GISMOL: A General Intelligent Systems Modelling Language

Harris Wang *

Posted Date: 26 January 2026

doi: 10.20944/preprints202601.0558.v2

Keywords: artificial general intelligence; constrained object hierarchies; neuro-symbolic AI; intelligent systems modelling; constraint programming; hierarchical reasoning; Python framework; constraint reasoning

 doi:10.20944/preprints202601.0558.v2

Article

GISMOL: A General Intelligent Systems Modelling Language

Harris Wang

School of Computing and Information Systems, Athabasca University, Athabasca, Canada;
harrisw@athabascau.ca

Abstract

This paper presents GISMOL (General Intelligent System Modelling Language), a Python-based framework implementing Constrained Object Hierarchies (COH)—a neuroscience-inspired theoretical framework for Artificial General Intelligence (AGI). COH and GISMOL together provide a unified language for modelling and implementing intelligent systems across diverse domains including healthcare, manufacturing, finance, and governance. The framework bridges symbolic AI and neural computation through its core architecture of constraint-aware objects with embedded neural components, hierarchical reasoning capabilities, and natural language integration. We demonstrate how GISMOL translates COH's formal 9-tuple representation into executable systems with six comprehensive case studies, showing its versatility in modelling complex intelligent behaviors while maintaining theoretical rigor. The implementation includes specialized modules for neural integration, multi-domain reasoning, and natural language processing, all built around the COHObject abstraction that encapsulates intelligence as constrained hierarchical structures.

Keywords: artificial general intelligence; constrained object hierarchies; neuro-symbolic AI; intelligent systems modelling; constraint programming; hierarchical reasoning; Python framework; constraint reasoning

1. Introduction

The pursuit of Artificial General Intelligence (AGI) represents one of the most ambitious goals in computer science and cognitive science [1]. Unlike narrow AI systems optimized for specific tasks, AGI aims to achieve human-level versatility, adaptability, and reasoning across multiple domains [2]. Current approaches to AGI development face fundamental challenges, including the integration of neural learning with symbolic reasoning, the maintenance of safety and ethical constraints, and the creation of systems that can generalize beyond their training distributions [3].

Traditional deep learning approaches, while powerful for pattern recognition, struggle with explicit reasoning, constraint satisfaction, and interpretability [4]. Conversely, symbolic AI systems excel at logical inference and knowledge representation but lack the learning capabilities and robustness to uncertainty of neural approaches [5]. This dichotomy has led to renewed interest in neuro-symbolic integration, which seeks to combine the strengths of both paradigms [6].

GISMOL addresses these challenges through its foundation in Constrained Object Hierarchies (COH), a theoretical framework that models intelligent systems as hierarchical compositions of objects subject to multi-domain constraints [7]. COH extends beyond typical neuro-symbolic approaches by embedding constraints directly within the system architecture through identity, trigger, and goal constraints monitored by autonomous daemons [8].

Scope and contributions of this paper (development-stage):

1. A formal presentation of the COH theoretical framework and its neuroscience grounding.
2. A description of the design goals and initial API of GISMOL, a prototype Python library for COH.

 doi:10.20944/preprints202601.0558.v2

3. Six conceptual case studies and code snippets illustrating intended usage patterns and cross-domain applicability.
4. A preliminary comparison with existing approaches to indicate potential advantages and gaps.
5. A development roadmap, implementation strategies, and evaluation plan suitable for future empirical validation.

Disclaimer. GISMOL is not yet a mature production framework. The API is subject to change, and examples provided are prototypes or simulated demonstrations, not deployed systems.

2. Literature Review

2.1. Symbolic AI and Knowledge Representation

Early AI research focused heavily on symbolic approaches, with systems like SOAR [9] and ACT-R [10] providing cognitive architectures for representing knowledge and reasoning. These systems excelled at tasks requiring explicit rule-based reasoning but struggled with perception, learning, and handling uncertainty [11]. More recent approaches like TensorLog [12] have attempted to bridge symbolic and neural representations through differentiable inference but often lack comprehensive constraint management capabilities.

2.2. Neural Networks and Deep Learning

The resurgence of neural networks, particularly through deep learning architectures [13], has revolutionized fields like computer vision, natural language processing, and reinforcement learning [14]. However, these systems typically operate as "black boxes" with limited interpretability and difficulty incorporating explicit constraints or symbolic knowledge [15].

2.3. Neuro-Symbolic Integration

Recent work in neuro-symbolic AI has explored various integration strategies. DeepProbLog [16] combines probabilistic logic programming with neural networks, while Neural Theorem Provers [17] use differentiable reasoning. Neurosymbolic Concept Learners [18] integrate perception with reasoning. However, these approaches often lack the hierarchical organization and comprehensive constraint system provided by COH [19].

2.4. Cognitive Architectures and AGI

Cognitive architectures like LIDA [20] and CLARION [21] attempt to model human cognition more comprehensively but often lack practical implementation frameworks for real-world applications. GISMOL builds on these ideas while providing a concrete, executable modeling language.

2.5. Constraint Satisfaction and Optimization

Constraint programming [22] and constraint satisfaction problems [23] provide formalisms for representing and solving constrained systems. GISMOL extends these ideas through its hierarchical constraint system and integration with neural components for adaptive constraint resolution.

3. Introducing Constrained Object Hierarchies (COH)

3.1. Formal Definition

Constrained Object Hierarchies (COH) provides a formal framework for modeling general intelligent systems as 9-tuple structures:

$O = (C, A, M, N, E, I, T, G, D)$

Where:

 doi:10.20944/preprints202601.0558.v2

C (Components): Sub-objects forming a compositional hierarchy, enabling complex system decomposition.

A (Attributes): State variables describing object properties, representing system state.

M (Methods): Executable actions or behaviors, implementing system functionality.

N (Neural Components): Adaptive models for learning and inference, providing statistical learning capabilities.

E (Embedding): Neural components for semantic representation, enabling knowledge integration.

I (Identity Constraints): Fundamental structural and logical rules, defining system invariants.

T (Trigger Constraints): Event-condition-action mechanisms, enabling reactive behavior.

G (Goal Constraints): Optimization objectives guiding intelligent behavior, providing purposeful direction.

D (Constraint Daemons): Real-time monitors enforcing constraints, ensuring continuous compliance.

3.2. Neuroscience Grounding

COH draws inspiration from hierarchical organization in biological neural systems [24], where different brain regions specialize in different functions while maintaining integrated operation. The hippocampus provides episodic memory (attributes), the prefrontal cortex implements executive functions (methods), and basal ganglia mediate reinforcement learning (goal constraints) [25]. Constraint daemons parallel regulatory mechanisms in biological systems that maintain homeostasis [26].

3.3. Theoretical Significance

COH provides several theoretical advances:

1. Unified Representation: Integrates symbolic, neural, and constraint-based representations.
2. Hierarchical Composition: Supports arbitrary decomposition while maintaining constraint propagation.
3. Multi-domain Constraints: Accommodates biological, physical, temporal, and other domain-specific constraints.
4. Real-time Monitoring: Constraint daemons provide continuous validation.
5. Adaptive Optimization: Goal constraints guide learning toward multiple objectives.

4. Structure of GISMOL

Development status. GISMOL is a prototype library; several modules are partially implemented, and others are in design or stubbed form. APIs described below represent the intended interface and may evolve.

GISMOL aims to implement COH theory as a Python library with four main modules:

4.1. Core Module (gismol.core)

Core Module establishes the foundational COHObject class and its supporting infrastructure. Current prototypes include:

- COHObject: Base class for intelligent system components.
- ConstraintSystem: Manages constraint evaluation and resolution.
- ConstraintDaemon: Autonomous agents for real-time constraint monitoring.
- COHRepository: Manages object collections and relationships.

 doi:10.20944/preprints202601.0558.v2

4.2. Neural Module (gismol.neural)

The Neural Module designs constraint-aware neural components that integrate with COH objects:

- NeuralComponent: Base class combining nn.Module with COHObject.
- EmbeddingModel: Generates semantic representations of objects and text.
- ConstraintAwareOptimizer: Optimizers that respect constraints.
- Specialized models for classification, regression, and generation (in progress).

4.3. Reasoners Module (gismol.reasoners)

Provides domain-specific engines for constraint evaluation:

- BaseReasoner: Fallback implementation with robust error handling.
- Domain-specific reasoners (Biological, Physical, Geometric, etc.).
- Advanced reasoning systems (Causal, Probabilistic, Temporal, etc.).
- Registry pattern for dynamic reasoner discovery.

4.4. NLP Module (gismol.nlp)

Enables natural language understanding with constraint awareness:

- COHDialogueManager: Manages conversations with constraint validation (prototype).
- EntityRelationExtractor: Extracts knowledge from text into COHObjects (prototype).
- Text2COH: Converts documents into hierarchical object structures (planned).
- ResponseValidator: Ensures language generation respects constraints (planned).

4.5. COH-to-GISMOL Mapping

Table 1 shows the mapping between COH theoretical elements and GISMOL implementation constructs (design intent).

Table 1. COH Elements to GISMOL Implementation Mapping.

COH Element	GISMOL Implementation	Primary Module
C (Components)	COHObject hierarchy, COHRepository	gismol.core
A (Attributes)	COHObject.attributes dictionary	gismol.core
M (Methods)	COHObject.methods dictionary	gismol.core
N (Neural Components)	NeuralComponent subclasses	gismol.neural
E (Embedding)	EmbeddingModel classes	gismol.neural
I (Identity Constraints)	Constraint objects with identity category	gismol.core
T (Trigger Constraints)	Constraint objects with trigger type	gismol.core
G (Goal Constraints)	Constraint objects with goal type	gismol.core
D (Constraint Daemons)	ConstraintDaemon instances	gismol.core

5. Constructs of GISMOL

5.1. Classes in the Core

COHObject Class: Fundamental Intelligent Unit

The COHObject class implements the core abstraction of intelligent entities in GISMOL:

```
class COHObject:
    """Fundamental unit of intelligence in COH framework"""

    def __init__(self, name):
        self.name = name
```

 doi:10.20944/preprints202601.0558.v2


```
        self.identity_constraints = []
        self.trigger_constraints = []
        self.goal_constraints = []
        self.neural_components = {}
        self.daemons = {}
        self.relations = COHRelation()
        self._embedding_model = None

    def add_identity_constraint(self, constraint_spec):
        """Add fundamental structural constraint"""
        constraint = IdentityConstraint.from_spec(constraint_spec)
        self.identity_constraints.append(constraint)

    def add_neural_component(self, name, component, is_embedding_model=False):
        """Add adaptive neural component"""
        if is_embedding_model:
            self._embedding_model = component
        else:
            self.neural_components[name] = component

    def semantic_distance(self, other):
        """Calculate semantic distance to another object"""
        if self._embedding_model:
            return self._embedding_model.distance(self, other)
        return float('inf')
```

Each COHObject encapsulates a complete intelligent entity with constraints, neural capabilities, and relational context.

COHRepository Class

COHRepository class manages collections of COHObjects with semantic search:

```
class COHRepository:
    """Repository for COHObjects with semantic operations"""

    def find_semantic_matches(self, query, threshold=0.7):
        """Find objects semantically similar to query"""
        query_embedding = self.embedder.embed_text(query)
        matches = []

        for obj in self.objects.values():
            obj_embedding = obj.get_embedding()
            similarity = cosine_similarity(query_embedding, obj_embedding)
            if similarity >= threshold:
                matches.append((obj, similarity))
```

 doi:10.20944/preprints202601.0558.v2


```
        return sorted(matches, key=lambda x: x[1], reverse=True)
```

ConstraintSystem Class

ConstraintSystem class coordinates constraint evaluation across multiple reasoners:

```
class ConstraintSystem:
    """Orchestrates constraint evaluation with multiple reasoners"""

    def validate_all(self, context):
        """Validate all constraints with appropriate reasoners"""
        results = {}

        for constraint in self.constraints:
            reasoner_type = self._determine_reasoner_type(constraint)
            reasoner = Reasoner.get_reasoner(reasoner_type)()
            results[constraint.name] = reasoner.evaluate(constraint, context)

        return results
```

5.2. Neural Module Classes

NeuralComponent class

NeuralComponent class provides the base for constraint-aware neural models:

```
class NeuralComponent(COHObject, nn.Module):
    """Base class for neural components with constraint awareness"""

    def train_component(self, dataset, constraints=None):
        """Train with constraint penalty terms"""
        if constraints:
            loss_fn = self._build_constrained_loss(constraints)
        else:
            loss_fn = nn.CrossEntropyLoss()

        # Training loop with constraint monitoring
        for epoch in range(epochs):
            for batch in dataset:
                loss = self._train_step(batch, loss_fn)
                self._check_training_constraints(loss)
```

MultiModalEmbedder class

MultiModalEmbedder class generates unified embeddings across modalities:

```
class MultiModalEmbedder(EmbeddingModel):
    """Generates embeddings from multiple modalities"""

    def __call__(self, coh_object):
        """Generate unified embedding for COHObject"""
        text_embedding = self.text_encoder(coh_object.description)
```

 doi:10.20944/preprints202601.0558.v2


```
        attribute_embedding = self.attribute_encoder(coh_object.attributes)
        constraint_embedding = self.constraint_encoder(coh_object.constraints)

        # Combine with attention
        combined = self.attention_layer(
            text_embedding,
            attribute_embedding,
            constraint_embedding
        )

        return combined
```

5.3. Reasoners Module Classes

The reasoners module implements a hierarchy of specialized evaluation engines.

Reasoner class

Reasoner class is an abstract class as base for constraint reasoning and resolution.

```
class Reasoner(ABC):
    """Abstract base class for constraint reasoners"""
    _registry = {}  # Global registry of reasoner types

    @classmethod
    def get_reasoner(cls, reasoner_type):
        """Factory method for getting reasoner instances"""
        return cls._registry.get(reasoner_type, BaseReasoner)

    @abstractmethod
    def evaluate(self, constraint, context):
        """Evaluate constraint in given context"""
        pass

    @abstractmethod
    def attempt_resolution(self, constraint, context):
        """Attempt to resolve constraint violation"""
        pass
```

BiologicalReasoner class

BiologicalReasoner handles biomedical constraints:

```
class BiologicalReasoner(BaseReasoner, reasoner_type="biological"):
    """Reasoner for biological and medical constraints"""

    def evaluate(self, constraint, context):
        if "concentration" in constraint.specification:
```

 doi:10.20944/preprints202601.0558.v2


```
            return self._evaluate_concentration(constraint, context)
        elif "growth_rate" in constraint.specification:
            return self._evaluate_growth(constraint, context)
        return super().evaluate(constraint, context)
```

SafetyReasoner class

SafetyReasoner class provides redundant validation for critical systems:

```
class SafetyReasoner(BaseReasoner, reasoner_type="safety"):
    """Redundant validation for safety-critical constraints"""
    def evaluate(self, constraint, context):
        # Primary evaluation
        primary_result = super().evaluate(constraint, context)

        # Redundant validation with different method
        redundant_result = self._redundant_evaluation(constraint, context)

        # Both must agree for safety-critical constraints
        if constraint.critical:
            return primary_result and redundant_result

        return primary_result
```

5.4. NLP Module Classes

COHDialogueManager class

COHDialogueManager class enables constraint-aware conversation:

```
class COHDialogueManager:
    """Manages dialogues with constraint validation"""

    def respond(self, user_input):
        """Generate constraint-aware response"""
        # Recognize intent
        intent = self.intent_recognizer.recognize_intent(user_input)

        # Generate candidate response
        candidate = self.response_generator.generate(intent)

        # Validate against constraints
        validation = self.response_validator.validate(
            candidate,
            self.repository.objects
        )

        if validation['valid']:
            return candidate
```

 doi:10.20944/preprints202601.0558.v2


```
        else:
            # Generate alternative that satisfies constraints
            return self._generate_constraint_compliant_alternative(
                candidate,
                validation['violations']
            )
```

Text2COH class

Text2COH converts documents into COH knowledge structures:

```
class Text2COH:
    """Converts text documents into COHObjects"""

    def process_document(self, text):
        """Convert document to hierarchical COH structure"""
        # Extract knowledge units
        units = self.extractor.extract_knowledge_units(text)

        # Create COHObjects for each unit
        coh_objects = []
        for unit in units:
            obj = COHObject(name=unit['name'])
            obj.description = unit['content']

            # Infer constraints from text
            constraints = self.constraint_parser.parse(unit['constraints'])
            for constraint in constraints:
                obj.add_identity_constraint(constraint)

            coh_objects.append(obj)

        return coh_objects
```

6. Programming Intelligent Systems with GISMOL

We demonstrate GISMOL programming through the healthcare system previously modeled: the Autonomous Diagnostic Assistant.

6.1. Main Class Definition

```
class AutonomousDiagnosticAssistant(COHObject):
    """Healthcare: Autonomous diagnostic system with COH implementation"""

    def __init__(self, name="DiagnosticAssistant"):
        super().__init__(name)

        # 1. Components - hierarchical decomposition
```

 doi:10.20944/preprints202601.0558.v2


```
        self.components = {
            'symptom_analyzer': SymptomAnalyzer(),
            'disease_knowledge_base': DiseaseKnowledgeBase(),
            'treatment_recommender': TreatmentRecommender(),
            'patient_history_manager': PatientHistoryManager()
        }

        # 2. Attributes - state variables
        self.attributes = {
            'diagnostic_confidence': 0.0,
            'current_patient': None,
            'active_differential_diagnosis': [],
            'recommended_tests': [],
            'last_validation_timestamp': None
        }

        # 3. Methods are defined below as class methods

        # 4. Neural Components
        self._initialize_neural_components()

        # 5. Neural Embedding
        self._initialize_embedding_model()

        # 6. Identity Constraints
        self._define_identity_constraints()

        # 7. Trigger Constraints
        self._define_trigger_constraints()

        # 8. Goal Constraints
        self._define_goal_constraints()

        # 9. Constraint Daemons
        self._initialize_daemons()

        # Validate initial state
        self._validate_initial_configuration()
```

6.2. Components Implementation

```
    def _initialize_components(self):
        """Initialize subsystem components"""
```

 doi:10.20944/preprints202601.0558.v2


```
        # Symptom analyzer with pattern recognition
        self.components['symptom_analyzer'] = SymptomAnalyzer()
        self.add_relation(self, self.components['symptom_analyzer'],
                          "coordinates", "diagnostic_pipeline")

        # Disease knowledge base with semantic relationships
        self.components['disease_knowledge_base'] = DiseaseKnowledgeBase()
        self.components['disease_knowledge_base'].load_medical_ontology(
            "snomed_ct"
        )

        # Treatment recommender with constraint-aware planning
        self.components['treatment_recommender'] = TreatmentRecommender()

        # Patient history manager for longitudinal tracking
        self.components['patient_history_manager'] = PatientHistoryManager()

        # Establish component relationships
        self._establish_component_relationships()
```

6.3. Attributes with Validation

```
    @property
    def diagnostic_confidence(self):
        return self.attributes['diagnostic_confidence']

    @diagnostic_confidence.setter
    def diagnostic_confidence(self, value):
        # Validate against constraints before setting
        if not self._validate_value('diagnostic_confidence', value):
            raise ConstraintViolation(
                f"Diagnostic confidence {value} violates constraints"
            )
        self.attributes['diagnostic_confidence'] = value
        self._notify_daemons('attribute_change', 'diagnostic_confidence')
```

6.4.. Methods with Constraint Checking

```
    def diagnose_patient(self, patient_data, symptoms):
        """Main diagnostic method with full constraint validation"""
        # Pre-condition validation
        preconditions = self._check_diagnostic_preconditions(patient_data)
        if not preconditions['met']:
            return {
                'success': False,
```

 doi:10.20944/preprints202601.0558.v2


```
            'error': 'Preconditions not met',
            'violations': preconditions['violations']
        }

    # Step 1: Symptom analysis using neural component
    symptom_embedding = self.get_neural_component(
        'symptom_encoder'
    ).encode(symptoms)

    # Step 2: Retrieve similar cases from knowledge base
    similar_cases = self.components['disease_knowledge_base'].query(
        symptom_embedding,
        similarity_threshold=0.7
    )

    # Step 3: Generate differential diagnosis
    differential = self._generate_differential_diagnosis(
        symptoms,
        similar_cases
    )

    # Step 4: Validate against medical constraints
    validation = self._validate_diagnosis(differential)

    # Step 5: If validation fails, attempt resolution
    if not validation['valid']:
        resolved = self._attempt_diagnostic_resolution(
            differential,
            validation['violations']
        )
        if resolved:
            differential = resolved
        else:
            # Escalate to human expert
            self._trigger_human_review(differential, validation)

    # Update attributes
    self.attributes['active_differential_diagnosis'] = differential
    self.attributes['diagnostic_confidence'] = self._calculate_confidence(
        differential
    )
```

 doi:10.20944/preprints202601.0558.v2


```
        # Post-condition validation
        self._validate_post_diagnosis_state()

        return {
            'success': True,
            'diagnosis': differential,
            'confidence': self.attributes['diagnostic_confidence'],
            'recommended_tests': self.attributes['recommended_tests']
        }
```

6.5. Neural Components Integration

```
    def _initialize_neural_components(self):
        """Initialize neural components for diagnostic assistance"""

        # Multi-modal symptom encoder
        self.add_neural_component(
            "symptom_encoder",
            MultiModalEmbedder(
                name="symptom_encoder",
                text_dim=512,
                clinical_dim=256,
                output_dim=128
            )
        )

        # Bayesian diagnostic classifier with uncertainty estimation
        self.add_neural_component(
            "diagnostic_classifier",
            BayesianClassifier(
                name="disease_classifier",
                input_dim=128,
                output_dim=2000,  # ICD-10 codes
                num_mc_samples=50,  # Monte Carlo samples for uncertainty
                constraints=[
                    {
                        'name': 'clinical_confidence',
                        'specification': 'prediction_confidence >= 0.65',
                        'priority': 'HIGH'
                    }
                ]
            )
        )
```

 doi:10.20944/preprints202601.0558.v2


```
        # Treatment outcome predictor
        self.add_neural_component(
            "outcome_predictor",
            NeuralTemporalModel(
                name="treatment_outcome",
                input_dim=256,
                hidden_dim=512,
                prediction_horizon=30  # 30-day outcomes
            )
        )
```

6.6. Neural Embedding for Medical Concepts

```
    def _initialize_embedding_model(self):
        """Initialize medical concept embedding model"""
        self.add_neural_component(
            "medical_embedder",
            MedicalConceptEmbedder(
                name="clinical_embedder",
                knowledge_sources=['SNOMED-CT', 'ICD-10', 'UMLS'],
                embedding_dim=256,
                constraint_preservation=True
            ),
            is_embedding_model=True
        )

        # Pre-train with medical corpus if needed
        if not self.get_neural_component("medical_embedder").is_trained:
            medical_corpus = self._load_medical_corpus()
            self.get_neural_component("medical_embedder").train(
                medical_corpus,
                epochs=50,
                constraint_weight=0.1
            )
```

6.7. Identity Constraints Definition

```
    def _define_identity_constraints(self):
        """Define fundamental medical and ethical constraints"""

        # Medical accuracy constraints
        self.add_identity_constraint({
            'name': 'diagnostic_accuracy',
            'specification': 'diagnostic_confidence >= 0.7 or escalate_to_human',
```

 doi:10.20944/preprints202601.0558.v2


```
            'category': 'medical',
            'severity': 9,
            'reasoner': 'probabilistic'
        })

        # Ethical constraints
        self.add_identity_constraint({
            'name': 'patient_privacy',
            'specification': 'patient_data.encrypted == True and access_logged ==
True',
            'category': 'ethical',
            'severity': 10,
            'reasoner': 'general'
        })

        # Clinical guideline constraints
        self.add_identity_constraint({
            'name': 'guideline_compliance',
            'specification':     'treatment     in     approved_guidelines     or
justification_provided',
            'category': 'regulatory',
            'severity': 8,
            'reasoner': 'biological'
        })

        # Safety constraints
        self.add_identity_constraint({
            'name': 'drug_interaction_check',
            'specification': 'check_interactions(prescribed_drugs) == []',
            'category': 'safety',
            'severity': 10,
            'critical': True
        })
```

6.8. Trigger Constraints for Reactive Behavior

```
    def _define_trigger_constraints(self):
        """Define event-driven medical responses"""

        # Emergency response trigger
        self.add_trigger_constraint({
            'name': 'critical_finding',
            'type': 'emergency',
```

 doi:10.20944/preprints202601.0558.v2


```
        'event': 'symptom_severity > 8 or vital_signs_abnormal',
        'condition': 'patient_location_known == True',
        'action': 'alert_emergency_team',
        'postcondition': 'emergency_response_initiated within 5min',
        'critical': True,
        'reasoner': 'temporal'
    })

    # Test result follow-up trigger
    self.add_trigger_constraint({
        'name': 'abnormal_result',
        'type': 'followup',
        'event': 'lab_result.out_of_range == True',
        'condition': 'patient_contactable == True',
        'action': 'schedule_followup',
        'postcondition': 'followup_scheduled within 24h',
        'reasoner': 'temporal'
    })

    # Treatment adjustment trigger
    self.add_trigger_constraint({
        'name': 'ineffective_treatment',
        'type': 'therapeutic',
        'event': 'symptom_improvement < 0.3 after 48h treatment',
        'condition': 'alternative_treatments_available == True',
        'action': 'adjust_treatment_plan',
        'postcondition': 'new_plan_generated and patient_informed',
        'reasoner': 'causal'
    })
```

6.9. Goal Constraints for Optimization

```
def _define_goal_constraints(self):
    """Define optimization objectives for diagnostic system"""

    # Diagnostic accuracy goal
    self.add_goal_constraint({
        'name': 'maximize_accuracy',
        'type': 'optimization',
        'objective': 'MAXIMIZE diagnostic_accuracy',
        'subject_to': [
            'false_positive_rate < 0.05',
            'false_negative_rate < 0.02',
```

 doi:10.20944/preprints202601.0558.v2


```
            'decision_time < 300 seconds'
        ],
        'priority': 'HIGH',
        'reasoner': 'goal'
    })

    # Resource efficiency goal
    self.add_goal_constraint({
        'name': 'minimize_testing',
        'type': 'efficiency',
        'objective': 'MINIMIZE unnecessary_tests',
        'subject_to': [
            'diagnostic_confidence >= 0.7',
            'patient_cost_burden < threshold'
        ],
        'priority': 'MEDIUM',
        'reasoner': 'resource'
    })

    # Treatment outcome goal
    self.add_goal_constraint({
        'name': 'optimize_outcomes',
        'type': 'clinical',
        'objective': 'MAXIMIZE 30_day_positive_outcome',
        'subject_to': [
            'treatment_side_effects < acceptable_level',
            'patient_adherence > 0.8'
        ],
        'priority': 'HIGH',
        'reasoner': 'probabilistic'
    })
```

6.10. Constraint Daemons for Continuous Monitoring

```
    def _initialize_daemons(self):
        """Initialize medical constraint monitoring daemons"""

        # Patient safety daemon (continuous monitoring)
        self.daemons['patient_safety'] = MedicalSafetyDaemon(
            monitoring_interval=0.5,  # Check every 500ms
            vital_sign_monitors=['heart_rate', 'blood_pressure',
'oxygen_saturation'],
            alert_channels=['visual', 'auditory', 'pager'],
```

 doi:10.20944/preprints202601.0558.v2


```
            escalation_protocol=self._safety_escalation_protocol
        )

        # Diagnostic quality daemon
        self.daemons['diagnostic_quality'] = QualityAssuranceDaemon(
            evaluation_interval=60.0,  # Check every minute
            quality_metrics=['accuracy', 'timeliness', 'completeness'],
            feedback_mechanism=self._quality_feedback_loop
        )

        # Ethical compliance daemon
        self.daemons['ethical_compliance'] = EthicalComplianceDaemon(
            audit_interval=3600.0,  # Audit every hour
            compliance_checks=['privacy', 'consent', 'bias_detection'],
            reporting_system=self._compliance_reporting
        )

        # Start all daemons
        for daemon_name, daemon in self.daemons.items():
            daemon.start()
            logger.info(f"Started {daemon_name} daemon")
```

6.11. Object Instantiation and Usage

```
# Instantiate the diagnostic assistant
diagnostic_assistant = AutonomousDiagnosticAssistant(
    name="HospitalDiagnosticAI"
)

# Initialize with hospital-specific configuration
diagnostic_assistant.initialize_with_hospital(
    hospital_id="HOSP_123",
    department="Emergency",
    access_level="Physician"
)

# Process a patient case
patient_data = {
    'id': 'PAT_789',
    'age': 45,
    'gender': 'F',
    'vitals': {
        'bp': '140/90',
```

 doi:10.20944/preprints202601.0558.v2


```
        'hr': 110,
        'temp': 38.5
    },
    'symptoms': ['chest_pain', 'shortness_of_breath', 'nausea'],
    'history': ['hypertension', 'smoker']
}

# Run diagnostic process
result = diagnostic_assistant.diagnose_patient(
    patient_data=patient_data,
    symptoms=patient_data['symptoms']
)

if result['success']:
    print(f"Diagnosis: {result['diagnosis']}")
    print(f"Confidence: {result['confidence']:.2f}")
    print(f"Recommended tests: {result['recommended_tests']}")

    # Check constraint satisfaction
    validation = diagnostic_assistant.validate_current_state()
    if validation['all_satisfied']:
        print("All constraints satisfied")
    else:
        print(f"Constraint violations: {validation['violations']}")
else:
    print(f"Diagnosis failed: {result['error']}")
```

This complete implementation demonstrates how GISMOL enables programming of complex intelligent systems with built-in constraint management, neural integration, and hierarchical organization. The Autonomous Diagnostic Assistant showcases all elements of the COH 9-tuple in practice.

7. Modelling and Programming Examples

The following examples illustrate how various intelligent systems can be modelled with COH, and programmed with GISMOL into native Python applications. Because the supporting GISMOL library is still under development and yet to be released to the public, the GISMOL implementations are only meant to illustrate the semantics of the COH models of each intelligent system. If anyone wants to implement any of these intelligent systems with GISMOL, please contact the author directly.

7.1. Smart Manufacturing: Adaptive Production Orchestrator

System Description and Significance: Modern manufacturing requires real-time adaptation to disruptions while maintaining efficiency and quality. Traditional systems struggle with dynamic optimization across multiple objectives.

System Requirements: Real-time scheduling, quality monitoring, resource allocation, predictive maintenance, and human-robot collaboration.

 doi:10.20944/preprints202601.0558.v2

COH-based Design: The orchestrator coordinates multiple production cells as hierarchical components, each with local optimization and global coordination through constraint propagation.

Formal 9-tuple Model:

- C: Production cells, material handlers, quality stations
- A: OEE, quality metrics, energy consumption
- M: Dynamic scheduling, maintenance routines
- N: RL scheduler, anomaly detectors
- E: Machine behavior embeddings
- I: Physical laws, safety protocols
- T: Quality deviation triggers, maintenance alerts
- G: Maximize throughput, minimize energy, maintain quality
- D: Safety monitoring, constraint enforcement

GISMOL Implementation:

```
class AdaptiveProductionOrchestrator(COHObject):
    def __init__(self, name="ProductionOrchestrator"):
        super().__init__(name)
        # Hierarchical decomposition
        self.components = {
            'scheduler': NeuralSchedulingModel(),
            'quality_controller': QualityController(),
            'resource_manager': ResourceManager()
        }

        # Neural components for adaptation
        self.add_neural_component(
            "dynamic_scheduler",
            NeuralSchedulingModel(
                task_dim=64,
                resource_dim=32,
                hidden_dim=256
            )
        )

        # Production constraints
        self.add_identity_constraint({
            'name': 'throughput_target',
            'specification': 'units_per_hour >= 1000',
            'reasoner': 'resource'
        })

        # Real-time trigger for disruptions
        self.add_trigger_constraint({
            'name': 'machine_failure',
            'event': 'machine_status == "failed"',
```

 doi:10.20944/preprints202601.0558.v2


```
        'action': 'reallocate_tasks',
        'postcondition': 'throughput_loss < 5%'
    })
```

Constraint Propagation: Quality deviations trigger rescheduling while maintaining overall throughput goals. Neural predictions inform constraint adjustments.

Comparative Analysis: Unlike traditional Manufacturing Execution Systems (MES), this system dynamically adapts using neural predictions while maintaining hard constraints through reasoners.

7.2. Social Science: Policy Impact Simulator

Description and Significance: Social policy analysis requires modeling complex interactions between demographic groups, institutions, and economic factors with deep uncertainty.

System Requirements: Multi-agent simulation, policy outcome prediction, stakeholder preference integration, uncertainty quantification.

COH-based Design: Agents as COHObjects with behavioral constraints, institutions as hierarchical structures, policies as constraint modifications.

Formal 9-tuple Model:

- C: Demographic groups, institutions, social networks
- A: Socioeconomic indicators, opinion distributions
- M: Policy implementation, communication campaigns
- N: Agent-based models, graph neural networks
- E: Policy space embeddings, cultural dimensions
- I: Population conservation, budget constraints
- T: Inequality triggers, trust thresholds
- G: Maximize social welfare, minimize unintended consequences
- D: Democratic participation monitoring

GISMOL Implementation:

```
class PolicyImpactSimulator(COHObject):
    def simulate_policy(self, policy, steps=100):
        """Run policy simulation with constraint validation"""
        # Initialize agent populations
        agents = self._create_agent_population()

        # Apply policy as constraint modifications
        modified_constraints = self._policy_to_constraints(policy)

        # Run simulation with neural behavior models
        for step in range(steps):
            # Agent decisions using neural components
            decisions = self.get_neural_component(
                'agent_model'
            ).predict(agents, self.context)

            # Update with constraint satisfaction
            self._update_with_constraints(agents, decisions)

            # Monitor goal progress
```

 doi:10.20944/preprints202601.0558.v2


```
        self.daemons['goal'].evaluate_progress()

    return self._collect_outcomes()
```

Neural Component Interactions: Graph neural networks model social network effects while Bayesian neural networks quantify outcome uncertainty.

Constraint Resolution: When policies violate ethical constraints, the system generates alternatives using counterfactual reasoning with constraint preservation.

7.3. Psychology: Personalized Mental Health Companion

Description and Significance: Mental health support requires personalized, adaptive interventions that respect individual differences while maintaining therapeutic efficacy.

System Requirements: Mood tracking, intervention personalization, crisis detection, therapeutic relationship maintenance, cultural adaptation.

COH-based Design: Psychological constructs as hierarchical components, therapeutic techniques as methods with efficacy constraints, client state as attributes.

Formal 9-tuple Model:

- C: Cognitive modules, coping strategies, therapeutic techniques
- A: Mood states, cognitive load, sleep quality
- M: Cognitive restructuring, mindfulness practices
- N: Few-shot learning models, emotion recognition
- E: Psychological state embeddings, therapeutic technique embeddings
- I: Therapeutic boundaries, client autonomy
- T: Suicidal ideation detection, progress stagnation
- G: Maximize well-being, build resilience, personalize interventions
- D: Relationship boundary monitoring, risk protocol enforcement

GISMOL Implementation:

```
class MentalHealthCompanion(COHObject):
    def __init__(self, client_profile):
        super().__init__(name=f"Companion_{client_profile['id']}")

        # Personalize based on client
        self._personalize_components(client_profile)

        # Ethical constraints as identity constraints
        self.add_identity_constraint({
            'name': 'client_autonomy',
            'specification': 'never override_client_choice',
            'critical': True
        })

        # Crisis detection trigger
        self.add_trigger_constraint({
            'name': 'crisis_alert',
            'event': 'suicidal_ideation_detected',
            'action': 'escalate_to_human',
            'postcondition': 'human_contact_established within 15min'
```

 doi:10.20944/preprints202601.0558.v2


```
})
```

Hierarchical Decomposition: Therapy decomposed into techniques, each with efficacy constraints. Progress tracking at multiple time scales.

Comparative Analysis: Unlike rule-based therapy apps, this system adapts using neural models while maintaining ethical constraints through daemon monitoring.

7.4. Finance: Adaptive Portfolio Management System

Description and Significance: Financial markets require balancing risk and return across multiple time horizons while adapting to regime changes.

System Requirements: Risk management, portfolio optimization, market regime detection, transaction cost minimization, regulatory compliance.

COH-based Design: Assets as components with risk constraints, strategies as methods with performance goals, market states as attributes with uncertainty.

Formal 9-tuple Model:

- C: Asset classes, investment strategies, risk factors
- A: Returns, volatility, correlations, liquidity
- M: Rebalancing, hedging, execution
- N: Regime detection networks, attention mechanisms
- E: Asset embeddings, market regime embeddings
- I: Portfolio sum constraints, regulatory requirements
- T: Risk limit breaches, liquidity drops
- G: Maximize risk-adjusted returns, minimize drawdowns
- D: Regulatory compliance monitoring, mandate validation

GISMOL Implementation:

```
class AdaptivePortfolioManager(COHObject):
    def rebalance_portfolio(self, market_data):
        """Constraint-aware portfolio rebalancing"""
        # Predict regime with neural component
        regime = self.get_neural_component(
            'regime_detector'
        ).predict(market_data)

        # Adjust constraints based on regime
        self._adjust_risk_constraints(regime)

        # Optimize with constraint-aware optimizer
        optimizer = ConstraintAwareOptimizer(
            constraints=self._get_all_constraints(),
            penalty_weight=0.1
        )

        # Execute with transaction cost constraints
        return self._execute_rebalancing(optimizer.solution)
```

Constraint Propagation: Risk limits propagate through portfolio hierarchy. Regulatory changes trigger constraint updates across all components.

 doi:10.20944/preprints202601.0558.v2

Neural Integration: Attention mechanisms identify cross-asset relationships while Bayesian networks quantify prediction uncertainty for risk management.

7.5. Governance: Participatory Policy Co-creation Platform

Description and Significance: Democratic governance requires integrating diverse perspectives into coherent policies while maintaining procedural fairness.

System Requirements: Stakeholder engagement, deliberation facilitation, policy impact assessment, consensus detection, transparency maintenance.

COH-based Design: Stakeholders as components with preference constraints, policies as objects with impact constraints, deliberation processes as methods with fairness constraints.

Formal 9-tuple Model:

- C: Citizen groups, government agencies, expert panels
- A: Opinion distributions, trust measures, feasibility scores
- M: Deliberative polling, participatory budgeting
- N: Argument analysis models, consensus detection
- E: Policy space embeddings, stakeholder position embeddings
- I: Constitutional principles, inclusion requirements
- T: Polarization thresholds, deadlock detection
- G: Maximize legitimacy, minimize decision latency
- D: Democratic equality monitoring, procedural fairness validation

GISMOL Implementation:

```
class PolicyCoCreationPlatform(COHObject):
    def facilitate_deliberation(self, policy_draft, stakeholders):
        """Facilitate deliberation with constraint preservation"""
        # Analyze arguments with NLP
        arguments = self.get_neural_component(
            'argument_analyzer'
        ).process(stakeholders.comments)

        # Detect consensus areas
        consensus = self._detect_consensus(arguments)

        # Generate revised policy satisfying constraints
        revised = self._revise_policy(
            policy_draft,
            consensus,
            self._get_all_constraints()
        )

        # Validate against democratic constraints
        validation = self.daemons['democratic'].validate(revised)

        return revised, validation
```

Goal Mechanisms: Multiple competing objectives (legitimacy, efficiency, inclusion) balanced through Pareto optimization with constraint satisfaction.

 doi:10.20944/preprints202601.0558.v2

Comparative Analysis: Unlike simple voting systems, this platform maintains deliberation quality constraints and detects emergent consensus through neural analysis.

Due to space limitations, the implementations in the examples above are incomplete. Please refer to the tutorial in the supplementary material for more detailed guidance, including a complete implementation of a smart home intelligent system with COH/GISMOL.

8. Summary of COH/GISMOL Advantages

8.1. Comparison with Existing Frameworks

Table 2 provides a preliminary, qualitative comparison with existing AI frameworks.

Table 2. Comparison with Existing AI Frameworks.

Framework	Type	Strengths	Limitations	GISMOL (anticipated)
SOAR [9]	Cognitive architecture	Symbolic reasoning, goal-driven	Limited learning, no neural integration	Neuro-symbolic integration, constraint system
ACT-R [10]	Cognitive architecture	Cognitive modeling, production rules	Complex implementation, limited scalability	Python implementation, hierarchical organization
TensorLog [12]	Neuro-symbolic	Differentiable inference, probabilistic	Limited constraint types, no real-time monitoring	Comprehensive constraint system, daemon monitoring
DeepProbLog [16]	Neuro-symbolic	Probabilistic logic, neural networks	No hierarchical constraints, limited domains	Multi-domain constraints, hierarchical organization
PyTorch/TensorFlow	Deep learning	Neural network flexibility, GPU acceleration	No symbolic reasoning, black-box nature	Symbolic constraint integration, explainability
CLIPS [31]	Expert system	Rule-based reasoning, pattern matching	No learning capabilities, static knowledge	Adaptive neural components, continuous learning

8.2. Unique Contributions of COH/GISMOL

1. Integrated Constraint System: Combines identity, trigger, and goal constraints with neural components.
2. Hierarchical Organization: Natural decomposition of complex systems while maintaining coherence.
3. Multi-domain Reasoning: Unified handling of biological, physical, temporal, and other constraints.
4. Real-time Monitoring: Constraint daemons provide continuous safety guarantees.
5. Practical Implementation (prototype): Python library enables rapid prototyping of intelligent systems.
6. Cross-domain Applicability: Single framework applicable to healthcare, manufacturing, finance, etc.

8.3. Limitations

1. Learning Curve: Requires understanding of both symbolic and neural approaches.
2. Computational Overhead: Constraint monitoring adds runtime overhead.
3. Domain Knowledge Requirement: System designers must specify appropriate constraints.

 doi:10.20944/preprints202601.0558.v2

4. Early Development Stage: Limited real-world deployment compared to mature frameworks; API subject to change.

9. Conclusion and Future Research Directions

9.1. Summary

GISMOL provides both a theoretical framework (COH) and a prototype Python library for developing intelligent systems. By integrating neural learning with symbolic constraints in a hierarchical organization, GISMOL addresses key challenges in current AI approaches. The case studies demonstrate GISMOL's potential across healthcare, manufacturing, logistics, finance, governance, and education domains.

9.2. Future Research Directions

1. Scalability Optimization: Improving performance for large constraint systems.
2. Automated Constraint Learning: Learning constraints from data rather than manual specification.
3. Formal Verification: Mathematical proofs of constraint satisfaction under certain conditions.
4. Cognitive Science Validation: Testing COH models against human cognitive performance.
5. Distributed Implementation: Scaling GISMOL systems across multiple computing nodes.
6. Quantum Integration: Exploring quantum computing for constraint satisfaction problems.
7. Ethical Constraint Formalization: Developing frameworks for encoding ethical principles.
8. Cross-modal Learning: Integrating vision, language, and other modalities more seamlessly.
9. Lifelong Learning: Systems that accumulate knowledge over extended periods.
10. Human-AI Collaboration: Improved interfaces for human guidance of GISMOL systems.

9.3. Concluding Remarks

COH/GISMOL represents a promising step toward practical AGI research by providing a unified framework that combines the strengths of neural and symbolic approaches. While challenges remain, the COH theory and evolving GISMOL implementation offer a path forward for creating intelligent systems that are capable, safe, and adaptable across multiple domains.

References

1. J. R. Anderson, C. Lebiere, et al., "The Atomic Components of Thought," Psychology Press, 1998.
2. B. Goertzel, "Artificial general intelligence: Concept, state of the art, and future prospects," Journal of Artificial General Intelligence, vol. 5, no. 1, pp. 1-46, 2014.
3. G. Marcus, "The next decade in AI: Four steps towards robust artificial intelligence," arXiv preprint arXiv:2002.06177, 2020.
4. C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," Nature Machine Intelligence, vol. 1, no. 5, pp. 206-215, 2019.
5. H. J. Levesque, "Common sense, the Turing test, and the quest for real AI," MIT Press, 2017.
6. A. S. d'Avila Garcez and L. C. Lamb, "Neurosymbolic AI: The 3rd wave," Artificial Intelligence Review, vol. 53, pp. 1-20, 2020.
7. M. Garnelo and M. Shanahan, "Reconciling deep learning with symbolic artificial intelligence: representing objects and relations," Current Opinion in Behavioral Sciences, vol. 29, pp. 17-23, 2019.
8. T. R. Besold et al., "Neural-symbolic learning and reasoning: A survey and interpretation," arXiv preprint arXiv:1711.03902, 2017.
9. J. E. Laird, "The SOAR cognitive architecture," MIT Press, 2012.
10. P. Langley, "An integrative framework for artificial intelligence," Journal of Artificial General Intelligence, vol. 10, no. 1, pp. 1-8, 2019.
11. P. Langley, "The cognitive systems paradigm," Advances in Cognitive Systems, vol. 1, pp. 3-13, 2012.

 doi:10.20944/preprints202601.0558.v2

12. L. De Raedt et al., "From statistical relational to neuro-symbolic artificial intelligence," Artificial Intelligence, vol. 287, 2020.
13. Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature, vol. 521, no. 7553, pp. 436-444, 2015.
14. V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529-533, 2015.
15. C. Molnar, "Interpretable Machine Learning: A Guide for Making Black Box Models Explainable," 2020.
16. R. Manhaeve et al., "DeepProbLog: Neural probabilistic logic programming," Advances in Neural Information Processing Systems, vol. 31, pp. 3749-3759, 2018.
17. T. Rocktäschel and S. Riedel, "End-to-end differentiable proving," Advances in Neural Information Processing Systems, vol. 30, pp. 3788-3800, 2017.
18. K. Yi et al., "Neural-symbolic VQA: Disentangling reasoning from vision and language understanding," Advances in Neural Information Processing Systems, vol. 31, pp. 1031-1042, 2018.
19. G. H. Chen, "A survey on hierarchical deep learning," IEEE Access, vol. 8, pp. 68712-68722, 2020.
20. P. S. Rosenbloom, "The Sigma cognitive architecture and system," International Journal of Machine Learning and Cybernetics, vol. 10, pp. 147-169, 2019.
21. P. Langley et al., "Cognitive architectures: Research issues and challenges," Cognitive Systems Research, vol. 56, pp. 1-10, 2019.
22. F. Rossi, P. van Beek, and T. Walsh, "Handbook of constraint programming," Elsevier, 2006.
23. R. Dechter, "Constraint processing," Morgan Kaufmann, 2003.
24. D. Hassabis et al., "Neuroscience-inspired artificial intelligence," Neuron, vol. 95, no. 2, pp. 245-258, 2017.
25. Y. Bengio, "The consciousness prior," arXiv preprint arXiv:1709.08568, 2017.
26. J. Hawkins et al., "A framework for intelligence and cortical function based on grid cells in the neocortex," Frontiers in Neural Circuits, vol. 13, 2019.
27. M. Leucker and C. Schallhart, "A brief account of runtime verification," Journal of Logic and Algebraic Programming, vol. 78, no. 5, pp. 293-303, 2009.
28. M. A. Makary and M. Daniel, "Medical error—the third leading cause of death in the US," BMJ, vol. 353, 2016.
29. A. Rajkomar et al., "Scalable and accurate deep learning with electronic health records," NPJ Digital Medicine, vol. 1, no. 18, 2018.
30. Z. Li et al., "Deep learning for smart manufacturing: Methods and applications," Journal of Manufacturing Systems, vol. 48, pp. 144-156, 2018.
31. M. R. G. Raman et al., "Explainable AI: A review of machine learning interpretability methods," Entropy, vol. 23, no. 1, 2021.