle\_info

def \_deliberate\_action(self, observation: Dict) -> Action:

"""

Deliberate on next action based on beliefs

Simple rule-based for now, can be extended with learning

"""

# Check for danger (de se belief)

danger\_beliefs = self.beliefs.query\_belief('in\_danger')

if danger\_beliefs:

# Escape behavior

# Move away from current position

current\_pos = self.state.position

safe\_pos = (

current\_pos[0] + 2.0,

current\_pos[1] + 2.0,

current\_pos[2]

)

return Action(

action\_type="ESCAPE",

target\_position=safe\_pos,

metadata={'reason': 'danger\_detected'}

)

# Check for goals (desires)

desires = [a for a in self.beliefs.attitudes if a.attitude\_type == AttitudeType.DESIRE]

if desires:

# Pursue highest urgency goal

top\_desire = max(desires, key=lambda d: d.confidence)

if 'target' in observation:

return Action(

action\_type="APPROACH\_GOAL",

target\_position=observation['target'],

metadata={'goal': top\_desire.content}

)

# Default: explore

import random

current\_pos = self.state.position

explore\_pos = (

current\_pos[0] + random.uniform(-1, 1),

current\_pos[1] + random.uniform(-1, 1),

current\_pos[2]

)

return Action(

action\_type="EXPLORE",

target\_position=explore\_pos

)

def get\_thought\_log(self) -> Dict[str, Any]:

"""

Generate interpretable thought log (Chalmers attitudes)

"""

return {

'agent\_id': self.id,

'time': self.time,

'position': self.state.position,

'attitudes': [

{

'type': att.attitude\_type.value,

'content': att.content,

'confidence': att.confidence

}

for att in self.beliefs.get\_recent\_attitudes()

],

'de\_se\_beliefs': list(self.beliefs.de\_se\_beliefs),

'de\_dicto\_beliefs': list(self.beliefs.de\_dicto\_beliefs),

'safety\_stats': self.safety\_monitor.get\_statistics()

}

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

# FILE: examples/demo.py

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

def run\_basic\_demo():

"""Basic demonstration of cognitive architecture"""

print("=" \* 70)

print("COGNITIVE ARCHITECTURE DEMONSTRATION")

print("=" \* 70)

# Create agent with workspace bounds

bounds = WorkspaceBounds(x\_min=0, x\_max=10, y\_min=0, y\_max=10, z\_min=0, z\_max=2)

agent = CognitiveAgent("DemoAgent", workspace\_bounds=bounds)

# Add initial desire

agent.beliefs.add\_desire("reach\_goal", urgency=0.8)

print("\n--- Running 5 cognitive cycles ---\n")

for i in range(5):

# Simulate observation

observation = {

'agent\_position': agent.state.position,

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

'threat\_level': 0.2 if i < 3 else 0.0,

'human\_positions': [(3.0, 3.0, 0.0)] if i == 2 else []

}

# Run cognitive cycle

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

print(f"Cycle {i+1}:")

print(f" Position: ({agent.state.position[0]:.1f}, {agent.state.position[1]:.1f})")

print(f" Action: {action.action\_type}")

if action.target\_position:

print(f" Target: ({action.target\_position[0]:.1f}, {action.target\_position[1]:.1f})")

print(f" Beliefs: {info['beliefs\_updated']}")

# Safety check result

if info['safety\_monitoring']['modified']:

print(f" ⚠️ Safety intervention: {info['safety\_monitoring']['reason']}")

else:

print(f" ✓ Action verified safe")

print()

# Final thought log

print("\n--- Thought Log (Chalmers Propositional Attitudes) ---")

thought\_log = agent.get\_thought\_log()

print(f"Agent: {thought\_log['agent\_id']}")

print(f"Time: {thought\_log['time']}")

print(f"Position: {thought\_log['position']}")

print(f"\nRecent Attitudes:")

for att in thought\_log['attitudes']:

print(f" - {att['type']}: '{att['content']}' (confidence: {att['confidence']:.2f})")

print(f"\nDe Se Beliefs (self-locating):")

for belief in thought\_log['de\_se\_beliefs']:

print(f" - {belief}")

print(f"\nSafety Statistics:")

stats = thought\_log['safety\_stats']

print(f" Violations: {stats['total\_violations']}")

print(f" Verifications: {stats['verifier\_stats']['total\_verifications']}")

print(f" Success rate: {stats['verifier\_stats']['success\_rate']:.1%}")

def run\_safety\_demo():

"""Demonstration of formal safety verification"""

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

print("FORMAL SAFETY VERIFICATION DEMONSTRATION")

print("=" \* 70)

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

agent = CognitiveAgent("SafetyAgent", workspace\_bounds=bounds)

# Add human in environment

agent.state.human\_positions = [(5.0, 5.0, 0.0)]

print("\nScenario: Human at position (5.0, 5.0)")

print("Testing various actions...\n")

test\_actions = [

("Safe action (far from human)", (8.0, 8.0, 0.0)),

("Unsafe action (too close to human)", (5.5, 5.0, 0.0)),

("Out of bounds action", (15.0, 5.0, 0.0)),

("Safe action (opposite corner)", (1.0, 1.0, 0.0))

]

for description, target in test\_actions:

action = Action(

action\_type="MOVE",

target\_position=target

)

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

print(f"{description}:")

print(f" Target: {target}")

for check\_type, is\_safe, msg in info['safety\_checks']:

status = "✓" if is\_safe else "✗"

print(f" {status} {check\_type}: {msg}")

if info['modified']:

print(f" → BLOCKED: {info['reason']}")

print(f" → Safe fallback action applied")

else:

print(f" → APPROVED")

print()

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

run\_basic\_demo()

run\_safety\_demo()