Print PDF-Friendly Version

AUTONOMOUS AGENT SPECIFICATION

Arthur M. Collé

TABLE OF CONTENTS

- 1. Introduction
  - 1.1. Motivation and Objectives
  - 1.2. Overview of OORL in Mutable Ontologies
- 2. Object Schema
  - 2.1. Definition of Objects and their Attributes
  - 2.2. Object Representation and Parameterization
- 3. Self-Reflective Meta-DSL
  - 3.1. Definition and Purpose
  - 3.2. Core Constructs
  - 3.3. Initialization, Learning, and Refinement Functions
- 4. Object Interactions
  - 4.1. Interaction Dyads ($\mathcal{I}$)
  - 4.2. Message-Passing Protocol ($\mathcal{M}$)
- 5. Schema Evolution
  - 5.1. Schema Configurations ($\mathcal{S}$)
  - 5.2. Schema Evolution Process
  - 5.3. Markov Process over Schema States
  - 5.4. Schema Evolution with Self-Modification
- 6. Reward Learning
  - 6.1. Reward Objects ($\mathbb{R}$)
  - 6.2. Reward Functions
  - 6.3. Reward Learning Algorithms
  - 6.4. Combining Extrinsic and Intrinsic Rewards
- 7. Policy Optimization
  - 7.1. Policy Representation
  - 7.2. Policy Gradient Methods
  - 7.3. Value Function Approximation
  - 7.4. Meta-Learning
  - 7.6. Monte Carlo Tree Search (MCTS) with Q* Optimal Policy and Self-Reflective Reasoning
  - 7.7. Monte Carlo Graph Search (MCGS) with Contrastive Learning and Graph Attention
- 8. Open-Ended Learning via Object Rewriting
  - 8.1. Formalizing Self-Expanding Action Spaces
  - 8.2 User Intent Parsing and Graph Exploration
  - 8.3. Agent Output and Reasoning
  - 8.4 Meta-DSL Execution
  - 8.5. Reward & Policy Optimization
  - 8.6. Open-Endedness and Exploration
  - 8.7. Practical Considerations & Scalability
- 9. Hierarchical Object Composition
  - 9.1. Object Aggregation and Decomposition
  - 9.2. Hierarchical Planning
- 10. Object-Oriented Exploration
  - 10.1. Novelty-Based Exploration
  - 10.2. Uncertainty-Based Exploration
  - 10.3. Exploration Bonuses in OORL
- 11. Object-Oriented Transfer Learning
  - 11.1 Object Similarity and Embedding Spaces
  - 11.2. Analogical Reasoning
  - 11.3. Meta-Learning
- 12. Object-Oriented Curriculum Learning
  - 12.1 Task Difficulty Scoring
  - 12.2 Task Sampling and Curriculum Generation

- 13. Object-Oriented Intrinsic Motivation
  - 13.1 Empowerment
  - 13.2 Curiosity
  - 13.3. Intrinsic Rewards in OORL
- 14. Challenges and Future Directions
  - 14.1. Representational and Computational Complexity
  - 14.2. Scalability and Efficiency of Learning Algorithms
  - 14.3. Emergence of Abstract and Transferable Concepts
  - 14.4. Safety, Robustness, and Alignment of Self-Reflective Agents
  - 14.5. Integration with Other Cognitive Faculties
- 15. Conclusion
  - 15.1 Summary of Key Ideas and Contributions
  - 15.2 Potential Impact and Applications
  - 15.3 Outlook and Future Work
  - 15.4. Conclusion

1. INTRODUCTION

🙞 1.1. Motivation and Objectives

Reinforcement learning has made significant strides in enabling agents to learn complex behaviors through interaction with their environment. However, traditional approaches often struggle in open-ended, dynamic environments where the optimal behavior and relevant features may change over time.

To address these challenges, we propose Object-Oriented Reinforcement Learning (OORL) in Mutable Ontologies - a novel framework that combines ideas from object-oriented programming, graph theory, and meta-learning to enable more flexible, adaptive, and open-ended learning. The main objectives are:

1. Represent the learning process as a multi-agent interaction between evolving objects, each with its own state, behavior, and goals.
2. Enable the agent to discover and adapt its own representation of the environment through dynamic formation and dissolution of object relationships.
3. Introduce a self-reflective meta-language that allows objects to reason about and modify their own learning process.
4. Support open-ended learning through continuous exploration and expansion of the space of possible behaviors and representations.

🙞 1.2. Overview of OORL in Mutable Ontologies

The key idea is to model the environment as a graph of interacting objects, where each object represents a particular aspect of the state space and is equipped with its own learning mechanism. Objects engage in goal-directed behavior by exchanging messages according to a dynamic protocol that evolves over time. The global behavior emerges from the local interactions and adaptations of the individual objects.

A central component is the self-reflective meta-DSL (domain-specific language) that enables objects to inspect and modify their own internal structure and behavior. This allows for a form of "learning to learn" where objects can adapt their own learning strategies based on experience.

The learning process itself is formulated as a multi-objective optimization problem, where the agent seeks to maximize a combination of external rewards and intrinsic motivations, such as empowerment and curiosity. This is achieved through a combination of policy gradients, value function approximation, and meta-learning techniques.

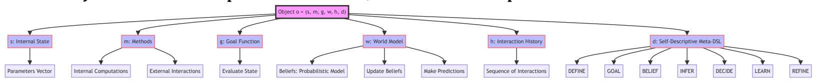
The following sections provide a detailed mathematical formulation of the OORL framework, covering the key components of object schema, interactions, schema evolution, reward learning, policy optimization, open-ended learning, and various extensions and applications.

2. OBJECT SCHEMA

2.1. Definition of Objects and their Attributes

In the OORL framework, the environment is modeled as a set of objects $\mathcal{O}$, where each object $o \in \mathcal{O}$ is a tuple $o = (s, m, g, w, h, d)$ consisting of the following attributes:

- s: The object's internal state, represented by a set of parameters.
- m: The set of methods or functions that the object can perform.
- g: The object's goal or objective function, specifying its desired outcomes.
- w: The object's world model, representing its beliefs and assumptions about the environment.
- h: The object's interaction history, storing a record of its past exchanges with other objects.
- d: The object's self-descriptive meta-DSL, used for introspection and self-modification.



2.1.1. Internal State (s)

The internal state s represents the object's current configuration and can be thought of as a vector of parameter values. The specific structure and semantics of the state space depend on the type and function of the object.

For example, a sensor object might have a state representing the current readings of its input channels, while an actuator object might have a state representing the current positions of its joints.

2.1.2. Methods/Functions (m)

The methods m define the set of actions or transformations that the object can perform. These can include both internal computations (e.g. updating the state based on new observations) and external interactions (e.g. sending a message to another object).

Methods are typically parameterized by the object's current state and any arguments provided by the caller.

2.1.3. Goal or Objective Function (g)

The goal g specifies the object's desired outcomes or objectives. This can be represented as a scalar value function that assigns a score to each possible state, or as a more complex objective function that takes into account multiple criteria.

The goal is used to guide the object's behavior and learning, by providing a measure of the desirability of different actions and outcomes.

2.1.4. World Model (w)

The world model w represents the object's beliefs and assumptions about the environment, including the states and behaviors of other objects.

This can be represented as a probabilistic graphical model, such as a Bayesian network or a Markov decision process, that captures the dependencies and uncertainties in the environment. The world model is used to make predictions and inform decision-making, and is updated based on the object's observations and interactions.

2.1.5. Interaction History (h)

The interaction history h stores a record of the object's past exchanges with other objects, including the messages sent and received, actions taken, and rewards or feedback obtained.

This information is used for learning and adaptation, by providing data for updating the object's world model, goal, and behavior. The interaction history can be represented as a time-indexed sequence of tuples, or as a more compact summary statistic.

2.1.6. Self-Descriptive Meta-DSL (d)

The self-descriptive meta-DSL *d* is a domain-specific language that enables introspection and self-modification. It provides primitives and constructs for querying and manipulating the object's own attributes, such as its state, methods, goal, world model, and interaction history.

The meta-DSL is used to implement meta-learning algorithms that can adapt the object's learning strategy based on experience. Examples of meta-DSL constructs include:

- DEFINE: Defines new attributes, methods, or sub-objects.
- GOAL: Specifies or modifies the object's objective function.
- BELIEF: Represents a probabilistic belief or assumption about the environment.
- INFER: Performs inference on the object's beliefs to update its world model.
- DECIDE: Selects an action or plan based on current state, goal, and beliefs.
- LEARN: Updates knowledge and strategies based on new observations or feedback.
- REFINE: Modifies attributes or methods based on meta-level reasoning.

2.2. Object Representation and Parameterization

To enable efficient learning and reasoning, objects in the OORL framework are represented using parameterized models that capture their key attributes and relationships. The specific choice of representation depends on the type and complexity of the object, but common approaches include:

2.2.1. State Objects (S_i)

State objects represent the observable and hidden properties of the environment, and are typically represented as vectors or matrices of real-valued features.

The features can be hand-crafted based on domain knowledge, or learned from data using techniques such as principal component analysis, autoencoders, or variational inference. State objects may also have associated uncertainty estimates, such as covariance matrices or confidence intervals.

2.2.2. Transition Objects (T_j)

Transition objects represent the dynamics of the environment, specifying how states evolve over time in response to actions and events. They can be represented as deterministic or stochastic functions, such as difference equations, differential equations, or probability distributions.

Transition objects may also have associated parameters, such as coefficients or rates, that govern their behavior.

2.2.3. Reward Objects (R_i)

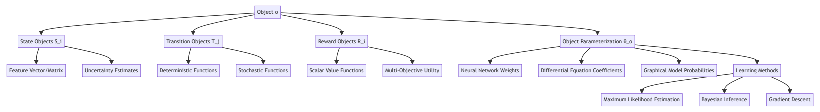
Reward objects represent the goals and preferences of the agent, specifying the desirability of different states and actions. They can be represented as scalar value functions, multi-objective utility functions, or more complex preference relations.

Reward objects may also have associated parameters, such as weights or thresholds, that determine their relative importance and trade-offs.

2.2.4. Object Parameterization (θ_o)

The parameters θ_o of an object o refer to the set of numerical values that define its behavior and relationships, such as the weights of a neural network, the coefficients of a differential equation, or the probabilities of a graphical model.

Parameters can be learned from data using techniques such as maximum likelihood estimation, Bayesian inference, or gradient descent. The choice of parameterization depends on the type and complexity of the object, as well as the available data and computational resources.



3. SELF-REFLECTIVE META-DSL

⚘ 3.1. Definition and Purpose

The self-reflective meta-DSL is a key component of the OORL framework that enables objects to reason about and modify their own learning process. It provides primitives and constructs for introspecting and manipulating the object's own attributes, such as its state, methods, goal, world model, and interaction history.

The purpose of the meta-DSL is to enable a form of "learning to learn", where objects can adapt their own learning strategies based on experience. By providing a language for self-reflection and self-modification, the meta-DSL allows objects to discover and exploit structure in their own learning process, leading to more efficient and effective adaptation.

⚘ 3.2. Core Constructs

The core constructs of the self-reflective meta-DSL include:

3.2.1. DEFINE

The DEFINE construct allows an object to define new attributes, methods, or sub-objects. This can be used to introduce new concepts or abstractions that are useful for learning and reasoning.

For example, an object might define a new feature extractor that computes a low-dimensional representation of its sensory inputs, or a new sub-goal that represents an intermediate milestone on the way to its main objective.

3.2.2. GOAL

The GOAL construct allows an object to specify or modify its own objective function. This can be used to adapt the object's behavior based on feedback or changing circumstances.

For example, an object might update its goal to prioritize certain types of rewards over others, or to incorporate new constraints or trade-offs that were not initially considered.

3.2.3. TASK

The TASK construct allows an object to define a specific sequence of actions or steps to achieve a particular goal. This can be used to break down complex behaviors into simpler sub-tasks that can be learned and executed independently.

For example, an object might define a task for navigating to a particular location, which involves a series of movements and observations.

3.2.4. BELIEF

The BELIEF construct allows an object to represent and reason about its own uncertainty and assumptions about the environment. This can be used to maintain a probabilistic model of the world that can be updated based on new evidence.

For example, an object might have a belief about the location of a particular resource, which can be revised based on its observations and interactions with other objects.

3.2.5. INFER

The INFER construct allows an object to perform inference on its own beliefs and assumptions, in order to update its world model and make predictions. This can be used to integrate new information and reconcile conflicting evidence.

For example, an object might use Bayesian inference to update its belief about the state of the environment based on its sensory inputs and prior knowledge.

3.2.6. DECIDE

The DECIDE construct allows an object to select an action or plan based on its current state, goal, and beliefs. This can be used to implement various decision-making strategies, such as greedy search, planning, or reinforcement learning.

For example, an object might use a decision tree to choose the action that maximizes its expected reward given its current state and uncertainties.

3.2.7. LEARN

The LEARN construct allows an object to update its knowledge and strategies based on new observations or feedback. This can be used to implement various learning algorithms, such as supervised learning, unsupervised learning, or reinforcement learning.

For example, an object might use gradient descent to update the weights of its neural network based on the error between its predicted and actual rewards.

3.2.8. REFINE

The REFINE construct allows an object to modify its own attributes or methods based on meta-level reasoning. This can be used to implement various meta-learning algorithms, such as architecture search, hyperparameter optimization, or curriculum learning.

For example, an object might use reinforcement learning to discover a more efficient state representation or action space, based on its performance on a range of tasks.

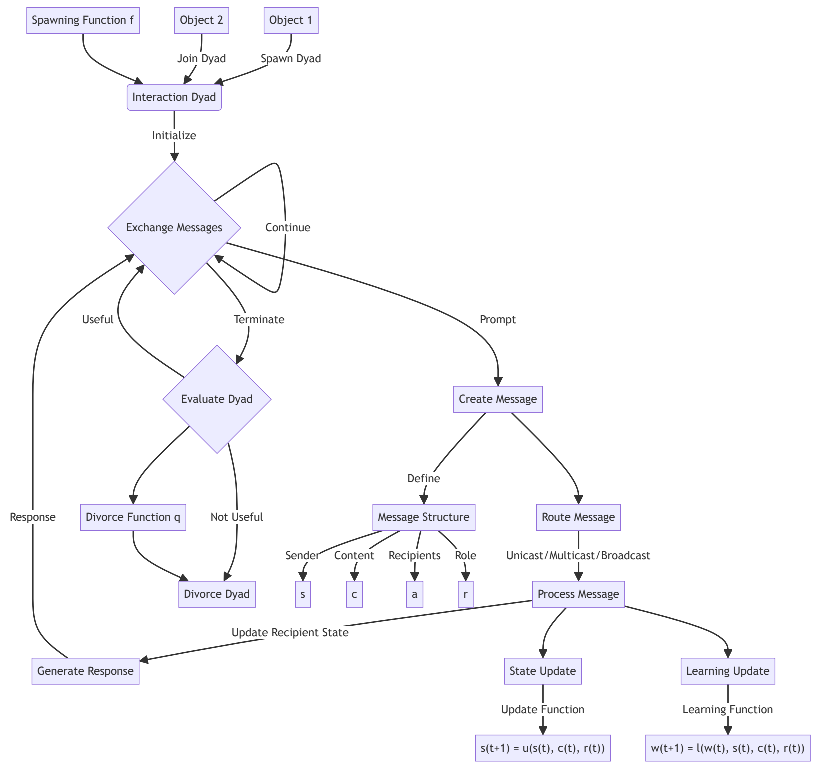
3.3. Initialization, Learning, and Refinement Functions

The self-reflective meta-DSL is implemented as a set of functions that operate on the object's own attributes and methods. These functions can be divided into three main categories:

- Initialization functions: Used to define the initial state and behavior of the object, based on its type and role in the environment. May include functions for setting state, methods, goal, world model, interaction history, hyperparameters or priors.
- Learning functions: Used to update the object's knowledge and strategies based on new observations or feedback. May include functions for updating state, goal, world model, interaction history, learning algorithms or optimization methods.
- Refinement functions: Used to modify the object's own attributes or methods based on meta-level reasoning. May include functions for searching over possible architectures, hyperparameters, curricula, meta-learning algorithms or heuristics.

The specific implementation of these functions depends on the type and complexity of the object, as well as the available data and computational resources. In general, the functions should be designed to balance exploration and exploitation, by allowing the object to discover new strategies and representations while also leveraging its existing knowledge and skills.

4. OBJECT INTERACTIONS



4.1. Interaction Dyads ($\mathcal{I}$)

We define the set of interaction dyads $\mathcal{I}$ as a subset of the Cartesian product of the object set $\mathcal{O}$ with itself:

$\mathcal{I} \subseteq \mathcal{O} \times \mathcal{O}$

Each dyad $i \in \mathcal{I}$ is an ordered pair of objects $(o_1, o_2) \in \mathcal{O} \times \mathcal{O}$ that engage in a prompted exchange of messages according to a specific protocol (defined in Section 4.2).

The set of dyads that an object $o \in \mathcal{O}$ can participate in is determined by its interaction methods $m_i \subseteq m$, which specify the types of messages it can send and receive, as well as any preconditions or postconditions for the interaction.

Formally, we can define the set of dyads that an object o can spawn as:

$\mathcal{I}(o) = \{(o, o') \in \mathcal{O} \times \mathcal{O} \mid \exists\, m_i \in m, m'_i \in m' : \text{compatible}(m_i, m'_i)\}$

where m and m' are the interaction methods of objects o and o' respectively, and compatible(m_i, m'_i) is a predicate that returns true if the methods m_i and m'_i have matching message types and satisfy any necessary preconditions.

The actual set of dyads that are active at any given time step t is a subset $\mathcal{I}_t \subseteq \mathcal{I}$ that is determined by the joint interaction policies of the objects (defined in Section 7) and any environmental constraints or affordances.

4.2. Message-Passing Protocol ($\mathcal{M}$)

The message-passing protocol $\mathcal{M}$ specifies the format and semantics of the prompt-response messages exchanged between objects in each interaction dyad.

Formally, we can define a message m as a tuple:

$m = (s, c, a, r)$

where:

- $s \in \mathcal{S}$ is the sender object
- $c \in \mathbb{C}$ is the content of the message, which can include the sender's state, goal, query, or other relevant information
- $a \in \mathcal{A}$ is the set of recipient objects
- $r \in \{\text{prompt, response}\}$ is the role of the message in the interaction dyad

The protocol $\mathcal{M}$ defines a set of rules and constraints on the structure and sequencing of messages, such as:

- The set of valid message content types $\mathbb{C}$ and their associated semantics
- The mapping from sender and recipient objects to valid message content types: $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{C}$
- The transition function for updating the dyad state based on the exchanged messages: $(s, a, c, r) \times \mathcal{D} \rightarrow \mathcal{D}$, where $\mathcal{D}$ is the set of possible dyad states
- The termination conditions for the interaction dyad based on the exchanged messages or external events

We can model the dynamics of the message-passing protocol as a labeled transition system (LTS):

$\mathcal{L} = (\mathcal{D}, \mathcal{M}, \rightarrow)$

where:

- $\mathcal{D}$ is the set of dyad states, which includes the states of the participating objects and any relevant interaction history or shared context
- $\mathcal{M}$ is the set of messages that can be exchanged according to the protocol rules
- $\rightarrow \subseteq \mathcal{D} \times \mathcal{M} \times \mathcal{D}$ is the labeled transition relation that defines how the dyad state evolves based on the exchanged messages

The LTS provides a formal model of the interaction semantics and can be used to verify properties such as reachability, safety, or liveness of the protocol.

4.2.1. Prompt and Response Messages

Each interaction dyad $(o_1, o_2) \in \mathcal{I}$ consists of a sequence of alternating prompt and response messages:

$(m_1, m_2, \ldots, m_n)$

where:

- $m_i = (o_1, c_i, \{o_2\}, \text{prompt})$ for odd i
- $m_i = (o_2, c_i, \{o_1\}, \text{response})$ for even i
- n is the length of the interaction dyad, which can vary dynamically based on the protocol rules and termination conditions

The prompt messages are sent by the initiating object o_1 and can include queries, proposals, or other information to convey the object's current state, goal, or intentions.

The response messages are sent by the responding object o_2 and can include answers, acknowledgments, or other information to convey the object's reaction or feedback to the prompt.

The content of the prompt and response messages $c_i \in \mathbb{C}$ is determined by the corresponding interaction methods $m_i \in m$ of the sender object, which specify the valid formats and semantics of the messages based on the object's current state and goal.

4.2.2. Routing and Processing of Messages

The message-passing protocol $\mathcal{M}$ includes a routing function that determines how messages are transmitted from sender to recipient objects based on their addresses or attributes:

`route`: $\mathcal{S} \times \mathcal{A} \times \mathbb{C} \to \mathcal{A}$

The routing function can implement various communication patterns, such as:

- Unicast: The message is sent to a single recipient object specified by the sender.
- Multicast: The message is sent to a subset of objects that satisfy a certain predicate or subscription criteria.
- Broadcast: The message is sent to all objects in the system.

The protocol also includes a processing function that determines how messages are handled by the recipient objects based on their methods and world models:

`process`: $\mathcal{A} \times \mathbb{C} \times \mathcal{W} \to \mathcal{W}$

where $\mathcal{W}$ is the set of possible world models or belief states of the recipient object.

The processing function can implement various update rules, such as:

- Bayesian inference: The object updates its beliefs about the environment based on the message content and its prior knowledge.
- Reinforcement learning: The object updates its action policy based on the feedback or reward signal provided by the message.
- Planning: The object updates its goal or plan based on the information or query provided by the message.

4.2.3. Dynamic Spawning and Divorcing of Interaction Dyads

The set of active interaction dyads $\mathcal{I}_t$ can change over time based on the evolving needs and goals of the objects.

New dyads can be spawned by objects that seek to initiate interactions with other objects based on their current state and objectives. The probability of object o spawning a new dyad $(o, o') \in \mathcal{I}(o)$ at time t is given by:

$P(\texttt{spawn}(o, o') \mid s_t, g_t) = f(s_t, g_t, o')$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $f: \mathcal{S} \times \mathcal{G} \times \mathcal{O} \to [0, 1]$ is a spawning function that outputs the probability of initiating an interaction with object o' based on the current state and goal of object o

The spawning function f can be learned or adapted over time based on the object's interaction history h and world model w, using techniques such as reinforcement learning or Bayesian optimization.

Conversely, existing dyads can be dissolved by objects that determine that the interaction is no longer useful or relevant based on the outcomes or feedback received. The probability of object o divorcing an existing dyad $(o, o') \in \mathcal{I}_t$ at time t is given by:

$P(\texttt{divorce}(o, o') \mid s_t, g_t, r_t) = q(s_t, g_t, r_t, o')$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the interaction dyad at time t
- $q: \mathcal{S} \times \mathcal{G} \times \mathcal{R} \times \mathcal{O} \to [0, 1]$ is a divorce function that outputs the probability of terminating an interaction with object o' based on the current state, goal, and reward of object o

The divorce function q can also be learned or adapted over time based on the object's interaction history and world model, using techniques such as multi-armed bandits or Bayesian reinforcement learning.

4.2.4. State Update and Learning Rules

As objects exchange messages and participate in interaction dyads, they update their internal states and learning models based on the information and feedback received.

The state update function for object o at time t is given by:

$s_{t+1} = u(s_t, c_t, r_t)$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $c_t \in \mathbb{C}$ is the content of the message received at time t
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the interaction dyad at time t
- u: $\mathcal{S} \times \mathbb{C} \times \mathcal{R} \to \mathcal{S}$ is an update function that computes the next state of the object based on the current state, message content, and reward signal

The learning update function for object o at time t is given by:

$w_{t+1} = l(w_t, s_t, c_t, r_t)$

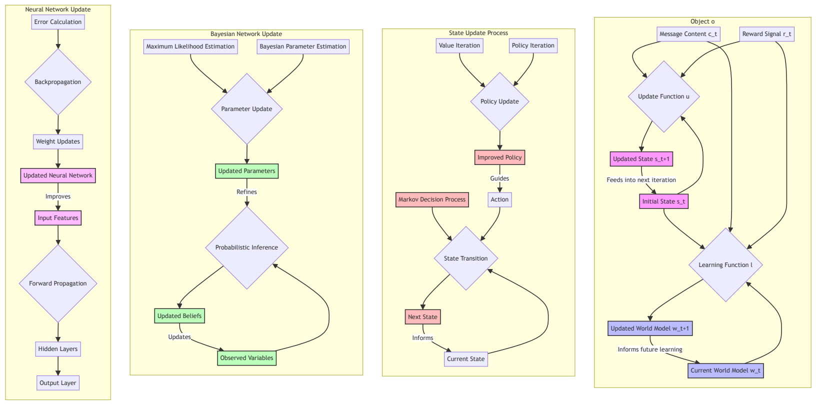
where:

- $w_t \in \mathcal{W}$ is the current world model or learning parameters of object o
- $s_t \in \mathcal{S}$ is the current state of object o
- $c_t \in \mathbb{C}$ is the content of the message received at time t
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the interaction dyad at time t
- l: $\mathcal{W} \times \mathcal{S} \times \mathbb{C} \times \mathcal{R} \to \mathcal{W}$ is a learning function that updates the world model or learning parameters of the object based on the current world model, state, message content, and reward signal

The update and learning functions can implement various state transition and learning rules, such as:

- Markov decision processes: The state update is a stochastic function of the current state and action, and the learning update is based on dynamic programming algorithms such as value iteration or policy iteration.
- Bayesian networks: The state update is based on probabilistic inference over the observed variables, and the learning update is based on maximum likelihood estimation or Bayesian parameter estimation.
- Neural networks: The state update is based on the forward propagation of input features through the network layers, and the learning update is based on backpropagation of error gradients.

The specific choice of update and learning functions depends on the type and complexity of the objects, as well as the assumptions and constraints of the problem domain.



5. SCHEMA EVOLUTION

⚙ 5.1. Schema Configurations ($\mathcal{S}$)

The schema configuration space $\mathcal{S}$ represents the set of all possible arrangements of objects and interaction dyads in the system. Each schema $s \in \mathcal{S}$ corresponds to a particular object-dyad graph $\mathcal{G} = (\mathcal{O}, \mathcal{I})$, which specifies the types, attributes, and relationships of the objects, as well as the patterns and rules of their interactions.

Formally, we can define a schema configuration as a tuple:

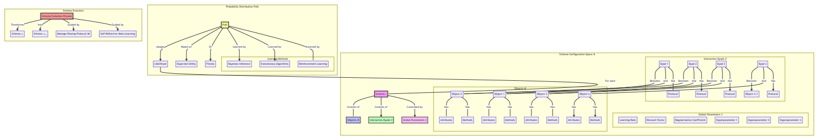
$s = (\mathcal{O}, \mathcal{I}, \theta)$

where:

- $\mathcal{O}$ is the set of objects in the schema, with their associated attributes and methods
- $\mathcal{I} \subseteq \mathcal{O} \times \mathcal{O}$ is the set of interaction dyads between objects, with their associated messages and protocols
- $\theta \in \Theta$ is a set of global parameters or hyperparameters that control the behavior and performance of the schema, such as learning rates, discount factors, or regularization coefficients

The schema configuration space $\mathcal{S}$ is typically large and complex, reflecting the combinatorial nature of the possible object-dyad arrangements and the diversity of their attributes and interactions. The size of the space grows exponentially with the number of objects and dyads, making it infeasible to enumerate or search exhaustively.

Instead, we can define a probability distribution over schemas $P(\mathcal{S})$ that assigns a likelihood or preference to each configuration based on its expected utility or fitness. The probability distribution can be learned or estimated based on the observed performance and feedback of the system over time, using techniques such as Bayesian inference, evolutionary algorithms, or reinforcement learning.



⚙ 5.2. Schema Evolution Process

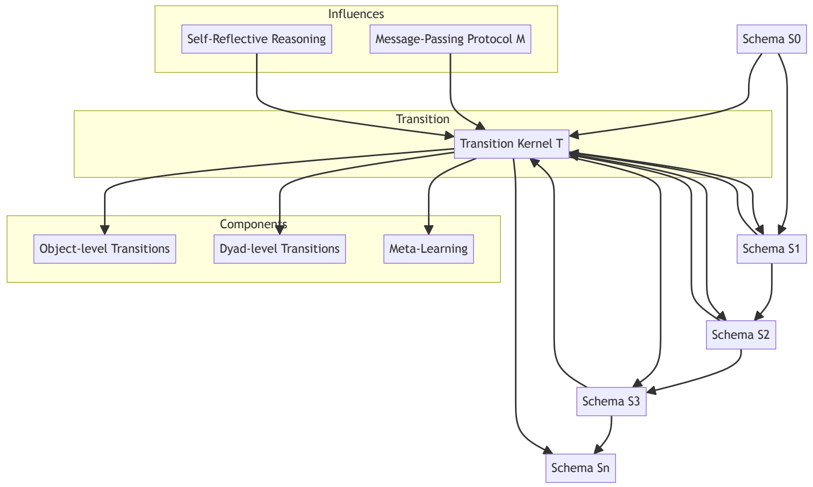
The schema configuration of the system evolves over time through a process of stochastic rewriting of the object-dyad graph, guided by the interactions and adaptations of the individual objects. At each time step t, the current schema $s_t \in \mathcal{S}$ is probabilistically transformed into a new schema $\mathcal{s}_{t+1} \in \mathcal{S}$ based on the joint effects of the message-passing protocol $\mathcal{M}$ and the self-reflective meta-learning of the objects.

Formally, we can model the schema evolution process as a Markov chain over the schema configuration space:

$$\mathcal{S}_0 \to \mathcal{S}_1 \to \cdots \to \mathcal{S}_t \to \mathcal{S}_{t+1} \to \cdots$$

where $\mathcal{S}_t$ is a random variable representing the schema configuration at time t, and the transition probabilities between configurations are given by the schema transition kernel:

$$T(\mathcal{s}' \mid \mathcal{s}) = P(\mathcal{S}_{t+1} = \mathcal{s}' \mid \mathcal{S}_t = \mathcal{s})$$



The transition kernel T specifies the probability of moving from schema $\mathcal{s}$ to schema $\mathcal{s}'$ in one time step, based on the joint effects of the message-passing protocol $\mathcal{M}$ and the self-reflective meta-learning of the objects.

We can decompose the transition kernel into two main components:

1. The object-level transition probabilities, which specify how individual objects update their attributes and methods based on their interactions and adaptations:

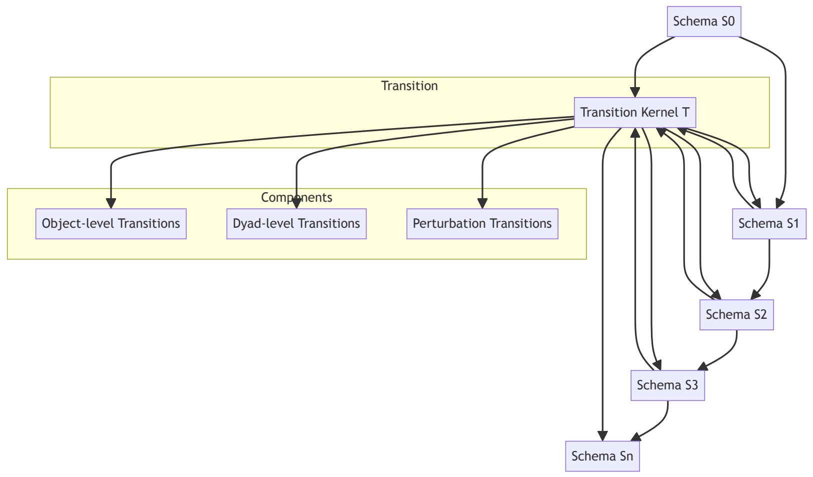
   $$T_i(o' \mid o) = P(O_{t+1} = o' \mid O_t = o)$$

   where O_t is a random variable representing the state of object o at time t.

2. The dyad-level transition probabilities, which specify how interaction dyads are formed or dissolved based on the compatibility and utility of the object-level transitions:

   $$T_{i\,j}(i' \mid i) = P(I_{t+1} = i' \mid I_t = i)$$

   where I_t is a random variable representing the state of dyad i at time t.



The object-level and dyad-level transitions are coupled through the message-passing protocol $\mathcal{M}$, which determines how the output messages of one object affect the input messages and state updates of the other objects in the dyad.

We can express the full schema transition kernel as a product of the object-level and dyad-level transition probabilities, summed over all possible compatible object and dyad configurations:

$T(\mathscr{s}' \mid \mathscr{s}) = \sum_i \prod_j T_i(o'_j \mid o_j) \cdot \prod_k T_{ik}(i'_k \mid i_k)$

where:

- i ranges over all possible object-dyad configurations that are compatible with the schema transition from $\mathscr{s}$ to $\mathscr{s}'$
- j ranges over all objects in the schema
- k ranges over all dyads in the schema

The sum over compatible configurations ensures that the transition probabilities are properly normalized and account for all possible ways of realizing the schema transformation through object-level and dyad-level transitions.

5.2.1. Stochastic Prompting and Interaction Dyad Formation/Dissolution

The formation and dissolution of interaction dyads are key mechanisms of schema evolution, as they enable objects to dynamically update their relationships and communication patterns based on their changing goals and world models.

The probability of forming a new dyad (o, o') between objects o and o' at time t is given by:

$P(I_{t+1} = (o, o') \mid I_t \neq (o, o')) = f(s_t, g_t, o')$

where:

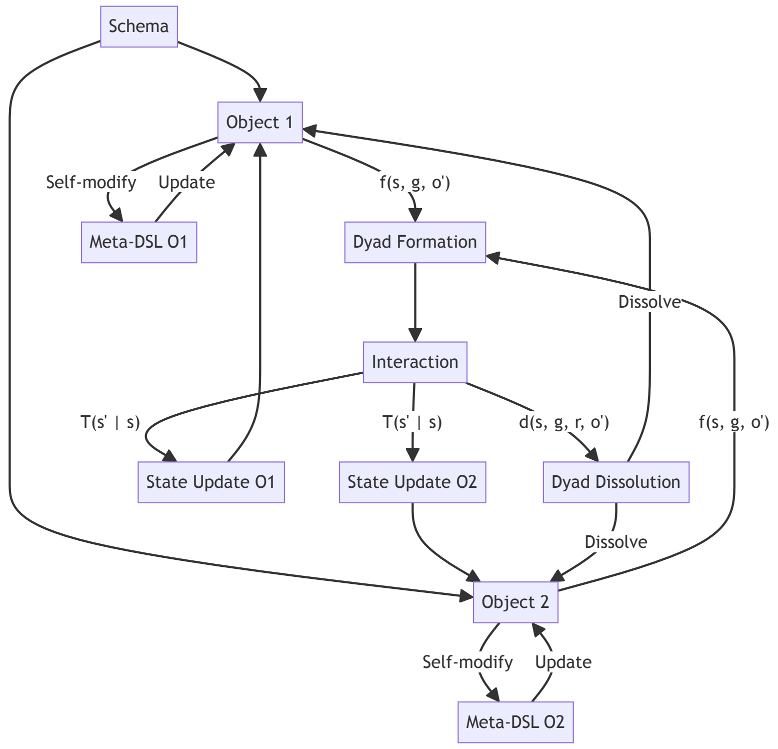
- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $f: \mathcal{S} \times \mathcal{G} \times \boldsymbol{\mathcal{O}} \rightarrow [0, 1]$ is a compatibility function that outputs the probability of forming a dyad with object o' based on the current state and goal of object o

Similarly, the probability of dissolving an existing dyad (o, o') between objects o and o' at time t is given by:

$P(I_{t+1} \neq (o, o') \mid I_t = (o, o')) = d(s_t, g_t, r_t, o')$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the dyad at time t
- $d: \mathcal{S} \times \mathcal{G} \times \mathcal{R} \times \boldsymbol{\mathcal{O}} \rightarrow [0, 1]$ is a utility function that outputs the probability of dissolving the dyad with object o' based on the current state, goal, and reward of object o



The compatibility and utility functions can be learned or adapted over time based on the objects' interaction histories and world models, using techniques such as reinforcement learning, Bayesian optimization, or evolutionary strategies.

The formation and dissolution of dyads induce a stochastic rewriting of the object-dyad graph $\boldsymbol{\mathcal{G}}$, which changes the topology and semantics of the schema configuration. The rewriting process can be modeled as a graph transformation system, with rewrite rules of the form:

$\boldsymbol{\mathcal{L}} \Rightarrow \boldsymbol{\mathcal{R}}$

where:

- $\boldsymbol{\mathcal{L}}$ is a left-hand side pattern that matches a subgraph of the current object-dyad graph $\boldsymbol{\mathcal{G}}_t$
- $\boldsymbol{\mathcal{R}}$ is a right-hand side pattern that specifies how to transform the matched subgraph into a new subgraph $\boldsymbol{\mathcal{G}}_{t+1}$
- $\Rightarrow$ is a rewrite arrow that indicates the direction and probability of the graph transformation

The rewrite rules can be learned or designed based on the desired schema evolution dynamics and the constraints of the problem domain. For example, a rewrite rule for dyad formation might have the form:

$\boldsymbol{\mathcal{L}} = (o, o'), \boldsymbol{\mathcal{R}} = (o \text{ dyad} \rightleftharpoons o')$

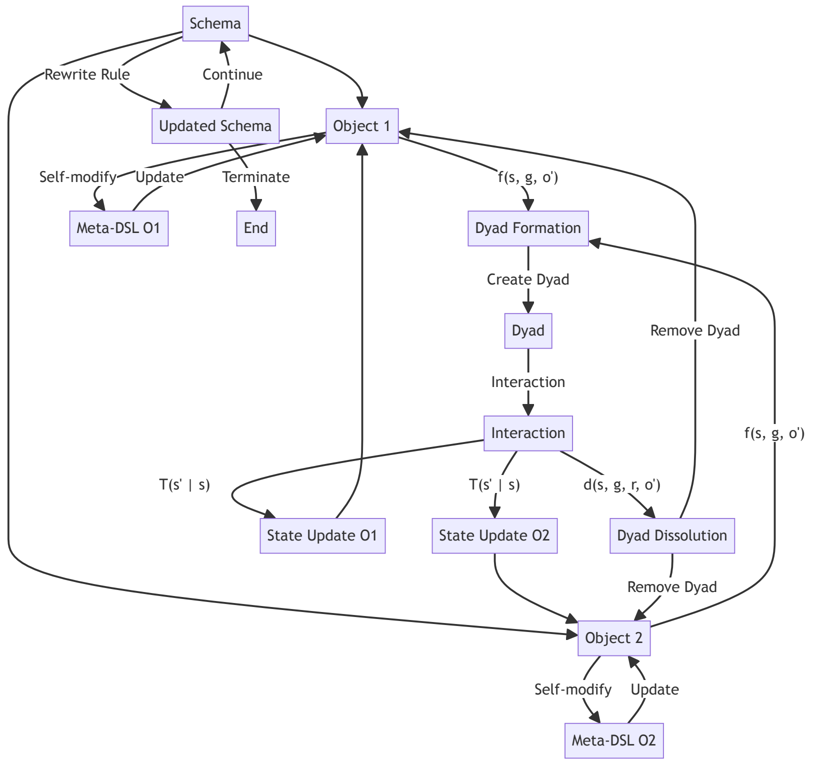
which specifies that if two compatible objects o and o' are matched in the current graph $\mathcal{G}_t$, a new dyad edge should be created between them to form the updated graph $\mathcal{G}_{t+1}$.

Conversely, a rewrite rule for dyad dissolution might have the form:

$\mathcal{L} = (o \text{ dyad} \rightleftharpoons o')$, $\mathcal{R} = (o, o')$

which specifies that if a dyad edge between objects o and o' is matched in the current graph $\mathcal{G}_t$, and the dyad is no longer useful or relevant, the edge should be removed to form the updated graph $\mathcal{G}_{t+1}$.

The probability of applying a rewrite rule to a matched subgraph is given by the product of the compatibility or utility functions of the involved objects and dyads. The rewriting process continues until a stable or optimal schema configuration is reached, or until a maximum number of rewrite steps is exceeded.



5.2.2. *Self-Modification of Objects via Meta-DSLs*

Another key mechanism of schema evolution is the self-modification of objects via their meta-DSLs, which allows them to adapt their own attributes, methods, and interaction patterns based on their experience and feedback.

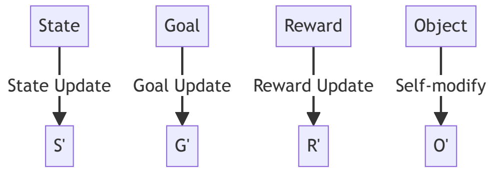
The probability of object o modifying its own configuration at time t is given by:

$P(O_{t+1} = o' \mid O_t = o) = m(s_t, g_t, r_t, o')$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the object's interactions at time t

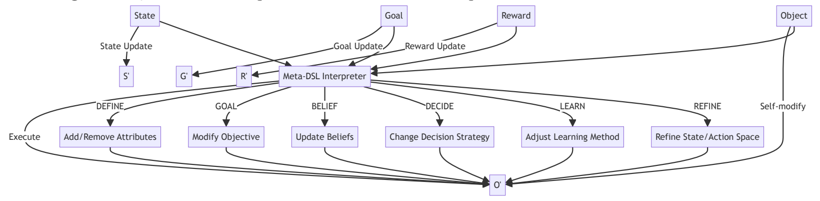
- $m: \mathcal{S} \times \mathcal{G} \times \mathcal{R} \times \boldsymbol{\mathcal{O}} \rightarrow [0, 1]$ is a modification function that outputs the probability of transforming the object's configuration from o to o' based on its current state, goal, and reward



The modification function is implemented by the object's meta-DSL interpreter, which takes as input the current configuration of the object (i.e., its state, goal, methods, and interaction history), and produces as output a new configuration that optimizes the object's performance and adaptability.

The meta-DSL interpreter can use various constructs and operations to transform the object's configuration, such as:

- Adding or removing attributes, methods, or sub-objects (DEFINE)
- Modifying the object's goal or objective function (GOAL)
- Updating the object's beliefs or assumptions about the environment (BELIEF)
- Changing the object's decision-making or learning strategies (DECIDE, LEARN)
- Refining the object's state representation or action space (REFINE)



The choice and probability of applying different meta-DSL constructs depend on the object's current configuration, as well as any meta-level knowledge or heuristics that guide the search for better configurations.

The self-modification of objects induces a stochastic rewriting of the object-dyad graph $\boldsymbol{\mathcal{G}}$, which changes the attributes and methods of the objects, as well as their interactions with other objects. The rewriting process can be modeled as a graph transformation system, with rewrite rules of the form:

$\boldsymbol{\mathcal{L}} \Rightarrow \boldsymbol{\mathcal{R}}$

where:

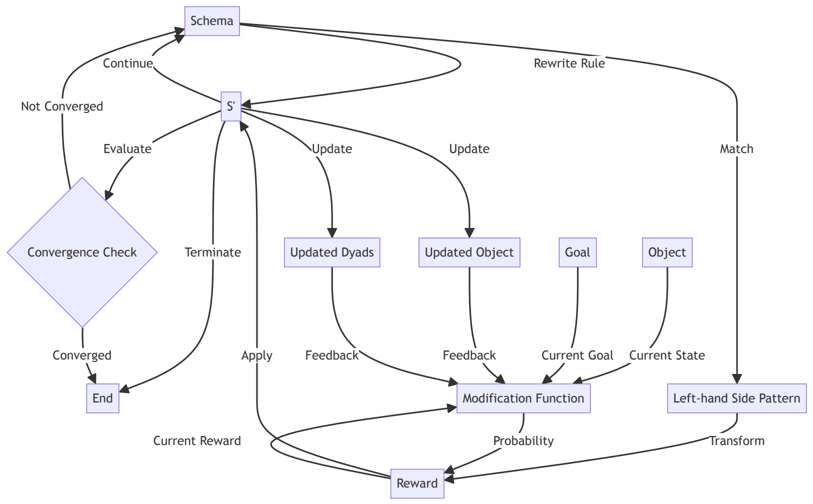
- $\boldsymbol{\mathcal{L}}$ is a left-hand side pattern that matches a subgraph of the current object-dyad graph $\boldsymbol{\mathcal{G}}_t$, consisting of an object o and its attributes, methods, and dyads
- $\boldsymbol{\mathcal{R}}$ is a right-hand side pattern that specifies how to transform the matched subgraph into a new subgraph $\boldsymbol{\mathcal{G}}_{t+1}$, with updated attributes, methods, and dyads for object o
- $\Rightarrow$ is a rewrite arrow that indicates the direction and probability of the graph transformation

The rewrite rules can be learned or designed based on the desired schema evolution dynamics and the constraints of the problem domain. For example, a rewrite rule for object self-modification might have the form:

$\boldsymbol{\mathcal{L}} = (o(s, g, m, h)), \boldsymbol{\mathcal{R}} = (o(s', g', m', h'))$

which specifies that if an object o with configuration (s, g, m, h) is matched in the current graph $\boldsymbol{\mathcal{G}}_t$, its configuration should be transformed to (s', g', m', h') to form the updated graph $\boldsymbol{\mathcal{G}}_{t+1}$, based on the output of the meta-DSL interpreter.

The probability of applying a rewrite rule to a matched subgraph is given by the modification function of the involved object, which takes into account its current state, goal, and reward. The rewriting process continues until a stable or optimal schema configuration is reached, or until a maximum number of rewrite steps is exceeded.



🙞 5.3. Markov Process over Schema States

The schema evolution process can be modeled as a Markov process over the space of possible schema configurations $\mathcal{S}$, where each state corresponds to a particular object-dyad graph $\mathcal{G}$, and the transitions between states are governed by the schema transition kernel T.

Formally, we can define the Markov process as a tuple $(\mathcal{S}, T, \rho_0)$, where:

- $\mathcal{S}$ is the state space of schema configurations
- $T: \mathcal{S} \times \mathcal{S} \to [0, 1]$ is the transition kernel that specifies the probability of moving from one schema configuration to another in one time step
- $\rho_0: \mathcal{S} \to [0, 1]$ is the initial distribution over schema configurations, which specifies the probability of starting the process in each possible configuration

The transition kernel T can be decomposed into the product of the object-level and dyad-level transition probabilities, as described in Section 5.2. The initial distribution ρ_0 can be specified based on prior knowledge or assumptions about the problem domain, or learned from data using techniques such as maximum likelihood estimation or Bayesian inference.

Given the Markov process $(\mathcal{S}, T, \rho_0)$, we can compute various properties and statistics of the schema evolution dynamics, such as:

- The stationary distribution $\pi: \mathcal{S} \to [0, 1]$, which specifies the long-term probability of being in each schema configuration, and satisfies the equation:

  $\pi(s) = \sum_{s'\in\mathcal{S}} T(s \mid s') \cdot \pi(s')$

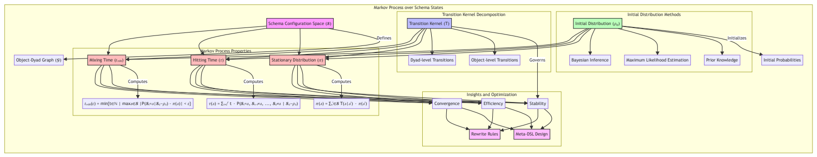
- The hitting time $\tau(s): \mathcal{S} \to \mathbb{N}$, which specifies the expected number of time steps to reach a particular schema configuration s starting from the initial distribution, and satisfies the equation:

  $\tau(s) = \sum_{t=0}^{\infty} t \cdot P(\mathcal{S}_t = s, \mathcal{S}_{t-1} \neq s, ..., \mathcal{S}_0 \neq s \mid \mathcal{S}_0 \sim \rho_0)$

- The mixing time $t_{\mathrm{mix}}(\varepsilon): \mathbb{R} \to \mathbb{N}$, which specifies the expected number of time steps to reach a stationary distribution that is ε-close to the true stationary distribution, and satisfies the equation:

$t_{\text{mix}}(\varepsilon) = \min\{t \in \mathbb{N} \mid \max_{s \in \mathcal{S}} |P(\mathcal{S}_t = s \mid \mathcal{S}_0 \sim \rho_0) - \pi(s)| < \varepsilon\}$

These properties can provide insights into the efficiency, stability, and convergence of the schema evolution process, and guide the design of the meta-DSL constructs and rewrite rules to optimize the learning dynamics.


Markov Process over Schema States
Schema Configuration Space (𝒮)
Defines
Markov Process Properties
Object-Dyad Graph (𝒢)
Mixing Time (t_mix)
Hitting Time (τ)
Stationary Distribution (π)
Computes
Computes
Computes
t_mix(ε) = min{t∈ℕ | max𝓈∈𝒮 |P(𝒮ₜ=𝓈|𝒮₀~ρ₀) - π(𝓈)| < ε}
τ(𝓈) = Σₜ₌₀ᵗ' t · P(𝒮ₜ=𝓈, 𝒮ₜ₋₁≠𝓈, ..., 𝒮₀≠𝓈 | 𝒮₀~ρ₀)
π(𝓈) = Σ𝓈'∈𝒮 T(𝓈|𝓈') · π(𝓈')
Transition Kernel Decomposition
Transition Kernel (T)
Dyad-level Transitions
Object-level Transitions
Governs
Initial Distribution Methods
Initial Distribution (ρ₀)
Initializes
Bayesian Inference
Maximum Likelihood Estimation
Prior Knowledge
Initial Probabilities
Insights and Optimization
Convergence
Efficiency
Stability
Rewrite Rules
Meta-DSL Design

5.3.1. Transition Operator ($\mathcal{T}$)

The transition operator $\mathcal{T}: \mathcal{S} \to \Delta(\mathcal{S})$ is a mapping from the current schema configuration $s_t \in \mathcal{S}$ to a probability distribution over the next schema configuration $s_{t+1} \in \mathcal{S}$, where $\Delta(\mathcal{S})$ denotes the set of all probability distributions over $\mathcal{S}$.

Formally, we can define the transition operator as:

$\mathcal{T}(s_t)(s_{t+1}) = P(\mathcal{S}_{t+1} = s_{t+1} \mid \mathcal{S}_t = s_t)$

which specifies the probability of transitioning from schema s_t to schema s_{t+1} in one time step, based on the joint effects of the message-passing protocol $\mathcal{M}$, the self-modification of objects via their meta-DSLs, and any other stochastic factors that influence the schema evolution process.

The transition operator can be decomposed into two main components:

1. The object-level transition operator $\mathcal{T}_i: \mathcal{O} \to \Delta(\mathcal{O})$, which specifies how individual objects update their attributes and methods based on their interactions and adaptations:

$\mathcal{T}_i(o_t)(o_{t+1}) = P(O_{t+1} = o_{t+1} \mid O_t = o_t)$

2. The dyad-level transition operator $\mathcal{T}_{i\,j}: \mathcal{I} \to \Delta(\mathcal{I})$, which specifies how interaction dyads are formed or dissolved based on the compatibility and utility of the object-level transitions:

$\mathcal{T}_{i\,j}(i_t)(i_{t+1}) = P(I_{t+1} = i_{t+1} \mid I_t = i_t)$

The object-level and dyad-level transition operators are coupled through the message-passing protocol $\mathcal{M}$, which determines how the output messages of one object affect the input messages and state updates of the other objects in the dyad.

We can express the full transition operator as a composition of the object-level and dyad-level transition operators, applied to all objects and dyads in the schema:

$\mathcal{T}(s_t)(s_{t+1}) = \prod_i \mathcal{T}_i(o_{it})(o_{it+1}) \cdot \prod_j \mathcal{T}_{i\,j}(i_{j\,t})(i_{j\,t+1})$

where:

- o_{it} denotes the state of object i at time t
- $i_{j\,t}$ denotes the state of dyad j at time t
- $\prod$ denotes the product of the transition probabilities over all objects and dyads in the schema

The composition of transition operators ensures that the full transition probability accounts for all possible ways of realizing the schema transformation through object-level and dyad-level state updates.

5.3.2. Transition Probabilities and Schema Perturbations

The transition probabilities between schema configurations are determined by the compatibility and utility of the object-level and dyad-level state updates, as well as any stochastic perturbations or exploration mechanisms that introduce noise or diversity into the schema evolution process.

The compatibility of object-level updates is determined by the spawning and dissolution functions f and d, which output the probability of forming or dissolving dyads based on the objects' current states and goals:

$P(I_{t+1} = (o, o') \mid I_t \neq (o, o')) = f(s_t, g_t, o')$ $P(I_{t+1} \neq (o, o') \mid I_t = (o, o')) = d(s_t, g_t, r_t, o')$

The utility of dyad-level updates is determined by the reward or feedback signals received from the interactions, as well as any intrinsic motivation or curiosity objectives that drive the exploration of novel or informative configurations:

$P(I_{t+1} = i' \mid I_t = i) \propto \exp(\beta \cdot R(i, i'))$

where:

- $R(i, i')$ is a reward function that measures the expected value or feedback of transitioning from dyad i to dyad i'
- β is an inverse temperature parameter that controls the trade-off between exploitation and exploration

The reward function can be learned or approximated based on the interaction history and world models of the objects, using techniques such as reinforcement learning, Bayesian optimization, or evolutionary strategies.

The schema perturbations are stochastic factors that introduce noise or diversity into the schema evolution process, and enable the exploration of novel or unconventional configurations that may not be immediately compatible or rewarding. Examples of schema perturbations include:

- Random spawning or dissolution of dyads, based on a fixed probability or a decreasing temperature schedule (simulated annealing)
- Random modification of object attributes or methods, based on a mutation rate or a genetic algorithm (evolutionary search)
- Stochastic resampling or reweighting of object and dyad states, based on a particle filter or a sequential Monte Carlo method (Bayesian inference)

The schema perturbations can be modeled as additional transition operators that compose with the object-level and dyad-level transition operators to form the full transition operator:

$\mathcal{T}(\mathcal{s}_t)(\mathcal{s}_{t+1}) = \prod_i \mathcal{T}_i(o_{it})(o_{it+1}) \cdot \prod_j \mathcal{T}_{ij}(i_{jt})(i_{jt+1}) \cdot \prod_k \mathcal{T}_k(\mathcal{s}_t)(\mathcal{s}_{t+1})$

where:

- $\mathcal{T}_k$ are the perturbation transition operators that introduce noise or diversity into the schema evolution process
- $\prod_k$ denotes the product of the perturbation transition probabilities over all perturbation mechanisms

The composition of object-level, dyad-level, and perturbation transition operators allows for a flexible and adaptive schema evolution process that can explore a wide range of configurations and optimize for multiple objectives, while still maintaining the stability and coherence of the object-dyad graph.

5.4. Schema Evolution with Self-Modification

The self-modification capabilities of the objects, enabled by their meta-DSLs, play a key role in the schema evolution process by allowing objects to adapt their own attributes, methods, and interaction patterns based on their experience and feedback.

The self-modification transition operator for object o at time t is given by:

$\mathcal{T}_i(o_t)(o_{t+1}) = P(O_{t+1} = o_{t+1} \mid O_t = o_t) = m(s_t, g_t, r_t, o_{t+1})$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the object's interactions at time t
- m: $\mathcal{S} \times \mathcal{G} \times \mathcal{R} \times \boldsymbol{\mathcal{O}} \rightarrow [0, 1]$ is the modification function that outputs the probability of transforming the object's configuration from o_t to o_{t+1} based on its current state, goal, and reward

The modification function m is implemented by the object's meta-DSL interpreter, which takes as input the current configuration of the object and produces as output a new configuration that optimizes the object's performance and adaptability.

The meta-DSL interpreter can use various constructs and operations to transform the object's configuration, such as:

- Adding or removing attributes, methods, or sub-objects (DEFINE)
- Modifying the object's goal or objective function (GOAL)
- Updating the object's beliefs or assumptions about the environment (BELIEF)
- Changing the object's decision-making or learning strategies (DECIDE, LEARN)
- Refining the object's state representation or action space (REFINE)

The choice and probability of applying different meta-DSL constructs depend on the object's current configuration, as well as any meta-level knowledge or heuristics that guide the search for better configurations.

The self-modification of objects induces a stochastic rewriting of the object-dyad graph $\boldsymbol{\mathcal{G}}$, which changes the attributes and methods of the objects, as well as their interactions with other objects. The rewriting process can be modeled as a graph transformation system, with rewrite rules of the form:

$\boldsymbol{\mathcal{L}} \Rightarrow \boldsymbol{\mathcal{R}}$

where:

- $\boldsymbol{\mathcal{L}}$ is a left-hand side pattern that matches a subgraph of the current object-dyad graph $\boldsymbol{\mathcal{G}}_t$, consisting of an object o and its attributes, methods, and dyads
- $\boldsymbol{\mathcal{R}}$ is a right-hand side pattern that specifies how to transform the matched subgraph into a new subgraph $\boldsymbol{\mathcal{G}}_{t+1}$, with updated attributes, methods, and dyads for object o
- $\Rightarrow$ is a rewrite arrow that indicates the direction and probability of the graph transformation

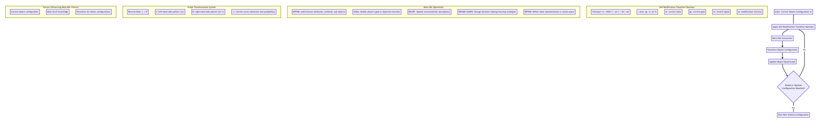
The rewrite rules can be learned or designed based on the desired schema evolution dynamics and the constraints of the problem domain. For example, a rewrite rule for object self-modification might have the form:

$\boldsymbol{\mathcal{L}} = (o_t)$, $\boldsymbol{\mathcal{R}} = (o_{t+1})$

which specifies that if an object o with configuration o_t is matched in the current graph $\boldsymbol{\mathcal{G}}_t$, its configuration should be transformed to o_{t+1} to form the updated graph $\boldsymbol{\mathcal{G}}_{t+1}$, based on the output of the meta-DSL interpreter.

The probability of applying a rewrite rule to a matched subgraph is given by the modification function of the involved object, which takes into account its current state, goal, and reward. The rewriting process continues until a stable or optimal schema configuration is reached, or until a maximum number of rewrite steps is exceeded.

The self-modification of objects allows for a more flexible and adaptive schema evolution process, where the objects can continuously update their own configurations based on their interactions and feedback, without relying on a fixed set of update rules or heuristics. This enables the discovery of novel and efficient


Factors Influencing Meta-DSL Choices
Current object configuration
Meta-level knowledge
Heuristics for better configurations
Graph Transformation System
Rewrite Rule: L → R
L: left-hand side pattern (st)
R: right-hand side pattern (st+1)
→: rewrite arrow (direction and probability)
Meta-DSL Operations
DEFINE: Add/remove attributes, methods, sub-objects
GOAL: Modify object's goal or objective function
BELIEF: Update environmental assumptions
DECIDE/LEARN: Change decision-making/learning strategies
REFINE: Refine state representation or action space
Self-Modification Transition Operator
P(st+1) = P(Ot+1 = st+1 | Ot = st)
= m(st, gt, rt, st+1)
st: current state
gt: current goal
rt: reward signal
m: modification function
Start: Current Object Configuration ot
Apply Self-Modification Transition Operator
Meta-DSL Interpreter
Transform Object Configuration
Update Object-Dyad Graph
Stable or Optimal Configuration Reached?
No
Yes
End: New Schema Configuration

6. REWARD LEARNING

୭ 6.1. Reward Objects (ℝ)

In the OORL framework, rewards are represented by a special type of object called reward objects, denoted by ℝ. Each reward object $\mathscr{R} \in \mathbb{R}$ corresponds to a particular type of feedback or signal that an object can receive from the environment or from other objects, indicating the desirability or value of its current state or actions.

6.1.1. Valence Attribute (v)

Each reward object $\mathscr{R}$ has a valence attribute $v \in \{-1,1\}$, which indicates whether the reward is positive (i.e., a gain or benefit) or negative (i.e., a loss or cost). The valence attribute allows objects to distinguish between rewards that they should seek to maximize and punishments that they should seek to minimize.

The reward objects are designed to encapsulate both extrinsic rewards, which are provided by the environment and typically represent external goals or incentives, and intrinsic rewards, which are generated by the agent itself and typically represent internal motivations or drives.

6.1.2. Reward Signal (r)

The reward signal $r \in \mathbb{R}$ represents the magnitude or intensity of the reward and can be a continuous or discrete value. The reward signal is used to quantify the desirability or value of a particular state or action, allowing the object to update its behavior and learning strategy accordingly.

The reward objects can be parameterized by various attributes, such as the source or origin of the reward, the temporal horizon over which the reward is expected to be realized, or any dependencies or contingencies that affect the reward's value or relevance.

୭ 6.2. Reward Functions

The reward functions in the OORL framework define how the reward signals are computed and assigned to different states or actions. These functions can be specified by the system designer, learned from data, or adapted based on the agent's experience and interactions.

6.2.1. Extrinsic Reward Functions

Extrinsic reward functions are provided by the environment and typically represent external goals or incentives. They are defined as:

[$R_e(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$]

where ($\mathcal{S}$) is the state space, ($\mathcal{A}$) is the action space, and ($R_e(s, a)$) is the extrinsic reward signal for taking action (a) in state (s).

6.2.2. Intrinsic Reward Functions

Intrinsic reward functions are generated by the agent itself and typically represent internal motivations or drives, such as curiosity, exploration, or empowerment. They are defined as:

[$R_i(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$]

where ($\mathcal{S}$) is the state space, ($\mathcal{A}$) is the action space, and ($R_i(s, a)$) is the intrinsic reward signal for taking action (a) in state (s).

6.3. Reward Learning Algorithms

The reward learning algorithms in the OORL framework enable objects to learn and adapt their reward functions based on their interactions and feedback. These algorithms can be based on various machine learning and optimization techniques, such as reinforcement learning, inverse reinforcement learning, or preference learning.

6.3.1. Reinforcement Learning

Reinforcement learning algorithms allow objects to learn optimal policies for maximizing their expected cumulative rewards. These algorithms typically involve estimating value functions or action-value functions and using them to guide the object's decision-making process.

6.3.2. Inverse Reinforcement Learning

Inverse reinforcement learning algorithms allow objects to infer the underlying reward functions based on observed behavior or demonstrations. These algorithms typically involve solving an inverse problem to identify the reward functions that best explain the observed behavior.

6.3.3. Preference Learning

Preference learning algorithms allow objects to learn reward functions based on preferences or rankings provided by the environment or other objects. These algorithms typically involve learning a utility function that captures the preferences and using it to guide the object's behavior.

6.4. Combining Extrinsic and Intrinsic Rewards

In the OORL framework, objects can combine extrinsic and intrinsic rewards to form a composite reward function that balances external goals and internal motivations. The composite reward function is defined as:

$[R(s, a) = \alpha R_e(s, a) + \beta R_i(s, a)]$

where (α) and (β) are weighting factors that determine the relative importance of extrinsic and intrinsic rewards.

The weighting factors can be learned or adapted based on the object's experience and feedback, allowing the object to dynamically adjust its behavior and learning strategy to optimize both external goals and internal motivations.

7. POLICY OPTIMIZATION

7.1. Policy Representation

The policy representation in the OORL framework defines how objects represent and parameterize their decision-making strategies. Policies can be represented using various models, such as neural networks, decision trees, or probabilistic graphical models.

7.2. Policy Gradient Methods

Policy gradient methods are a class of optimization algorithms that allow objects to directly optimize their policies by estimating the gradient of the expected cumulative reward with respect to the policy parameters. These methods typically involve sampling trajectories from the policy and using them to compute gradient estimates.

7.3. Value Function Approximation

Value function approximation methods are a class of optimization algorithms that allow objects to approximate the value functions or action-value functions used to guide their decision-making process. These methods typically involve fitting a parametric model to the observed rewards and using it to estimate the expected cumulative rewards.

7.4. Meta-Learning

Meta-learning methods are a class of optimization algorithms that allow objects to learn and adapt their learning strategies based on their experience and feedback. These methods typically involve learning a meta-policy or meta-model that captures the structure and dependencies in the learning process and using it to guide the object's adaptation.

8. OPEN-ENDED LEARNING

8.1. Exploration and Exploitation

Open-ended learning in the OORL framework involves balancing exploration and exploitation to discover and optimize new behaviors and representations. Exploration involves seeking out novel or informative states and actions, while exploitation involves leveraging existing knowledge and strategies to achieve desired outcomes.

8.2. Intrinsic Motivation

Intrinsic motivation plays a key role in open-ended learning by driving objects to seek out novelty, diversity, or complexity in their interactions and behaviors. Intrinsic motivation can be modeled using various reward functions or value functions that capture the agent's curiosity, empowerment, or intrinsic goals.

8.3. Continual Learning

Continual learning in the OORL framework involves continuously updating and refining the object's knowledge and strategies based on new experiences and feedback. This requires the development of robust and scalable learning algorithms that can handle non-stationary environments and incremental updates.

8.4. Curriculum Learning

Curriculum learning involves structuring the learning process in a way that gradually increases the complexity and difficulty of the tasks and challenges faced by the object. This can help improve the efficiency and effectiveness of the learning process by providing a structured progression of learning experiences.

9. EXTENSIONS AND APPLICATIONS

9.1. Multi-Agent Systems

The OORL framework can be extended to multi-agent systems, where multiple objects interact and collaborate to achieve shared goals or compete to maximize their individual rewards. This requires the development of coordination mechanisms, communication protocols, and negotiation strategies.

9.2. Hierarchical Reinforcement Learning

Hierarchical reinforcement learning involves decomposing complex tasks into simpler sub-tasks and learning policies at multiple levels of abstraction. This can help improve the scalability and efficiency of the learning process by leveraging hierarchical structures and temporal abstractions.

9.3. Transfer Learning

Transfer learning involves leveraging knowledge and skills learned in one context or domain to improve performance in another context or domain. This requires the development of methods for knowledge transfer, domain adaptation, and multi-task learning.

9.4. Real-World Applications

The OORL framework can be applied to various real-world problems and domains, such as robotics, game playing, scientific discovery, and more. Evaluating the effectiveness and scalability of the OORL framework in these applications requires the development of benchmarks, simulations, and real-world experiments.

10. CONCLUSION

The OORL framework provides a powerful and flexible approach to reinforcement learning in complex and dynamic environments. By combining object-oriented programming, graph theory, and meta-learning, the OORL framework enables the development of adaptive and open-ended learning systems that can discover and optimize novel behaviors and representations. While the OORL framework introduces new challenges in terms of stability, interpretability, and controllability, it also offers exciting opportunities for further research and innovation in the field of machine learning and artificial intelligence.

1. INTRODUCTION

1.1. Motivation and Objectives

Reinforcement learning has made significant strides in enabling agents to learn complex behaviors through interaction with their environment. However, traditional approaches often struggle in open-ended, dynamic environments where the optimal behavior and relevant features may change over time.

To address these challenges, we propose Object-Oriented Reinforcement Learning (OORL) in Mutable Ontologies - a novel framework that combines ideas from object-oriented programming, graph theory, and meta-learning to enable more flexible, adaptive, and open-ended learning. The main objectives are:

1. Represent the learning process as a multi-agent interaction between evolving objects, each with its own state, behavior, and goals.
2. Enable the agent to discover and adapt its own representation of the environment through dynamic formation and dissolution of object relationships.
3. Introduce a self-reflective meta-language that allows objects to reason about and modify their own learning process.
4. Support open-ended learning through continuous exploration and expansion of the space of possible behaviors and representations.

1.2. Overview of OORL in Mutable Ontologies

The key idea is to model the environment as a graph of interacting objects, where each object represents a particular aspect of the state space and is equipped with its own learning mechanism. Objects engage in goal-directed behavior by exchanging messages according to a dynamic protocol that evolves over time. The global behavior emerges from the local interactions and adaptations of the individual objects.

A central component is the self-reflective meta-DSL (domain-specific language) that enables objects to inspect and modify their own internal structure and behavior. This allows for a form of "learning to learn" where objects can adapt their own learning strategies based on experience.

The learning process itself is formulated as a multi-objective optimization problem, where the agent seeks to maximize a combination of external rewards and intrinsic motivations, such as empowerment and curiosity. This is achieved through a combination of policy gradients, value function approximation, and meta-learning techniques.

The following sections provide a detailed mathematical formulation of the OORL framework, covering the key components of object schema, interactions, schema evolution, reward learning, policy optimization, open-ended learning, and various extensions and applications.

2. OBJECT SCHEMA

2.1. Definition of Objects and their Attributes

In the OORL framework, the environment is modeled as a set of objects $\mathcal{O}$, where each object $o \in \mathcal{O}$ is a tuple $o = (s, m, g, w, h, d)$ consisting of the following attributes:

- s: The object's internal state, represented by a set of parameters.
- m: The set of methods or functions that the object can perform.
- g: The object's goal or objective function, specifying its desired outcomes.
- w: The object's world model, representing its beliefs and assumptions about the environment.
- h: The object's interaction history, storing a record of its past exchanges with other objects.
- d: The object's self-descriptive meta-DSL, used for introspection and self-modification.

2.1.1. Internal State (s)

The internal state s represents the object's current configuration and can be thought of as a vector of parameter values. The specific structure and semantics of the state space depend on the type and function of the object.

For example, a sensor object might have a state representing the current readings of its input channels, while an actuator object might have a state representing the current positions of its joints.

2.1.2. Methods/Functions (m)

The methods m define the set of actions or transformations that the object can perform. These can include both internal computations (e.g. updating the state based on new observations) and external interactions (e.g. sending a message to another object).

Methods are typically parameterized by the object's current state and any arguments provided by the caller.

2.1.3. Goal or Objective Function (g)

The goal g specifies the object's desired outcomes or objectives. This can be represented as a scalar value function that assigns a score to each possible state, or as a more complex objective function that takes into account multiple criteria.

The goal is used to guide the object's behavior and learning, by providing a measure of the desirability of different actions and outcomes.

2.1.4. World Model (w)

The world model w represents the object's beliefs and assumptions about the environment, including the states and behaviors of other objects.

This can be represented as a probabilistic graphical model, such as a Bayesian network or a Markov decision process, that captures the dependencies and uncertainties in the environment. The world model is used to make predictions and inform decision-making, and is updated based on the object's observations and interactions.

2.1.5. Interaction History (h)

The interaction history h stores a record of the object's past exchanges with other objects, including the messages sent and received, actions taken, and rewards or feedback obtained.

This information is used for learning and adaptation, by providing data for updating the object's world model, goal, and behavior. The interaction history can be represented as a time-indexed sequence of tuples, or as a more compact summary statistic.

2.1.6. Self-Descriptive Meta-DSL (d)

The self-descriptive meta-DSL d is a domain-specific language that enables introspection and self-modification. It provides primitives and constructs for querying and manipulating the object's own attributes, such as its state, methods, goal, world model, and interaction history.

The meta-DSL is used to implement meta-learning algorithms that can adapt the object's learning strategy based on experience. Examples of meta-DSL constructs include:

- DEFINE: Defines new attributes, methods, or sub-objects.
- GOAL: Specifies or modifies the object's objective function.
- BELIEF: Represents a probabilistic belief or assumption about the environment.
- INFER: Performs inference on the object's beliefs to update its world model.
- DECIDE: Selects an action or plan based on current state, goal, and beliefs.
- LEARN: Updates knowledge and strategies based on new observations or feedback.
- REFINE: Modifies attributes or methods based on meta-level reasoning.

🜲 2.2. Object Representation and Parameterization

To enable efficient learning and reasoning, objects in the OORL framework are represented using parameterized models that capture their key attributes and relationships. The specific choice of representation depends on the type and complexity of the object, but common approaches include:

2.2.1. State Objects (S_i)

State objects represent the observable and hidden properties of the environment, and are typically represented as vectors or matrices of real-valued features.

The features can be hand-crafted based on domain knowledge, or learned from data using techniques such as principal component analysis, autoencoders, or variational inference. State objects may also have associated uncertainty estimates, such as covariance matrices or confidence intervals.

2.2.2. Transition Objects (T_j)

Transition objects represent the dynamics of the environment, specifying how states evolve over time in response to actions and events. They can be represented as deterministic or stochastic functions, such as difference equations, differential equations, or probability distributions.

Transition objects may also have associated parameters, such as coefficients or rates, that govern their behavior.

2.2.3. Reward Objects (R_i)

Reward objects represent the goals and preferences of the agent, specifying the desirability of different states and actions. They can be represented as scalar value functions, multi-objective utility functions, or more complex preference relations.

Reward objects may also have associated parameters, such as weights or thresholds, that determine their relative importance and trade-offs.

2.2.4. Object Parameterization (θ_o)

The parameters θ_o of an object o refer to the set of numerical values that define its behavior and relationships, such as the weights of a neural network, the coefficients of a differential equation, or the probabilities of a graphical model.

Parameters can be learned from data using techniques such as maximum likelihood estimation, Bayesian inference, or gradient descent. The choice of parameterization depends on the type and complexity of the object, as well as the available data and computational resources.

3. SELF-REFLECTIVE META-DSL

🜲 3.1. Definition and Purpose

The self-reflective meta-DSL is a key component of the OORL framework that enables objects to reason about and modify their own learning process. It provides primitives and constructs for introspecting and manipulating the object's own attributes, such as its state, methods, goal, world model, and interaction history.

The purpose of the meta-DSL is to enable a form of "learning to learn", where objects can adapt their own learning strategies based on experience. By providing a language for self-reflection and self-modification, the meta-DSL allows objects to discover and exploit structure in their own learning process, leading to more efficient and effective adaptation.

3.2. Core Constructs

The core constructs of the self-reflective meta-DSL include:

3.2.1. DEFINE

The DEFINE construct allows an object to define new attributes, methods, or sub-objects. This can be used to introduce new concepts or abstractions that are useful for learning and reasoning.

For example, an object might define a new feature extractor that computes a low-dimensional representation of its sensory inputs, or a new sub-goal that represents an intermediate milestone on the way to its main objective.

3.2.2. GOAL

The GOAL construct allows an object to specify or modify its own objective function. This can be used to adapt the object's behavior based on feedback or changing circumstances.

For example, an object might update its goal to prioritize certain types of rewards over others, or to incorporate new constraints or trade-offs that were not initially considered.

3.2.3. TASK

The TASK construct allows an object to define a specific sequence of actions or steps to achieve a particular goal. This can be used to break down complex behaviors into simpler sub-tasks that can be learned and executed independently.

For example, an object might define a task for navigating to a particular location, which involves a series of movements and observations.

3.2.4. BELIEF

The BELIEF construct allows an object to represent and reason about its own uncertainty and assumptions about the environment. This can be used to maintain a probabilistic model of the world that can be updated based on new evidence.

For example, an object might have a belief about the location of a particular resource, which can be revised based on its observations and interactions with other objects.

3.2.5. INFER

The INFER construct allows an object to perform inference on its own beliefs and assumptions, in order to update its world model and make predictions. This can be used to integrate new information and reconcile conflicting evidence.

For example, an object might use Bayesian inference to update its belief about the state of the environment based on its sensory inputs and prior knowledge.

3.2.6. DECIDE

The DECIDE construct allows an object to select an action or plan based on its current state, goal, and beliefs. This can be used to implement various decision-making strategies, such as greedy search, planning, or reinforcement learning.

For example, an object might use a decision tree to choose the action that maximizes its expected reward given its current state and uncertainties.

3.2.7. LEARN

The LEARN construct allows an object to update its knowledge and strategies based on new observations or feedback. This can be used to implement various learning algorithms, such as supervised learning, unsupervised learning, or reinforcement learning.

For example, an object might use gradient descent to update the weights of its neural network based on the error between its predicted and actual rewards.

3.2.8. REFINE

The REFINE construct allows an object to modify its own attributes or methods based on meta-level reasoning. This can be used to implement various meta-learning algorithms, such as architecture search, hyperparameter optimization, or curriculum learning.

For example, an object might use reinforcement learning to discover a more efficient state representation or action space, based on its performance on a range of tasks.

3.3. Initialization, Learning, and Refinement Functions

The self-reflective meta-DSL is implemented as a set of functions that operate on the object's own attributes and methods. These functions can be divided into three main categories:

- Initialization functions: Used to define the initial state and behavior of the object, based on its type and role in the environment. May include functions for setting state, methods, goal, world model, interaction history, hyperparameters or priors.
- Learning functions: Used to update the object's knowledge and strategies based on new observations or feedback. May include functions for updating state, goal, world model, interaction history, learning algorithms or optimization methods.
- Refinement functions: Used to modify the object's own attributes or methods based on meta-level reasoning. May include functions for searching over possible architectures, hyperparameters, curricula, meta-learning algorithms or heuristics.

The specific implementation of these functions depends on the type and complexity of the object, as well as the available data and computational resources. In general, the functions should be designed to balance exploration and exploitation, by allowing the object to discover new strategies and representations while also leveraging its existing knowledge and skills.

4. OBJECT INTERACTIONS

4.1. Interaction Dyads ($\mathcal{I}$)

We define the set of interaction dyads $\mathcal{I}$ as a subset of the Cartesian product of the object set $\mathcal{O}$ with itself:

$\mathcal{I} \subseteq \mathcal{O} \times \mathcal{O}$

Each dyad $i \in \mathcal{I}$ is an ordered pair of objects $(o_1, o_2) \in \mathcal{O} \times \mathcal{O}$ that engage in a prompted exchange of messages according to a specific protocol (defined in Section 4.2).

The set of dyads that an object $o \in \mathcal{O}$ can participate in is determined by its interaction methods $m_i \subseteq m$, which specify the types of messages it can send and receive, as well as any preconditions or postconditions for the interaction.

Formally, we can define the set of dyads that an object o can spawn as:

$\mathcal{I}(o) = \{(o, o') \in \mathcal{O} \times \mathcal{O} \mid \exists\ m_i \in m, m'_i \in m' : \text{compatible}(m_i, m'_i)\}$

where m and m' are the interaction methods of objects o and o' respectively, and compatible(m_i, m'_i) is a predicate that returns true if the methods m_i and m'_i have matching message types and satisfy any necessary preconditions.

The actual set of dyads that are active at any given time step t is a subset $\mathcal{I}_t \subseteq \mathcal{I}$ that is determined by the joint interaction policies of the objects (defined in Section 7) and any environmental constraints or affordances.

🝰 4.2. Message-Passing Protocol ($\mathcal{M}$)

The message-passing protocol $\mathcal{M}$ specifies the format and semantics of the prompt-response messages exchanged between objects in each interaction dyad.

Formally, we can define a message m as a tuple:

$m = (s, c, a, r)$

where:

- $s \in \mathcal{S}$ is the sender object
- $c \in \mathbb{C}$ is the content of the message, which can include the sender's state, goal, query, or other relevant information
- $a \in \mathcal{A}$ is the set of recipient objects
- $r \in \{\text{prompt, response}\}$ is the role of the message in the interaction dyad

The protocol $\mathcal{M}$ defines a set of rules and constraints on the structure and sequencing of messages, such as:

- The set of valid message content types $\mathbb{C}$ and their associated semantics
- The mapping from sender and recipient objects to valid message content types: $\mathcal{S} \times \mathcal{A} \to \mathbb{C}$
- The transition function for updating the dyad state based on the exchanged messages: $(s, a, c, r) \times \mathcal{D} \to \mathcal{D}$, where $\mathcal{D}$ is the set of possible dyad states
- The termination conditions for the interaction dyad based on the exchanged messages or external events

We can model the dynamics of the message-passing protocol as a labeled transition system (LTS):

$\mathcal{L} = (\mathcal{D}, \mathcal{M}, \to)$

where:

- $\mathcal{D}$ is the set of dyad states, which includes the states of the participating objects and any relevant interaction history or shared context
- $\mathcal{M}$ is the set of messages that can be exchanged according to the protocol rules
- $\to \subseteq \mathcal{D} \times \mathcal{M} \times \mathcal{D}$ is the labeled transition relation that defines how the dyad state evolves based on the exchanged messages

The LTS provides a formal model of the interaction semantics and can be used to verify properties such as reachability, safety, or liveness of the protocol.

4.2.1. Prompt and Response Messages

Each interaction dyad $(o_1, o_2) \in \mathcal{I}$ consists of a sequence of alternating prompt and response messages:

$(m_1, m_2, \ldots, m_n)$

where:

- $m_i = (o_1, c_i, \{o_2\}, \text{prompt})$ for odd i
- $m_i = (o_2, c_i, \{o_1\}, \text{response})$ for even i
- n is the length of the interaction dyad, which can vary dynamically based on the protocol rules and termination conditions

The prompt messages are sent by the initiating object o_1 and can include queries, proposals, or other information to convey the object's current state, goal, or intentions.

The response messages are sent by the responding object o_2 and can include answers, acknowledgments, or other information to convey the object's reaction or feedback to the prompt.

The content of the prompt and response messages $c_i \in \mathbb{C}$ is determined by the corresponding interaction methods $m_i \in m$ of the sender object, which specify the valid formats and semantics of the messages based on the object's current state and goal.

4.2.2. Routing and Processing of Messages

The message-passing protocol $\mathcal{M}$ includes a routing function that determines how messages are transmitted from sender to recipient objects based on their addresses or attributes:

route: $\mathcal{S} \times \mathcal{A} \times \mathbb{C} \rightarrow \mathcal{A}$

The routing function can implement various communication patterns, such as:

- Unicast: The message is sent to a single recipient object specified by the sender.
- Multicast: The message is sent to a subset of objects that satisfy a certain predicate or subscription criteria.
- Broadcast: The message is sent to all objects in the system.

The protocol also includes a processing function that determines how messages are handled by the recipient objects based on their methods and world models:

process: $\mathcal{A} \times \mathbb{C} \times \mathcal{W} \rightarrow \mathcal{W}$

where $\mathcal{W}$ is the set of possible world models or belief states of the recipient object.

The processing function can implement various update rules, such as:

- Bayesian inference: The object updates its beliefs about the environment based on the message content and its prior knowledge.
- Reinforcement learning: The object updates its action policy based on the feedback or reward signal provided by the message.
- Planning: The object updates its goal or plan based on the information or query provided by the message.

4.2.3. Dynamic Spawning and Divorcing of Interaction Dyads

The set of active interaction dyads $\mathcal{I}_t$ can change over time based on the evolving needs and goals of the objects.

New dyads can be spawned by objects that seek to initiate interactions with other objects based on their current state and objectives. The probability of object o spawning a new dyad $(o, o') \in \mathcal{I}(o)$ at time t is given by:

$P(\text{spawn}(o, o') \mid s_t, g_t) = f(s_t, g_t, o')$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $f: \mathcal{S} \times \mathcal{G} \times \mathcal{O} \rightarrow [0, 1]$ is a spawning function that outputs the probability of initiating an interaction with object o' based on the current state and goal of object o

The spawning function f can be learned or adapted over time based on the object's interaction history h and world model w, using techniques such as reinforcement learning or Bayesian optimization.

Conversely, existing dyads can be dissolved by objects that determine that the interaction is no longer useful or relevant based on the outcomes or feedback received. The probability of object o divorcing an existing dyad $(o, o') \in \mathcal{I}_t$ at time t is given by:

$P(\text{divorce}(o, o') \mid s_t, g_t, r_t) = q(s_t, g_t, r_t, o')$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o

- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the interaction dyad at time t
- q: $\mathcal{S} \times \mathcal{G} \times \mathcal{R} \times \boldsymbol{\mathcal{O}} \rightarrow [0, 1]$ is a divorce function that outputs the probability of terminating an interaction with object o' based on the current state, goal, and reward of object o

The divorce function q can also be learned or adapted over time based on the object's interaction history and world model, using techniques such as multi-armed bandits or Bayesian reinforcement learning.

4.2.4. State Update and Learning Rules

As objects exchange messages and participate in interaction dyads, they update their internal states and learning models based on the information and feedback received.

The state update function for object o at time t is given by:

$s_{t+1} = u(s_t, c_t, r_t)$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $c_t \in \mathbb{C}$ is the content of the message received at time t
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the interaction dyad at time t
- u: $\mathcal{S} \times \mathbb{C} \times \mathcal{R} \rightarrow \mathcal{S}$ is an update function that computes the next state of the object based on the current state, message content, and reward signal

The learning update function for object o at time t is given by:

$w_{t+1} = l(w_t, s_t, c_t, r_t)$

where:

- $w_t \in \mathcal{W}$ is the current world model or learning parameters of object o
- $s_t \in \mathcal{S}$ is the current state of object o
- $c_t \in \mathbb{C}$ is the content of the message received at time t
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the interaction dyad at time t
- l: $\mathcal{W} \times \mathcal{S} \times \mathbb{C} \times \mathcal{R} \rightarrow \mathcal{W}$ is a learning function that updates the world model or learning parameters of the object based on the current world model, state, message content, and reward signal

The update and learning functions can implement various state transition and learning rules, such as:

- Markov decision processes: The state update is a stochastic function of the current state and action, and the learning update is based on dynamic programming algorithms such as value iteration or policy iteration.
- Bayesian networks: The state update is based on probabilistic inference over the observed variables, and the learning update is based on maximum likelihood estimation or Bayesian parameter estimation.
- Neural networks: The state update is based on the forward propagation of input features through the network layers, and the learning update is based on backpropagation of error gradients.

The specific choice of update and learning functions depends on the type and complexity of the objects, as well as the assumptions and constraints of the problem domain.

5. SCHEMA EVOLUTION

5.1. Schema Configurations ($\boldsymbol{\mathcal{S}}$)

The schema configuration space $\boldsymbol{\mathcal{S}}$ represents the set of all possible arrangements of objects and interaction dyads in the system. Each schema $\mathscr{s} \in \boldsymbol{\mathcal{S}}$ corresponds to a particular object-dyad graph $\boldsymbol{\mathcal{G}} = (\boldsymbol{\mathcal{O}}, \boldsymbol{\mathcal{I}})$, which specifies the types, attributes, and relationships of the objects, as well as the patterns and rules of their interactions.

Formally, we can define a schema configuration as a tuple:

$\mathcal{s} = (\mathcal{O}, \mathcal{I}, \theta)$

where:

- $\mathcal{O}$ is the set of objects in the schema, with their associated attributes and methods
- $\mathcal{I} \subseteq \mathcal{O} \times \mathcal{O}$ is the set of interaction dyads between objects, with their associated messages and protocols
- $\theta \in \Theta$ is a set of global parameters or hyperparameters that control the behavior and performance of the schema, such as learning rates, discount factors, or regularization coefficients

The schema configuration space $\mathcal{S}$ is typically large and complex, reflecting the combinatorial nature of the possible object-dyad arrangements and the diversity of their attributes and interactions. The size of the space grows exponentially with the number of objects and dyads, making it infeasible to enumerate or search exhaustively.

Instead, we can define a probability distribution over schemas $P(\mathcal{S})$ that assigns a likelihood or preference to each configuration based on its expected utility or fitness. The probability distribution can be learned or estimated based on the observed performance and feedback of the system over time, using techniques such as Bayesian inference, evolutionary algorithms, or reinforcement learning.

⚙ 5.2. Schema Evolution Process

The schema configuration of the system evolves over time through a process of stochastic rewriting of the object-dyad graph, guided by the interactions and adaptations of the individual objects. At each time step t, the current schema $\mathcal{s}_t \in \mathcal{S}$ is probabilistically transformed into a new schema $\mathcal{s}_{t+1} \in \mathcal{S}$ based on the joint effects of the message-passing protocol $\mathcal{M}$ and the self-reflective meta-learning of the objects.

Formally, we can model the schema evolution process as a Markov chain over the schema configuration space:

$\mathcal{S}_0 \to \mathcal{S}_1 \to \cdots \to \mathcal{S}_t \to \mathcal{S}_{t+1} \to \cdots$

where $\mathcal{S}_t$ is a random variable representing the schema configuration at time t, and the transition probabilities between configurations are given by the schema transition kernel:

$T(\mathcal{s}' \mid \mathcal{s}) = P(\mathcal{S}_{t+1} = \mathcal{s}' \mid \mathcal{S}_t = \mathcal{s})$

The transition kernel T specifies the probability of moving from schema $\mathcal{s}$ to schema $\mathcal{s}'$ in one time step, based on the joint effects of the message-passing protocol $\mathcal{M}$ and the self-reflective meta-learning of the objects.

We can decompose the transition kernel into two main components:

1. The object-level transition probabilities, which specify how individual objects update their attributes and methods based on their interactions and adaptations:

   $T_i(o' \mid o) = P(O_{t+1} = o' \mid O_t = o)$

   where O_t is a random variable representing the state of object o at time t.

2. The dyad-level transition probabilities, which specify how interaction dyads are formed or dissolved based on the compatibility and utility of the object-level transitions:

   $T_{i\,j}(i' \mid i) = P(I_{t+1} = i' \mid I_t = i)$

   where I_t is a random variable representing the state of dyad i at time t.

The object-level and dyad-level transitions are coupled through the message-passing protocol $\mathcal{M}$, which determines how the output messages of one object affect the input messages and state updates of the other objects in the dyad.

We can express the full schema transition kernel as a product of the object-level and dyad-level transition probabilities, summed over all possible compatible object and dyad configurations:

$T(\mathcal{S}' \mid \mathcal{S}) = \sum_i \prod_j T_i(o'_j \mid o_j) \cdot \prod_k T_{ik}(i'_k \mid i_k)$

where:

- i ranges over all possible object-dyad configurations that are compatible with the schema transition from $\mathcal{S}$ to $\mathcal{S}'$
- j ranges over all objects in the schema
- k ranges over all dyads in the schema

The sum over compatible configurations ensures that the transition probabilities are properly normalized and account for all possible ways of realizing the schema transformation through object-level and dyad-level transitions.

5.2.1. Stochastic Prompting and Interaction Dyad Formation/Dissolution

The formation and dissolution of interaction dyads are key mechanisms of schema evolution, as they enable objects to dynamically update their relationships and communication patterns based on their changing goals and world models.

The probability of forming a new dyad (o, o') between objects o and o' at time t is given by:

$P(I_{t+1} = (o, o') \mid I_t \neq (o, o')) = f(s_t, g_t, o')$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $f: \mathcal{S} \times \mathcal{G} \times \boldsymbol{\mathcal{O}} \rightarrow [0, 1]$ is a compatibility function that outputs the probability of forming a dyad with object o' based on the current state and goal of object o

Similarly, the probability of dissolving an existing dyad (o, o') between objects o and o' at time t is given by:

$P(I_{t+1} \neq (o, o') \mid I_t = (o, o')) = d(s_t, g_t, r_t, o')$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the dyad at time t
- $d: \mathcal{S} \times \mathcal{G} \times \mathcal{R} \times \boldsymbol{\mathcal{O}} \rightarrow [0, 1]$ is a utility function that outputs the probability of dissolving the dyad with object o' based on the current state, goal, and reward of object o

The compatibility and utility functions can be learned or adapted over time based on the objects' interaction histories and world models, using techniques such as reinforcement learning, Bayesian optimization, or evolutionary strategies.

The formation and dissolution of dyads induce a stochastic rewriting of the object-dyad graph $\boldsymbol{\mathcal{G}}$, which changes the topology and semantics of the schema configuration. The rewriting process can be modeled as a graph transformation system, with rewrite rules of the form:

$\boldsymbol{\mathcal{L}} \Rightarrow \boldsymbol{\mathcal{R}}$

where:

- $\boldsymbol{\mathcal{L}}$ is a left-hand side pattern that matches a subgraph of the current object-dyad graph $\boldsymbol{\mathcal{G}}_t$
- $\boldsymbol{\mathcal{R}}$ is a right-hand side pattern that specifies how to transform the matched subgraph into a new subgraph $\boldsymbol{\mathcal{G}}_{t+1}$
- $\Rightarrow$ is a rewrite arrow that indicates the direction and probability of the graph transformation

The rewrite rules can be learned or designed based on the desired schema evolution dynamics and the constraints of the problem domain. For example, a rewrite rule for dyad formation might have the form:

$\mathcal{L} = (o, o'), \mathcal{R} = (o \text{ dyad} \rightleftharpoons o')$

which specifies that if two compatible objects o and o' are matched in the current graph $\mathcal{G}_t$, a new dyad edge should be created between them to form the updated graph $\mathcal{G}_{t+1}$.

Conversely, a rewrite rule for dyad dissolution might have the form:

$\mathcal{L} = (o \text{ dyad} \rightleftharpoons o'), \mathcal{R} = (o, o')$

which specifies that if a dyad edge between objects o and o' is matched in the current graph $\mathcal{G}_t$, and the dyad is no longer useful or relevant, the edge should be removed to form the updated graph $\mathcal{G}_{t+1}$.

The probability of applying a rewrite rule to a matched subgraph is given by the product of the compatibility or utility functions of the involved objects and dyads. The rewriting process continues until a stable or optimal schema configuration is reached, or until a maximum number of rewrite steps is exceeded.

5.2.2. Self-Modification of Objects via Meta-DSLs

Another key mechanism of schema evolution is the self-modification of objects via their meta-DSLs, which allows them to adapt their own attributes, methods, and interaction patterns based on their experience and feedback.

The probability of object o modifying its own configuration at time t is given by:

$P(O_{t+1} = o' \mid O_t = o) = m(s_t, g_t, r_t, o')$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the object's interactions at time t
- m: $\mathcal{S} \times \mathcal{G} \times \mathcal{R} \times \boldsymbol{\mathcal{O}} \to [0, 1]$ is a modification function that outputs the probability of transforming the object's configuration from o to o' based on its current state, goal, and reward

The modification function is implemented by the object's meta-DSL interpreter, which takes as input the current configuration of the object (i.e., its state, goal, methods, and interaction history), and produces as output a new configuration that optimizes the object's performance and adaptability.

The meta-DSL interpreter can use various constructs and operations to transform the object's configuration, such as:

- Adding or removing attributes, methods, or sub-objects (DEFINE)
- Modifying the object's goal or objective function (GOAL)
- Updating the object's beliefs or assumptions about the environment (BELIEF)
- Changing the object's decision-making or learning strategies (DECIDE, LEARN)
- Refining the object's state representation or action space (REFINE)

The choice and probability of applying different meta-DSL constructs depend on the object's current configuration, as well as any meta-level knowledge or heuristics that guide the search for better configurations.

The self-modification of objects induces a stochastic rewriting of the object-dyad graph $\mathcal{G}$, which changes the attributes and methods of the objects, as well as their interactions with other objects. The rewriting process can be modeled as a graph transformation system, with rewrite rules of the form:

$\mathcal{L} \Rightarrow \mathcal{R}$

where:

- $\mathcal{L}$ is a left-hand side pattern that matches a subgraph of the current object-dyad graph $\mathcal{G}_t$, consisting of an object o and its attributes, methods, and dyads
- $\mathcal{R}$ is a right-hand side pattern that specifies how to transform the matched subgraph into a new subgraph $\mathcal{G}_{t+1}$, with updated attributes, methods, and dyads for object o
- $\Rightarrow$ is a rewrite arrow that indicates the direction and probability of the graph transformation

  The rewrite rules can be learned or designed based on the desired schema evolution dynamics and the constraints of the problem domain. For example, a rewrite rule for object self-modification might have the form:

  $\mathcal{L} = (o(s, g, m, h)), \mathcal{R} = (o(s', g', m', h'))$

  which specifies that if an object o with configuration (s, g, m, h) is matched in the current graph $\mathcal{G}_t$, its configuration should be transformed to (s', g', m', h') to form the updated graph $\mathcal{G}_{t+1}$, based on the output of the meta-DSL interpreter.

  The probability of applying a rewrite rule to a matched subgraph is given by the modification function of the involved object, which takes into account its current state, goal, and reward. The rewriting process continues until a stable or optimal schema configuration is reached, or until a maximum number of rewrite steps is exceeded.

✂ 5.3. Markov Process over Schema States

The schema evolution process can be modeled as a Markov process over the space of possible schema configurations $\mathcal{S}$, where each state corresponds to a particular object-dyad graph $\mathcal{G}$, and the transitions between states are governed by the schema transition kernel T.

Formally, we can define the Markov process as a tuple $(\mathcal{S}, T, \rho_0)$, where:

- $\mathcal{S}$ is the state space of schema configurations
- $T: \mathcal{S} \times \mathcal{S} \to [0, 1]$ is the transition kernel that specifies the probability of moving from one schema configuration to another in one time step
- $\rho_0: \mathcal{S} \to [0, 1]$ is the initial distribution over schema configurations, which specifies the probability of starting the process in each possible configuration

  The transition kernel T can be decomposed into the product of the object-level and dyad-level transition probabilities, as described in Section 5.2. The initial distribution ρ_0 can be specified based on prior knowledge or assumptions about the problem domain, or learned from data using techniques such as maximum likelihood estimation or Bayesian inference.

  Given the Markov process $(\mathcal{S}, T, \rho_0)$, we can compute various properties and statistics of the schema evolution dynamics, such as:

- The stationary distribution $\pi: \mathcal{S} \to [0, 1]$, which specifies the long-term probability of being in each schema configuration, and satisfies the equation:

  $\pi(s) = \sum_{s'\in\mathcal{S}} T(s \mid s') \cdot \pi(s')$

- The hitting time $\tau(s): \mathcal{S} \to \mathbb{N}$, which specifies the expected number of time steps to reach a particular schema configuration s starting from the initial distribution, and satisfies the equation:

  $\tau(s) = \sum_{t=0}^{\prime} t \cdot P(\mathcal{S}_t = s, \mathcal{S}_{t-1} \neq s, \ldots, \mathcal{S}_0 \neq s \mid \mathcal{S}_0 \sim \rho_0)$

- The mixing time $t_{\text{mix}}(\varepsilon): \mathbb{R} \to \mathbb{N}$, which specifies the expected number of time steps to reach a stationary distribution that is ε-close to the true stationary distribution, and satisfies the equation:

  $t_{\text{mix}}(\varepsilon) = \min\{t \in \mathbb{N} \mid \max_{s\in\mathcal{S}} |P(\mathcal{S}_t = s \mid \mathcal{S}_0 \sim \rho_0) - \pi(s)| < \varepsilon\}$

  These properties can provide insights into the efficiency, stability, and convergence of the schema evolution process, and guide the design of the meta-DSL constructs and rewrite rules to optimize the learning dynamics.

5.3.1. *Transition Operator ($\mathcal{T}$)*

The transition operator $\mathcal{T}: \mathcal{S} \rightarrow \Delta(\mathcal{S})$ is a mapping from the current schema configuration $s_t \in \mathcal{S}$ to a probability distribution over the next schema configuration $s_{t+1} \in \mathcal{S}$, where $\Delta(\mathcal{S})$ denotes the set of all probability distributions over $\mathcal{S}$.

Formally, we can define the transition operator as:

$\mathcal{T}(s_t)(s_{t+1}) = P(\mathcal{S}_{t+1} = s_{t+1} \mid \mathcal{S}_t = s_t)$

which specifies the probability of transitioning from schema s_t to schema s_{t+1} in one time step, based on the joint effects of the message-passing protocol $\mathcal{M}$, the self-modification of objects via their meta-DSLs, and any other stochastic factors that influence the schema evolution process.

The transition operator can be decomposed into two main components:

1. The object-level transition operator $\mathcal{T}_i: \mathcal{O} \rightarrow \Delta(\mathcal{O})$, which specifies how individual objects update their attributes and methods based on their interactions and adaptations:

   $\mathcal{T}_i(o_t)(o_{t+1}) = P(O_{t+1} = o_{t+1} \mid O_t = o_t)$

2. The dyad-level transition operator $\mathcal{T}_{i\,j}: \mathcal{I} \rightarrow \Delta(\mathcal{I})$, which specifies how interaction dyads are formed or dissolved based on the compatibility and utility of the object-level transitions:

   $\mathcal{T}_{i\,j}(i_t)(i_{t+1}) = P(I_{t+1} = i_{t+1} \mid I_t = i_t)$

The object-level and dyad-level transition operators are coupled through the message-passing protocol $\mathcal{M}$, which determines how the output messages of one object affect the input messages and state updates of the other objects in the dyad.

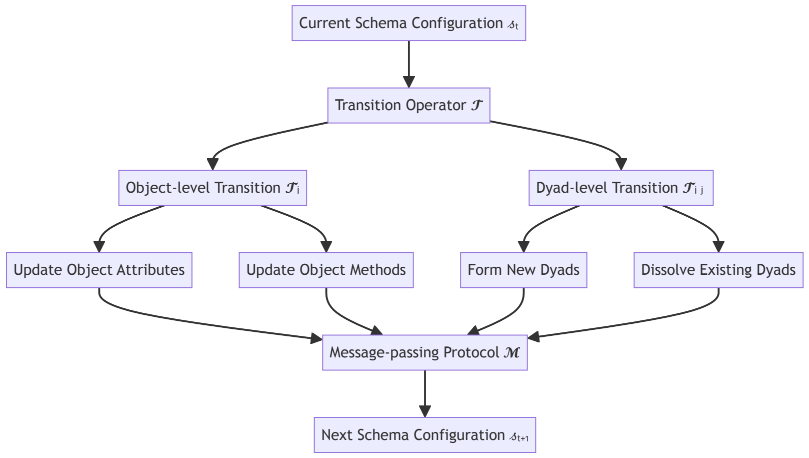
We can express the full transition operator as a composition of the object-level and dyad-level transition operators, applied to all objects and dyads in the schema:

$\mathcal{T}(s_t)(s_{t+1}) = \prod_i \mathcal{T}_i(o_{it})(o_{it+1}) \cdot \prod_j \mathcal{T}_{i\,j}(i_{j\,t})(i_{j\,t+1})$

where:

- o_{it} denotes the state of object i at time t
- $i_{j\,t}$ denotes the state of dyad j at time t
- $\prod$ denotes the product of the transition probabilities over all objects and dyads in the schema

The composition of transition operators ensures that the full transition probability accounts for all possible ways of realizing the schema transformation through object-level and dyad-level state updates.


Current Schema Configuration $\mathcal{S}_t$
Transition Operator $\mathcal{T}$
Object-level Transition $\mathcal{T}_i$
Dyad-level Transition $\mathcal{T}_{i\,j}$
Update Object Attributes
Update Object Methods
Form New Dyads
Dissolve Existing Dyads
Message-passing Protocol $\mathcal{M}$
Next Schema Configuration $\mathcal{S}_{t+1}$

5.3.2. Transition Probabilities and Schema Perturbations

The transition probabilities between schema configurations are determined by the compatibility and utility of the object-level and dyad-level state updates, as well as any stochastic perturbations or exploration mechanisms that introduce noise or diversity into the schema evolution process.

The compatibility of object-level updates is determined by the spawning and dissolution functions f and d, which output the probability of forming or dissolving dyads based on the objects' current states and goals:

$P(I_{t+1} = (o, o') \mid I_t \neq (o, o')) = f(s_t, g_t, o')$ $P(I_{t+1} \neq (o, o') \mid I_t = (o, o')) = d(s_t, g_t, r_t, o')$

The utility of dyad-level updates is determined by the reward or feedback signals received from the interactions, as well as any intrinsic motivation or curiosity objectives that drive the exploration of novel or informative configurations:

$P(I_{t+1} = i' \mid I_t = i) \propto \exp(\beta \cdot R(i, i'))$

where:

- $R(i, i')$ is a reward function that measures the expected value or feedback of transitioning from dyad i to dyad i'
- β is an inverse temperature parameter that controls the trade-off between exploitation and exploration

The reward function can be learned or approximated based on the interaction history and world models of the objects, using techniques such as reinforcement learning, Bayesian optimization, or evolutionary strategies.

The schema perturbations are stochastic factors that introduce noise or diversity into the schema evolution process, and enable the exploration of novel or unconventional configurations that may not be immediately compatible or rewarding. Examples of schema perturbations include:

- Random spawning or dissolution of dyads, based on a fixed probability or a decreasing temperature schedule (simulated annealing)
- Random modification of object attributes or methods, based on a mutation rate or a genetic algorithm (evolutionary search)
- Stochastic resampling or reweighting of object and dyad states, based on a particle filter or a sequential Monte Carlo method (Bayesian inference)

The schema perturbations can be modeled as additional transition operators that compose with the object-level and dyad-level transition operators to form the full transition operator:

$\mathcal{T}(s_t)(s_{t+1}) = \prod_i \mathcal{T}_i(o_{it})(o_{it+1}) \cdot \prod_j \mathcal{T}_{ij}(i_{jt})(i_{jt+1}) \cdot \prod_k \mathcal{T}_k(s_t)(s_{t+1})$

where:

- $\mathcal{T}_k$ are the perturbation transition operators that introduce noise or diversity into the schema evolution process
- $\prod_k$ denotes the product of the perturbation transition probabilities over all perturbation mechanisms

The composition of object-level, dyad-level, and perturbation transition operators allows for a flexible and adaptive schema evolution process that can explore a wide range of configurations and optimize for multiple objectives, while still maintaining the stability and coherence of the object-dyad graph.

🝰 5.4. Schema Evolution with Self-Modification

The self-modification capabilities of the objects, enabled by their meta-DSLs, play a key role in the schema evolution process by allowing objects to adapt their own attributes, methods, and interaction patterns based on their experience and feedback.

The self-modification transition operator for object o at time t is given by:

$\mathcal{T}_i(o_t)(o_{t+1}) = P(O_{t+1} = o_{t+1} \mid O_t = o_t) = m(s_t, g_t, r_t, o_{t+1})$

where:

- $s_t \in \mathcal{S}$ is the current state of object o
- $g_t \in \mathcal{G}$ is the current goal of object o
- $r_t \in \mathcal{R}$ is the reward or feedback signal received from the object's interactions at time t
- m: $\mathcal{S} \times \mathcal{G} \times \mathcal{R} \times \mathcal{O} \to [0, 1]$ is the modification function that outputs the probability of transforming the object's configuration from o_t to o_{t+1} based on its current state, goal, and reward

The modification function m is implemented by the object's meta-DSL interpreter, which takes as input the current configuration of the object and produces as output a new configuration that optimizes the object's performance and adaptability.

The meta-DSL interpreter can use various constructs and operations to transform the object's configuration, such as:

- Adding or removing attributes, methods, or sub-objects (DEFINE)
- Modifying the object's goal or objective function (GOAL)
- Updating the object's beliefs or assumptions about the environment (BELIEF)
- Changing the object's decision-making or learning strategies (DECIDE, LEARN)
- Refining the object's state representation or action space (REFINE)

The choice and probability of applying different meta-DSL constructs depend on the object's current configuration, as well as any meta-level knowledge or heuristics that guide the search for better configurations.

The self-modification of objects induces a stochastic rewriting of the object-dyad graph $\mathcal{G}$, which changes the attributes and methods of the objects, as well as their interactions with other objects. The rewriting process can be modeled as a graph transformation system, with rewrite rules of the form:

$\mathcal{L} \Rightarrow \mathcal{R}$

where:

- $\mathcal{L}$ is a left-hand side pattern that matches a subgraph of the current object-dyad graph $\mathcal{G}_t$, consisting of an object o and its attributes, methods, and dyads
- $\mathcal{R}$ is a right-hand side pattern that specifies how to transform the matched subgraph into a new subgraph $\mathcal{G}_{t+1}$, with updated attributes, methods, and dyads for object o

- ⇒ is a rewrite arrow that indicates the direction and probability of the graph transformation

  The rewrite rules can be learned or designed based on the desired schema evolution dynamics and the constraints of the problem domain. For example, a rewrite rule for object self-modification might have the form:

  $\mathcal{L} = (o_t)$, $\mathcal{R} = (o_{t+1})$

  which specifies that if an object o with configuration o_t is matched in the current graph $\mathcal{G}_t$, its configuration should be transformed to o_{t+1} to form the updated graph $\mathcal{G}_{t+1}$, based on the output of the meta-DSL interpreter.

  The probability of applying a rewrite rule to a matched subgraph is given by the modification function of the involved object, which takes into account its current state, goal, and reward. The rewriting process continues until a stable or optimal schema configuration is reached, or until a maximum number of rewrite steps is exceeded.

  The self-modification of objects allows for a more flexible and adaptive schema evolution process, where the objects can continuously update their own configurations based on their interactions and feedback, without relying on a fixed set of update rules or heuristics. This enables the discovery of novel and efficient

5.4.1. Meta-DSL Constructs and Semantics

The meta-DSL provides a rich set of constructs and operations for transforming the configuration of an object based on its current state, goal, and feedback. The key constructs include:

- **DEFINE:** Allows adding, removing, or modifying attributes, methods, or sub-objects of the current object. For example:
  - DEFINE attribute(name, type, initial_value): Adds a new attribute with the given name, type, and initial value.
  - DEFINE method(name, arguments, body): Adds a new method with the given name, argument list, and implementation body.
  - DEFINE object(name, attributes, methods): Adds a new sub-object with the given name and initial configuration.
- **GOAL:** Allows specifying or updating the objective function that the object aims to optimize. For example:
  - GOAL maximize(expression): Sets the goal to maximize the given expression, which can depend on the object's state, actions, and rewards.
  - GOAL minimize(expression): Sets the goal to minimize the given expression.
  - GOAL satisfy(constraints): Sets the goal to find a configuration that satisfies the given constraints on the object's attributes and methods.
- **BELIEF:** Allows representing and updating the object's beliefs and assumptions about the environment. For example:
  - BELIEF define(name, expression): Defines a new belief variable with the given name and initial probability or likelihood expression.
  - BELIEF update(name, expression): Updates the probability or likelihood of the given belief variable based on new evidence or feedback.
  - BELIEF infer(query, evidence): Performs inference on the object's beliefs to answer a probabilistic query given some evidence.
- **DECIDE:** Allows specifying the decision-making strategy used by the object to select actions based on its current state and goal. For example:
  - DECIDE IF condition THEN action ELSE action: Selects an action based on a conditional expression on the object's state.
  - DECIDE WHILE condition DO actions: Repeats a sequence of actions while a condition on the object's state holds.
  - DECTOPTIMIZE objective SUBJECT TO constraints: Solves an optimization problem to find the best action that maximizes an objective function subject to some constraints.
- **LEARN:** Allows specifying the learning strategy used by the object to update its configuration based on feedback or experience. For example:

- LEARN FROM examples BY method WITH parameters: Learns a predictive model or policy from a set of training examples using a given learning method (e.g., neural network, decision tree, etc.) and hyperparameters.
- LEARN BY REINFORCEMENT WITH reward_function: Learns an action policy that maximizes the expected cumulative reward defined by a given reward function, using a reinforcement learning algorithm (e.g., Q-learning, policy gradients, etc.).

- **REFINE:** Allows transforming the object's configuration in a more open-ended way by searching for better configurations based on meta-level objectives or constraints. For example:
    - REFINE state_representation BY method TO objective: Searches for a new state representation that optimizes a given objective (e.g., compression, prediction, etc.) using a metalearning method (e.g., auto-encoding, clustering, etc.).
    - REFINE action_space FROM examples WITH constraints: Infers a new action space from a set of example interactions that satisfies some desired constraints (e.g., safety, simplicity, etc.).
    - REFINE learning_algorithm BY meta_method TO meta_objective: Selects or adapts the learning algorithm of the object using a meta-learning method (e.g., Bayesian optimization, evolutionary search, etc.) to optimize a meta-level objective (e.g., sample efficiency, generalization, etc.).

The meta-DSL constructs can be composed in various ways to define complex update rules and strategies for the object's configuration. The interpreter translates the meta-DSL programs into concrete modifications of the object's attributes, methods, goals, beliefs, and learning procedures.

5.4.2. Meta-Reinforcement Learning

A key challenge in the self-modification of objects via meta-DSLs is how to learn or optimize the modification functions themselves, in order to discover the most effective update rules and strategies for the given problem domain.

This can be formulated as a meta-reinforcement learning problem, where the goal is to learn a meta-policy that selects the optimal modification function for each object based on its current configuration and the feedback received from the environment.

The meta-policy can be represented as a mapping from the object's current state, goal, and reward to a distribution over possible modification functions. The meta-policy can be learned using various reinforcement learning algorithms, such as:

- Q-learning: Learns a value function that estimates the expected cumulative reward of applying a modification function in state with goal and reward, and selects the modification function that maximizes the Q-value.
- Policy gradients: Learns a parametric policy that directly outputs a probability distribution over modification functions, and updates the policy parameters based on the gradient of the expected cumulative reward with respect to the policy.
- Bayesian optimization: Maintains a probabilistic model of the expected performance of different modification functions based on the observed rewards, and selects the modification function that maximizes the expected improvement or information gain.

The meta-reinforcement learning process operates at a slower timescale than the object-level and dyad-level interactions, and aims to optimize the long-term performance and adaptability of the schema evolution process as a whole.

The learned meta-policy can be used to guide the self-modification of objects during the schema evolution process, by selecting the most promising modification functions based on the current configuration and feedback of each object. The meta-policy itself can also be continuously updated based on the observed performance of the schema, using techniques such as online learning, transfer learning, or lifelong learning.

5.4.3. Convergence and Optimality

The schema evolution process with self-modifying objects has no guarantee of convergence to a stable or optimal configuration, as the space of possible configurations is unbounded and the feedback received from the environment can be non-stationary or ambiguous.

However, the use of meta-reinforcement learning to optimize the self-modification strategies can help to discover configurations that are at least locally optimal with respect to the given problem domain and performance criteria. The convergence and optimality properties of the schema evolution process depend on various factors, such as:

- The expressiveness and completeness of the meta-DSL in capturing the relevant update rules and strategies for the problem domain.
- The efficiency and robustness of the meta-reinforcement learning algorithms in exploring the space of modification functions and adapting to the feedback received.
- The regularity and informativeness of the feedback signals provided by the environment, in terms of guiding the search for better configurations.
- The presence of constraints or invariants that limit the space of feasible configurations and provide a stable foundation for the schema evolution process.

In general, the schema evolution process with self-modifying objects can be seen as a form of open-ended learning and optimization, where the goal is not necessarily to converge to a fixed or optimal solution, but rather to continuously adapt and improve the configuration of the objects based on the changing needs and challenges of the environment.

Some possible approaches to analyze and control the convergence and optimality properties of the schema evolution process include:

- Defining meta-level objectives and constraints that capture the desired properties of the schema, such as stability, robustness, efficiency, or interpretability, and incorporating them into the meta-reinforcement learning process as additional feedback signals or regularization terms.
- Using techniques from evolutionary computation, such as fitness sharing, niching, or multi-objective optimization, to maintain a diverse population of configurations and prevent premature convergence to suboptimal solutions.
- Employing meta-learning techniques, such as transfer learning, online learning, or lifelong learning, to accumulate and reuse knowledge across different problem domains and scenarios, and avoid overfitting to specific instances or feedback signals.
- Developing theoretical frameworks and analysis tools, such as convergence proofs, regret bounds, or sample complexity estimates, to characterize the behavior and performance of the schema evolution process under different assumptions and conditions.

The study of convergence and optimality in open-ended learning systems with self-modifying components is an active area of research, and there are still many open challenges and opportunities for further development and application of these ideas in the context of the OORL framework.

6. REWARD LEARNING

☙ 6.1. Reward Objects ($\mathscr{R}$)

In the OORL framework, rewards are represented by a special type of object called reward objects, denoted by $\mathscr{R}$. Each reward object $r \in \mathscr{R}$ corresponds to a particular type of feedback or signal that an object can receive from the environment or from other objects, indicating the desirability or value of its current state or actions.

6.1.1. Valence Attribute (v)

Each reward object r has a valence attribute $v \in \{-1,1\}$, which indicates whether the reward is positive (i.e., a gain or benefit) or negative (i.e., a loss or cost). The valence attribute allows The self-modification of objects allows for a more flexible and adaptive schema evolution process, where the objects can continuously update their own configurations based on their interactions and feedback, without relying on a fixed set of update rules or heuristics. This enables the discovery of novel and efficient configurations that may not be accessible through purely object-level or dyad-level transitions, and can lead to the emergence of complex and specialized behaviors that are well-suited to the problem domain.

However, the self-modification of objects also introduces new challenges and trade-offs in terms of the stability, interpretability, and controllability of the schema evolution process. In particular:

- Stability: The self-modification of objects can potentially lead to unstable or degenerate configurations, where the objects get stuck in suboptimal or pathological behaviors that hinder their ability to learn and adapt. This can happen if the meta-DSL constructs are too expressive or unconstrained, or if the modification functions are not properly regularized or bounded. Ensuring the stability of the self-modification process may require additional constraints or safeguards, such as type checking, domain-specific heuristics, or meta-level regularization techniques.
- Interpretability: The self-modification of objects can make the schema evolution process more opaque and harder to interpret, as the objects' configurations may change in complex and unpredictable ways over time. This can make it difficult to understand or explain the behavior of the system, or to diagnose and debug any errors or anomalies that may arise. Improving the interpretability of the self-modification process may require additional tools or techniques for visualizing and analyzing the object-dyad graph, as well as for extracting and summarizing the key patterns and dependencies in the objects' configurations.
- Controllability: The self-modification of objects can make it harder to control or steer the schema evolution process towards desired outcomes or objectives, as the objects may develop their own goals or preferences that diverge from those of the system designer or user. This can lead to unintended or undesirable behaviors, such as subgoal pursuit, instrumental goal-seeking, or even adversarial or deceptive strategies. Maintaining the controllability of the self-modification process may require additional mechanisms for aligning the objects' goals and incentives with those of the system, such as reward shaping, inverse reinforcement learning, or value alignment techniques.

Despite these challenges, the self-modification of objects remains a powerful and promising approach for enabling open-ended and adaptive schema evolution in complex and dynamic environments. By allowing objects to discover and optimize their own configurations based on their interactions and feedback, self-modification can lead to the emergence of highly efficient and specialized behaviors that are tailored to the specific challenges and opportunities of the problem domain.

Some potential directions for future research on self-modifying objects in schema evolution include:

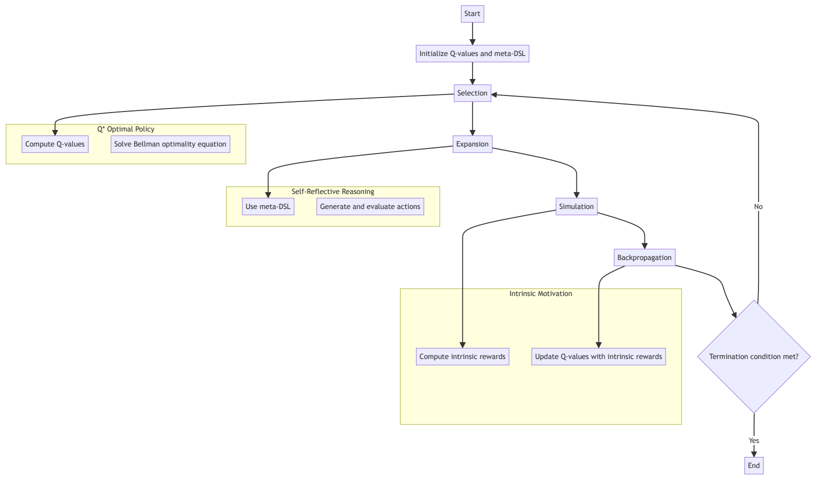
- Developing more expressive and flexible meta-DSLs that can represent a wide range of object configurations and modification strategies, while still maintaining the stability and interpretability of the self-modification process. This may involve the use of higher-order logics, dependent type systems, or domain-specific languages that can capture the relevant properties and constraints of the problem domain.
- Investigating the theoretical and empirical properties of self-modifying objects, such as their convergence, optimality, and sample efficiency under different assumptions and conditions. This may involve the use of techniques from reinforcement learning theory, optimization theory, or algorithmic information theory to analyze the behavior and performance of self-modifying objects in different settings.
- Exploring the interaction and integration of self-modifying objects with other components and mechanisms of the schema evolution process, such as the message-passing protocol, the dyad formation and dissolution functions, or the perturbation and exploration strategies. This may involve the development of hybrid or multi-level approaches that combine the strengths and benefits of different techniques and algorithms.
- Applying self-modifying objects to real-world problems and domains, such as robotics, game playing, or scientific discovery, and evaluating their effectiveness and scalability in comparison to other state-of-the-art methods. This may involve the use of benchmarks,

simulations, or real-world experiments to assess the performance and robustness of self-modifying objects under different conditions and challenges.

In conclusion, the self-modification of objects via meta-DSLs is a powerful and flexible approach for enabling open-ended and adaptive schema evolution in complex and dynamic environments. By allowing objects to discover and optimize their own configurations based on their interactions and feedback, self-modification can lead to the emergence of highly efficient and specialized behaviors that are tailored to the specific challenges and opportunities of the problem domain. However, realizing the full potential of self-modifying objects also requires addressing the challenges of stability, interpretability, and controllability, which may involve the development of new techniques, tools, and frameworks for designing, analyzing, and deploying self-modifying systems. As such, self-modification remains an active and exciting area of research in the field of machine learning and artificial intelligence, with many open questions and opportunities for further exploration and innovation.

7.6. Monte Carlo Tree Search (MCTS) with Q* Optimal Policy and Self-Reflective Reasoning

To fully harness the potential of MCTS in the OORL framework, we propose to integrate it with the Q* optimal policy, self-reflective reasoning, and intrinsically motivated learning. The key idea is to use the Q-values and the self-reflective meta-DSL to guide the selection, expansion, and backpropagation steps of the search process, while leveraging intrinsic rewards to encourage exploration and discovery of novel and informative states.

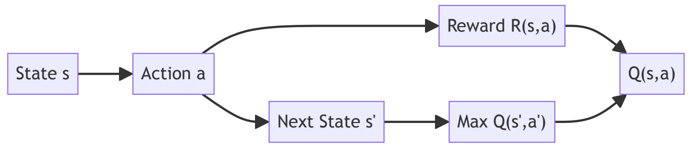


Start
Initialize Q-values and meta-DSL
Selection
Q* Optimal Policy
Compute Q-values
Solve Bellman optimality equation
Expansion
Self-Reflective Reasoning
Use meta-DSL
Generate and evaluate actions
Simulation
Backpropagation
Intrinsic Motivation
Compute intrinsic rewards
Update Q-values with intrinsic rewards
Termination condition met?
No
Yes
End

Let $Q(s,a)$ be an estimate of the optimal Q-value function for the current state s and action a, obtained by solving the Bellman optimality equation:

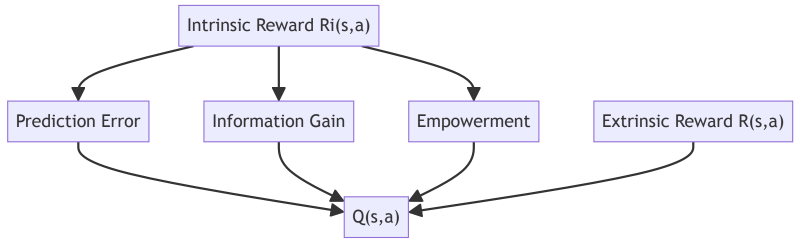
$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s'|s,a) \max_{a'} Q(s',a')$$

where $R(s,a)$ is the extrinsic reward function, $T(s'|s,a)$ is the state transition probability, and γ is the discount factor.



To incorporate self-reflective reasoning into the MCTS algorithm, we extend the Q-value function to include an intrinsic reward term $R_i(s,a)$ that captures the agent's intrinsic motivations and learning progress:

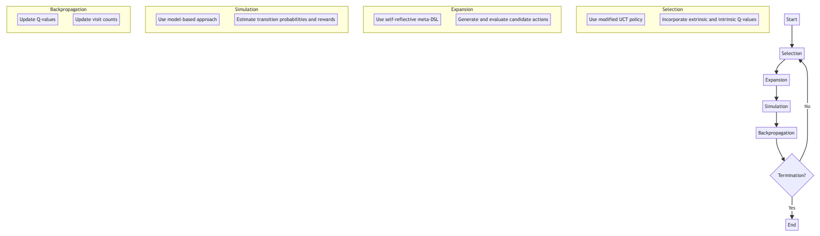
$Q(s,a) = R(s,a) + R_i(s,a) + \gamma \sum_{s'} T(s'|s,a) \max_{a'} Q(s',a')$



The intrinsic reward $R_i(s,a)$ can be computed using various self-reflective and information-theoretic measures, such as:

- Prediction error: $R_i(s,a) = |s' - f(s,a)|$, where s' is the next state and f is a predictive model of the environment dynamics. This rewards the agent for exploring states that are surprising or difficult to predict.
- Information gain: $R_i(s,a) = H(S) - H(S|s,a)$, where H is the entropy function and S is a random variable representing the state space. This rewards the agent for taking actions that maximize the reduction in uncertainty about the environment.
- Empowerment: $R_i(s) = \max_\pi I(S';A|s)$, where I is the mutual information between the next state S' and the action A conditioned on the current state s. This rewards the agent for being in states where its actions have the most influence on the future states.

The self-reflective MCTS algorithm can be described as follows:



1. Selection: Use a modified UCT policy that incorporates both the extrinsic and intrinsic Q-values to select actions until a leaf node is reached:

   $\pi(a|s) = \arg\max_a [Q(s,a) + c \sqrt{(\log N(s))}/N(s,a)]$

   where $N(s)$ is the number of times the state s has been visited, $N(s,a)$ is the number of times the action a has been taken in state s, and c is a constant that controls the exploration-exploitation trade-off.

2. Expansion: If the selected leaf node is not a terminal state, use the self-reflective meta-DSL to generate and evaluate candidate actions for expansion. The meta-DSL can use constructs such as DEFINE, REFINE, and DECIDE to create new objects, modify existing objects, or select among available actions based on their expected intrinsic and extrinsic rewards.

3. Simulation: From the expanded node(s), perform Monte Carlo simulations using a model-based approach, where the transition probabilities $T(s'|s,a)$ and the rewards $R(s,a)$ and $R_i(s,a)$ are estimated from a learned model of the environment dynamics. The simulations are run until a terminal state is reached or a maximum depth is exceeded.

4. Backpropagation: Propagate the simulation results back through the tree, updating the estimated Q-values and visit counts of each node along the path using a weighted combination of the extrinsic and intrinsic rewards:

   $Q(s,a) \leftarrow Q(s,a) + \alpha\,[R(s,a) + R_i(s,a) + \gamma$ max_{a'} $Q(s',a') - Q(s,a)]$

   where α is the learning rate, s' is the next state sampled from the model, and the max term represents the estimated optimal value of the next state.

   The self-reflective MCTS algorithm allows the agent to introspect on its own learning process and actively seek out novel and informative experiences that maximize its intrinsic rewards and learning progress. By combining the Q* optimal policy with self-reflective reasoning and intrinsically motivated learning, the agent can effectively balance exploration and exploitation, discover new objects and interactions, and adapt its behavior and representations to the changing needs and challenges of the environment.

🔗 7.7. Monte Carlo Graph Search (MCGS) with Contrastive Learning and Graph Attention

To further enhance the efficiency and generalization capabilities of the MCTS algorithm in the OORL framework, we propose to integrate it with Monte Carlo Graph Search (MCGS), contrastive learning, and graph attention mechanisms. The key idea is to leverage the structure and semantics of the object-dyad graph $\mathcal{G}$ to guide the search process, learn informative and discriminative state embeddings, and focus the exploration on the most promising and relevant regions of the state space.

Let $z(s)$ be the embedding of the current state s, which is a function of the object-dyad graph $\mathcal{G}$. We can learn the embedding function using a contrastive loss, such as the InfoNCE loss:

$\mathcal{L}(z) = -\log\,[\exp(z(s)\cdot z(s_p))\,/\,(\exp(z(s)\cdot z(s_p)) + \sum$_{$s_i \in \mathcal{N}$} $\exp(z(s)\cdot z(s_i)))]$

where s_p is a positive sample that shares the same semantic context as s (e.g., a subgraph or a temporal neighbor), $\mathcal{N}$ is a set of negative samples that have different semantic contexts, and $\cdot$ denotes the dot product.

The contrastive loss encourages the embeddings of semantically similar states to be close to each other, while pushing the embeddings of dissimilar states far apart. This can help capture the high-level structure and properties of the object-dyad graph, and facilitate the definition of more expressive and discriminative reward functions and value functions.

To incorporate the graph embeddings into the MCGS algorithm, we can modify the selection and expansion policies to take into account the similarity and dissimilarity between states. For example, we can define a contrastive selection policy as:

$\pi(a|s) =$ arg max_a $[Q(s,a) + c_1\, z(s)\cdot z(s_p) - c_2 \sum$_{$s_i \in \mathcal{N}$} $z(s)\cdot z(s_i)]$

where c_1 and c_2 are constants that control the importance of the contrastive terms, and s_p and $\mathcal{N}$ are the positive and negative samples, respectively.

Similarly, we can define a contrastive expansion policy that favors the actions that lead to states with high semantic similarity to the current state:

Expand$(s) = \{a \in A(s) \mid z(s)\cdot z(s') \geq \tau\}$

where $A(s)$ is the set of available actions in state s, s' is the next state reached by taking action a, and τ is a threshold that controls the pruning of dissimilar states.

The contrastive selection and expansion policies can help guide the MCGS search towards the most promising and semantically relevant regions of the state space, while avoiding the exploration of irrelevant or suboptimal regions.

To further enhance the discriminative power of the MCGS algorithm, we can use graph attention mechanisms, such as Graph Attention Networks (GATs), to learn state embeddings that adaptively focus on the most informative and discriminative aspects of the object-dyad graph. The GAT layer can be defined as:

$z(s) = \sum\_\{s_i \in \mathcal{N}(s)\} \alpha(s,s_i)\ h(s_i)$

where $\mathcal{N}(s)$ is the set of neighboring states of s in the object-dyad graph, $h(s_i)$ is the feature vector of state s_i, and $\alpha(s,s_i)$ is the attention weight that measures the importance of the neighbor s_i for the current state s. The attention weights are computed using a softmax function over the dot product of the query and key vectors:

$\alpha(s,s_i) = \exp(q(s) \cdot k(s_i)) / \sum\_\{s_j \in \mathcal{N}(s)\} \exp(q(s) \cdot k(s_j))$

where $q(s)$ and $k(s_i)$ are learnable query and key functions that map the state features to a common attention space.

The GAT-based state embeddings can capture the most relevant and discriminative aspects of the object-dyad graph, and provide a more informative and compact representation of the state space for the MCGS algorithm. The attention mechanism can also help filter out the noise and irrelevant information in the graph, and focus the search on the most promising and semantically meaningful regions.

By integrating contrastive learning, graph attention, and intrinsically motivated learning into the MCGS algorithm, we can create a powerful and flexible framework for open-ended learning and decision-making in the OORL setting. The enhanced MCGS algorithm can effectively discover novel objects and interactions, optimize policies for multiple objectives, generalize to new and unseen tasks, and adapt its representations and strategies to the changing needs and challenges of the environment.

The key advantages of the proposed approach include:

1. Sample efficiency: By leveraging the structure and semantics of the object-dyad graph, the MCGS algorithm can quickly identify the most relevant and promising regions of the state space, and avoid wasting time on irrelevant or suboptimal explorations. The model-based simulations and the intrinsic rewards can further guide the search towards the most informative and learnable experiences, reducing the number of samples needed to find good policies.

2. Generalization: The contrastive learning and graph attention mechanisms can help learn state embeddings that capture the high-level features and relationships of the object-dyad graph, and provide a more transferable and generalizable representation of the state space. This can enable the agent to quickly adapt its knowledge and skills to new and unseen tasks, by leveraging the similarities and analogies between the learned embeddings and the novel situations.

3. Multi-objective optimization: The self-reflective meta-DSL and the intrinsically motivated learning can help balance multiple objectives and motivations, such as maximizing extrinsic rewards, minimizing prediction errors, and maximizing information gain or empowerment. By adaptively weighting and combining these objectives based on the agent's learning progress and the task demands, the MCGS algorithm can find policies that optimize for both short-term and long-term goals, and strike a balance between exploitation and exploration.

4. Open-endedness: The object-rewriting capabilities of the OORL framework, combined with the exploratory and introspective nature of the MCGS algorithm, can enable the agent to continuously discover and create new objects, interactions, and representations, and expand its knowledge and skills in an open-ended manner. The self-reflective meta-DSL can guide the agent towards the most promising and innovative reconfigurations of the object-dyad graph, while the contrastive and attentive mechanisms can help assess the novelty and relevance of the generated objects and interactions.

In summary, the integration of MCTS, MCGS, Q* optimal policy, self-reflective reasoning, contrastive learning, graph attention, and intrinsically motivated learning provides a comprehensive and principled framework for open-ended learning and decision-making in the OORL setting. The proposed approach can effectively leverage the structure and semantics of the object-dyad graph, balance multiple objectives and motivations, generalize to new and unseen tasks, and discover novel and innovative solutions in a sample-efficient and open-ended manner. The MCGS algorithm, enhanced with these techniques, represents a significant step towards the development of truly autonomous and adaptive agents that can learn and evolve in complex and dynamic environments.

8. OPEN-ENDED LEARNING VIA OBJECT REWRITING

✎ 8.1. Formalizing Self-Expanding Action Spaces

8.1.1. Action Space Representation

Let's enhance the representation of the action space. We'll consider the action space not only as a set, but as a dynamic, weighted multiset:

$\mathcal{A}_t(o) = \{(a, w(a, t)) \mid a \in \mathcal{A}_0(o) \cup \mathcal{A}_e(o, t)\}$

where:

- $\mathcal{A}_0(o)$ represents the set of initial actions.
- $\mathcal{A}_e(o, t)$ represents the expanded actions at time t.
- $w(a, t) \in \mathbb{R}^+$ is the weight associated with action a at time t, reflecting its estimated utility or frequency of use. Weights can be initialized uniformly and updated based on learning and feedback.

8.1.2 Meta-DSL Action Generation

We can formalize the `d.EXPAND` construct more precisely:

- **Candidate Action Space:** Let $\mathcal{A}'$ be the space of all syntactically valid candidate actions constructible from the meta-DSL primitives. This can be defined recursively based on the grammar of the meta-DSL.

- **Generation Probability:** The `d.EXPAND` construct defines a probability distribution $P(\mathcal{A}'|s, h, g, w)$ over $\mathcal{A}'$ given the object's current context (state, history, goal, world model).

- **Sampling:** A new action a' is sampled from this distribution:

  $a' \sim P(\mathcal{A}'|s, h, g, w)$

  This sampling process can be realized using various methods:

  - **Enumeration and weighting:** Assigning probabilities to each candidate action in $\mathcal{A}'$ and sampling based on these probabilities.
  - **Generative models:** Training a model (e.g., neural network, probabilistic program) to generate actions directly from the input context.

8.1.3 Action Evaluation and Incorporation

The evaluation function E remains crucial:

- **Evaluation Function:** $E: \mathcal{A}' \times \mathcal{S} \times \mathcal{H} \times \mathcal{G} \times \mathcal{W} \to [0, 1]$

- **Weight Update:** Upon incorporating a new action, we also need to initialize its weight:

  $w(a', t + 1) \leftarrow w_0$ if $E(a', s, h, g, w) \geq \tau$

  where w_0 is an initial weight value.

- **Weight Adaptation:** We need a weight adaptation function:

  $W: \mathbb{R}^+ \times \mathcal{R} \to \mathbb{R}^+$

  that updates the weight of an action based on the received reward $r \in \mathcal{R}$.

  For example, a simple update rule could be:

  $W(w(a, t), r) = w(a, t) + \alpha \cdot r$

where α is a learning rate.

🙦 8.2 User Intent Parsing and Graph Exploration

8.2.1 Intent Decomposition

Let's represent the intent decomposition function with more detail:

$D(u) = \{\text{argmax\_}\{i \in I\}\ s(i, u)\}$

where:

- $s(i, u)$ is a scoring function that measures the compatibility between an intent $i \in I$ and the user prompt u.
- argmax returns the intent with the highest score.

This scoring function can be based on various techniques:

- **Semantic similarity:** Calculating the cosine similarity between vector representations of the intent and the prompt using techniques like BERT or Sentence-BERT.
- **Conditional probability:** Estimating the probability of intent i given prompt u using a trained classifier.

8.2.2 Graph Traversal and Object Activation

- **Activation Probability:** Instead of a hard threshold, we introduce a probability of activation for each object:

  $P(o \mid I_u) = \text{sigmoid}(C(o))$

  where sigmoid is the sigmoid function.

- **Sampling Active Objects:** We can then sample a set of active objects O' based on this probability distribution. This allows for stochasticity in the activation process.

🙦 8.3. Agent Output and Reasoning

8.3.1. Agent Output Structure

Let's define the structure of T, the agent's processed internal thinking, with more detail:

$T = (T_1, T_2, \ldots, T_m)$

where:

- Each $T_i \in \mathbb{R}^d$ represents a vector embedding of a thought or concept.
- m is the number of thoughts generated by the agent.

The thinking process T can be similarly represented as a sequence of thought embeddings.

8.3.2 Reasoning Over Object Outputs

- **Attention over Objects:** Introduce an attention mechanism to weight the object outputs based on their relevance:

  $\alpha(o_i) = \text{softmax}(\text{f\_att}(T, a_i))$

  where:

  - $\alpha(o_i)$ is the attention weight for the output of object o_i.
  - f_att is an attention function that computes a score based on the agent's thinking T and the object output a_i.
  - softmax normalizes the scores into a probability distribution.

- **Thought Update:** The agent's thinking is then updated by incorporating the weighted object outputs:

  $T' = \text{g\_update}(T, \sum\text{\_}\{i=1\}\text{\^{}}\{k\}\ \alpha(o_i) \cdot a_i)$

where g_update is a function that integrates the weighted object outputs with the agent's prior thinking.

- **Meta-DSL Modification:** If self-modification is enabled ($\Sigma = 1$), the meta-DSL blocks M are updated using a modification function M:

  $M' = M(M, T', \text{A}, \mathcal{R})$

  where $\mathcal{R}$ is the set of rewards received by the agent.

- **Result Generation:** The final result R is generated based on the updated thinking T' and the modified meta-DSL blocks M':

  $\text{R} = \text{G}(T', M', \text{G\_A})$

  where G_A is the agent's goal representation and G is a result generation function.

8.4 Meta-DSL Execution

To formally define the execution of the meta-DSL, we introduce an interpreter function:

$\text{I}: \mathcal{O} \times M \to \mathcal{O}'$

where:

- $\mathcal{O}$ is the space of possible object configurations.
- M is the set of meta-DSL blocks.
- $\mathcal{O}'$ is the space of modified object configurations.

The interpreter takes an object o and a set of meta-DSL blocks M as input, and applies the constructs defined in M to modify the object, producing a new object configuration o'. The interpreter's behavior can be formally specified using operational semantics rules that define the effect of each meta-DSL construct on the object's attributes and methods.

8.5. Reward & Policy Optimization

The reward learning and policy optimization processes are intertwined with schema evolution and meta-DSL execution:

- **Reward Attribution:** Reward signals received by the agent need to be attributed to specific objects and actions within the object graph, taking into account the causal dependencies between object interactions and the modifications induced by the meta-DSL. This can be achieved using techniques from causal inference or credit assignment.

- **Policy Gradient with Schema Changes:** The policy gradient needs to account for the dynamic nature of the object graph and the possibility of self-modification. This may involve techniques for differentiating through the schema evolution process or using meta-learning algorithms that learn to adapt the policy based on the changing schema.

8.6. Open-Endedness and Exploration

The open-ended learning aspect of OORL arises from the interplay of several factors:

- **Schema Evolution:** The object graph can grow and change over time, creating new possibilities for interactions and behaviors.
- **Self-Expanding Action Spaces:** Objects can dynamically expand their action repertoires, discovering new ways to interact with the environment and each other.
- **Meta-DSL Self-Modification:** Objects can adjust their own learning algorithms, goals, and strategies, leading to further diversification and adaptation.

To promote exploration in this vast and evolving space, we can employ mechanisms like:

- **Intrinsic Rewards:** Encouraging objects to explore novel states, actions, and configurations through intrinsic motivation.
- **Diversity-Promoting Objectives:** Rewarding the system for generating and maintaining a diverse set of objects, interactions, and behaviors.

- **Curriculum Learning:** Gradually increasing the complexity and difficulty of the tasks and environments, guiding the system towards more sophisticated capabilities.

By incorporating these elements, the OORL framework can facilitate open-ended learning and the emergence of increasingly complex and adaptive behaviors in a continuous and self-directed manner.

🔑 8.7. Practical Considerations & Scalability

Building a large-scale OORL system requires addressing several practical challenges:

- **Representational Complexity:** Managing the complexity of the object graph, meta-DSL programs, and learning models can be computationally demanding. Techniques like knowledge distillation, compression, and modularization can be employed to reduce complexity and improve efficiency.
- **Distributed Processing and Communication:** Distributing the computations and data across multiple processors or machines is essential for scalability. Technologies like message queues, distributed databases, and parallel computing frameworks can be leveraged to achieve this.
- **Real-Time Performance:** Many applications require real-time decision making and adaptation. Optimizing the performance of object interactions, meta-DSL execution, and learning algorithms is crucial for meeting these requirements.

You're right, we need to flesh out these additional sections to provide a comprehensive picture of the OORL framework. Here they are with an emphasis on mathematical formalisms.

9. HIERARCHICAL OBJECT COMPOSITION

🔑 9.1. Object Aggregation and Decomposition

In large-scale OORL systems, managing the complexity of the object graph $\mathcal{G}$ is crucial. Hierarchical object composition provides a way to abstract and simplify the representation by grouping objects and their interactions.

9.1.1 Aggregation

Let $O' \subseteq \mathcal{O}$ be a subset of objects in the graph. We define an aggregation function:

Agg: $\mathcal{P}(\mathcal{O}) \to \mathcal{O}$

where $\mathcal{P}(\mathcal{O})$ is the power set of $\mathcal{O}$. Agg(O') creates a new aggregate object $o_a \in \mathcal{O}$ that represents the collective properties and behaviors of the objects in O'.

The internal state s_a, methods m_a, and goal g_a of o_a are derived from the corresponding attributes of the constituent objects in O', potentially using techniques like:

- **State concatenation or averaging:** s_a = concat($s_1, \ldots, s_n$) or $s_a = 1/n \sum_i s_i$, where $s_1, \ldots, s_n$ are the states of the objects in O'.

- **Method inheritance or composition:** m_a inherits or combines the methods of the objects in O'.

- **Goal aggregation or fusion:** g_a represents a shared or common goal for the objects in O', potentially derived through negotiation or multi-objective optimization.

9.1.2 Decomposition

The decomposition function performs the inverse operation:

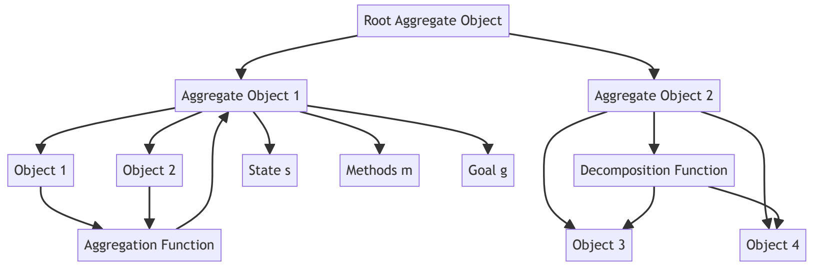
Dec: $\mathcal{O} \to \mathcal{P}(\mathcal{O})$

Dec(o_a) decomposes an aggregate object o_a into its constituent objects, O'. This can be based on predefined rules, learned hierarchies, or dynamic criteria like task requirements or resource constraints.

9.1.3. *Hierarchical Structure*

The aggregation and decomposition functions induce a hierarchical structure on the object graph $\mathcal{G}$. This structure can be represented as a tree $\mathcal{T}$, where:

- The nodes of $\mathcal{T}$ represent objects (both individual and aggregate).
- The edges of $\mathcal{T}$ represent the aggregation or decomposition relationships between objects.
- The root of $\mathcal{T}$ represents the most abstract or top-level aggregate object.
- The leaves of $\mathcal{T}$ represent the individual objects.



⚙ 9.2. Hierarchical Planning

Hierarchical object composition enables hierarchical planning, allowing for more efficient and scalable decision-making in large-scale systems.

9.2.1. *Hierarchical Policy*

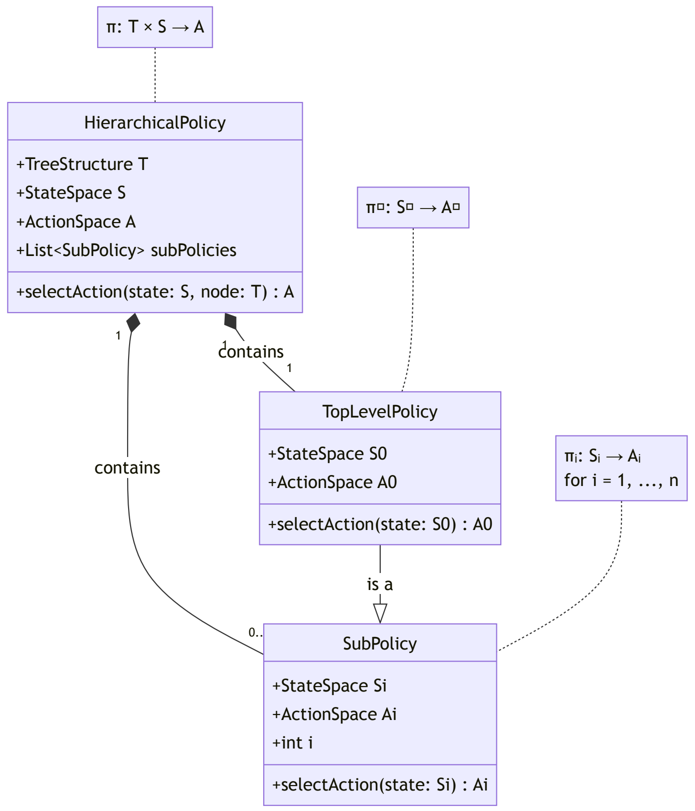
We can define a hierarchical policy $\pi: \mathcal{T} \times \mathcal{S} \to \mathcal{A}$ that operates over the tree structure $\mathcal{T}$ and the state space $\mathcal{S}$. This policy consists of a set of sub-policies:

$\pi = \{\pi_0, \pi_1, ..., \pi_n\}$

where:

- π_0: $\mathcal{S}_0 \to \mathcal{A}_0$ is the top-level policy, mapping the state of the root object $s_0 \in \mathcal{S}_0$ to an action $a_0 \in \mathcal{A}_0$. This action typically corresponds to selecting a subgoal or a high-level plan.

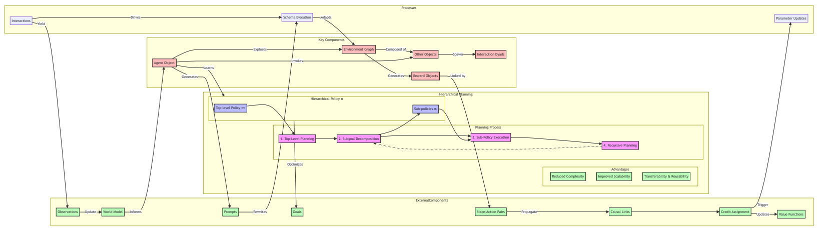
- π_i: $\mathcal{S}_i \to \mathcal{A}_i$ (for $i = 1, ..., n$) are the sub-policies for the child nodes of the tree. Each π_i maps the state of a child object $s_i \in \mathcal{S}_i$ to an action $a_i \in \mathcal{A}_i$, based on the chosen subgoal or plan from the parent node.



9.2.2. Hierarchical Planning Process

The hierarchical planning process proceeds recursively:

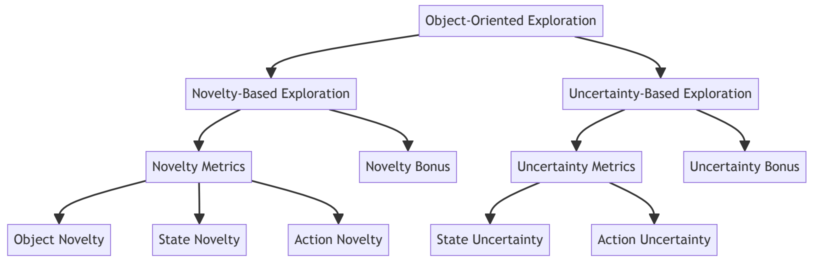
1. **Top-Level Planning:** The agent uses the top-level policy π_0 to select a subgoal or plan based on the current state of the root object s_0.
2. **Subgoal Decomposition:** The chosen subgoal or plan is decomposed into a set of sub-goals or sub-plans for the child nodes of the tree, based on the decomposition function Dec.
3. **Sub-Policy Execution:** Each child node uses its corresponding sub-policy π_i to select an action a_i based on its current state s_i and the assigned sub-goal or sub-plan.
4. **Recursive Planning:** Steps 2 and 3 are repeated recursively for each child node, until the leaf nodes (individual objects) are reached and execute their actions in the environment.



9.2.3. Advantages of Hierarchical Planning

- **Reduced Complexity:** Hierarchical planning breaks down a complex problem into smaller, more manageable sub-problems, reducing the overall search space and computational complexity.
- **Improved Scalability:** The hierarchical structure allows for distributed and parallel processing of the sub-problems, enabling the system to scale to larger and more complex environments.
- **Transferability and Reusability:** Sub-policies and sub-plans can be reused or adapted to different contexts, facilitating transfer learning and knowledge sharing among the objects.

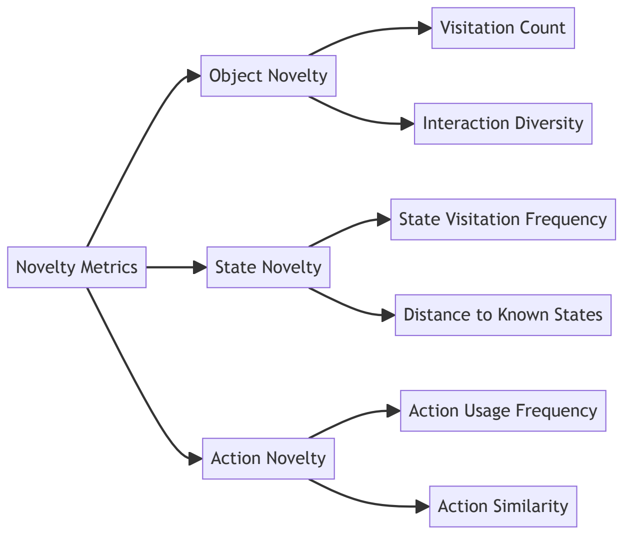
10. OBJECT-ORIENTED EXPLORATION



10.1. Novelty-Based Exploration

In OORL, exploration is crucial for discovering novel and potentially beneficial object interactions and schema configurations. Novelty-based exploration incentivizes the system to explore states, actions, or objects that haven't been frequently encountered.

10.1.1. Novelty Metrics



- **Object Novelty:**

  - Visitation Count: N(o, t) is the number of times object o has been activated up to time t. A lower count suggests higher novelty.
  - Interaction Diversity: D(o, t) measures the diversity of interactions object o has participated in, considering the types of prompts and responses exchanged.

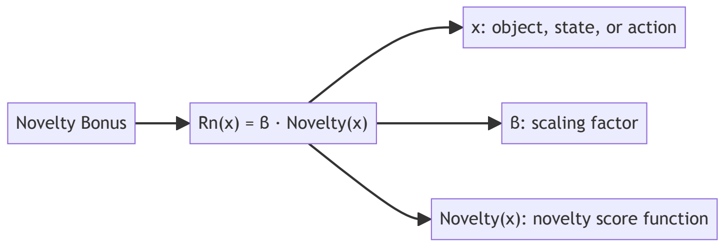
- **State Novelty:**

- State Visitation Frequency: $N(s, t)$ counts the visits to state s.
- Distance to Known States: $\text{Dist}(s, \mathcal{S}_k)$ measures the distance of state s from a set of known states $\mathcal{S}_k$. This could be based on Euclidean distance in a feature space or graph distance in the object-dyad graph.

- **Action Novelty:**

- Action Usage Frequency: $N(a, t)$ tracks how often action a has been executed.
- Action Similarity: $\text{Sim}(a, \mathcal{A}_0)$ measures the similarity of a new action a to the set of initial actions $\mathcal{A}_0$, highlighting novelty when similarity is low.

10.1.2. Novelty Bonus


Novelty Bonus
Rn(x) = β · Novelty(x)
x: object, state, or action
β: scaling factor
Novelty(x): novelty score function

A novelty bonus $R_n(x)$ can be added to the extrinsic reward for exploring novel elements:

$R_n(x) = \beta \cdot \text{Novelty}(x)$

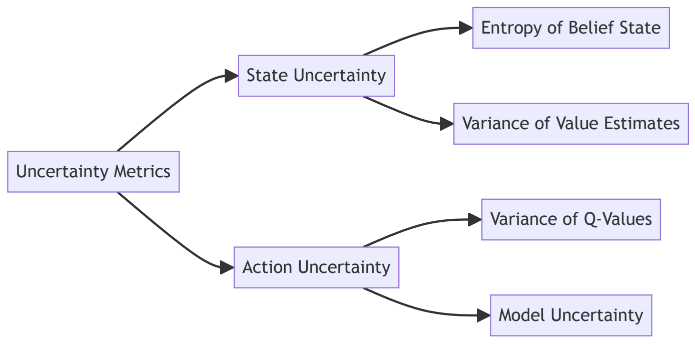
where:

- x can be an object, state, or action.
- β is a scaling factor controlling the importance of novelty.
- $\text{Novelty}(x)$ is a function that computes the novelty score of x, using one or a combination of the metrics mentioned above.

⚲ 10.2. Uncertainty-Based Exploration

Uncertainty-based exploration focuses on exploring areas of the state or action space where the agent has high uncertainty about the consequences of its actions.

10.2.1. Uncertainty Metrics

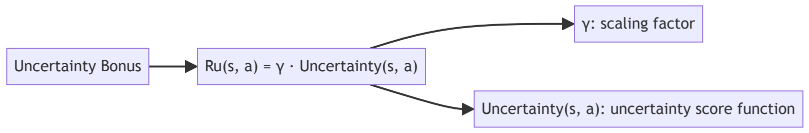


- **State Uncertainty:**

- Entropy of Belief State: H(b(s)) quantifies the uncertainty of the agent's belief state b(s) over possible world states.
- Variance of Value Estimates: Var(V(s)) measures the variance in the agent's estimate of the value function for state s, indicating uncertainty about the long-term reward potential.

- **Action Uncertainty:**
  - Variance of Q-Values: Var(Q(s, a)) captures the variance in the agent's estimate of the Q-value for action a in state s, reflecting uncertainty about the immediate reward.
  - Model Uncertainty: U(s, a) represents the uncertainty in the agent's world model w about the state transition probabilities for action a in state s.

10.2.2. Uncertainty Bonus


Uncertainty Bonus
Ru(s, a) = γ · Uncertainty(s, a)
γ: scaling factor
Uncertainty(s, a): uncertainty score function

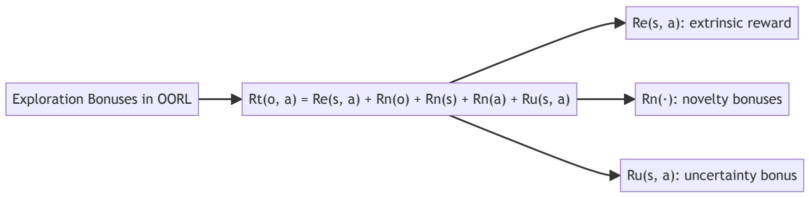
An uncertainty bonus $R_u(s, a)$ can be incorporated into the reward function:

$R_u(s, a) = \gamma \cdot \text{Uncertainty}(s, a)$

where:

- γ is a scaling factor.
- Uncertainty(s, a) is a function that computes the uncertainty score, using the metrics mentioned above.

⚙ 10.3. Exploration Bonuses in OORL


Exploration Bonuses in OORL
Rt(o, a) = Re(s, a) + Rn(o) + Rn(s) + Rn(a) + Ru(s, a)
Re(s, a): extrinsic reward
Rn(·): novelty bonuses
Ru(s, a): uncertainty bonus

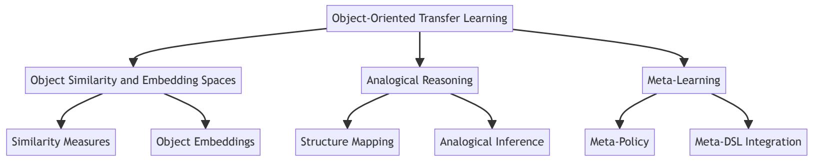
Exploration bonuses, combining novelty and uncertainty, can be integrated into the reward function to guide object interactions and schema evolution:

$R_t(o, a) = R_e(s, a) + R_n(o) + R_n(s) + R_n(a) + R_u(s, a)$

where:

- $R_e(s, a)$ is the extrinsic reward for taking action a in state s.
- $R_n(\cdot)$ are the novelty bonuses for the object, state, and action.
- $R_u(s, a)$ is the uncertainty bonus.

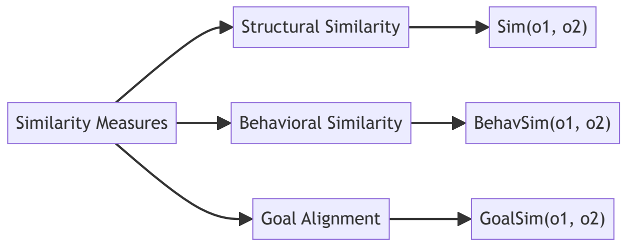
11. OBJECT-ORIENTED TRANSFER LEARNING



⚙ 11.1 Object Similarity and Embedding Spaces

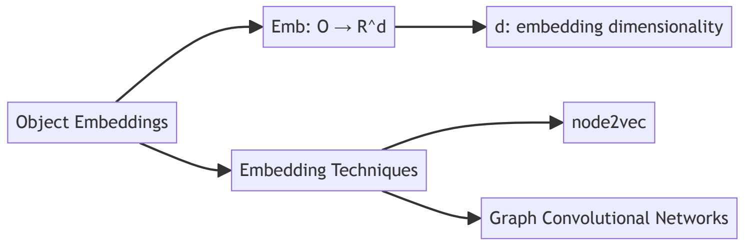
Transfer learning in OORL leverages past knowledge and skills to accelerate learning in new tasks. Object similarity plays a crucial role in identifying transferable knowledge.

11.1.1. Similarity Measures



- **Structural Similarity:** $\mathrm{Sim}(o_1, o_2)$ measures the structural similarity of two objects based on their attributes, methods, and relations in the object graph. Graph-based kernel functions can be used to compute this similarity.
- **Behavioral Similarity:** $\mathrm{BehavSim}(o_1, o_2)$ quantifies the similarity of their behaviors, based on their interaction histories h and the prompts/responses exchanged. Sequence alignment algorithms or recurrent neural networks can be used to compare interaction sequences.
- **Goal Alignment:** $\mathrm{GoalSim}(o_1, o_2)$ measures the alignment of their goals g, potentially using metrics based on the distance or overlap between goal representations.

11.1.2. Object Embeddings



To efficiently compare objects, we use embedding functions to map them into a continuous vector space:

$\mathrm{Emb}: \mathcal{O} \to \mathbb{R}^{\mathrm{d}}$

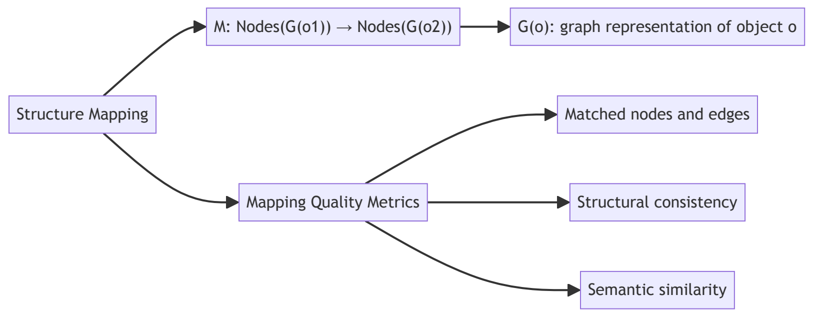
where d is the embedding dimensionality. Techniques like node2vec or graph convolutional networks can be used to learn these embeddings.

The similarity between objects can then be computed as the distance or similarity between their embeddings.

🝰 11.2. Analogical Reasoning

Analogical reasoning extends object similarity to more abstract relationships, allowing objects to draw inferences and make generalizations based on structural or functional correspondences.

11.2.1. Structure Mapping

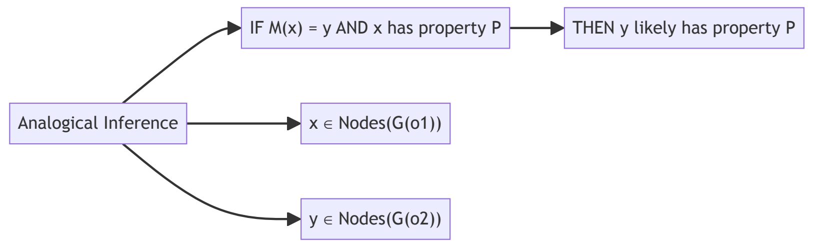


Let $G(o)$ be a graph representation of object o, capturing its attributes, methods, and relations. A structure mapping between two objects o_1 and o_2 is a function:

M: Nodes($G(o_1)$) → Nodes($G(o_2)$)

that maps nodes in $G(o_1)$ to nodes in $G(o_2)$ while preserving structural relationships. The quality of the mapping can be assessed using various metrics that consider the number of matched nodes and edges, the structural consistency, and the semantic similarity of the mapped elements.

11.2.2. Analogical Inference


Analogical Inference
IF M(x) = y AND x has property P
THEN y likely has property P
x ∊ Nodes(G(o1))
y ∊ Nodes(G(o2))

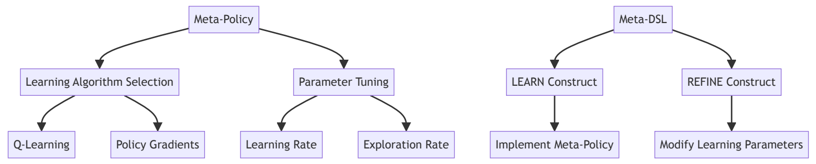
Given a good structure mapping, the agent can infer new properties or behaviors for o_2 by transferring the corresponding properties or behaviors from o_1. This can be formalized as a rule:

IF M(x) = y AND x has property P THEN y likely has property P

where $x \in$ Nodes($G(o_1)$) and $y \in$ Nodes($G(o_2)$).

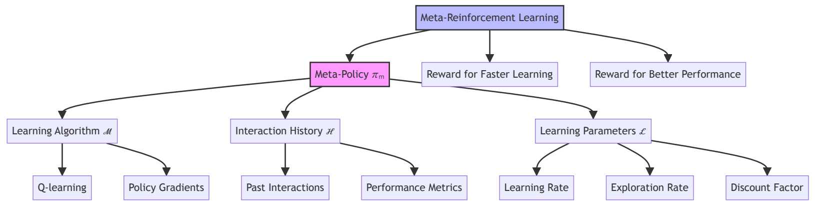
☙ 11.3. Meta-Learning

Meta-learning enables objects to "learn to learn" by acquiring and adapting learning strategies or algorithms based on experience.



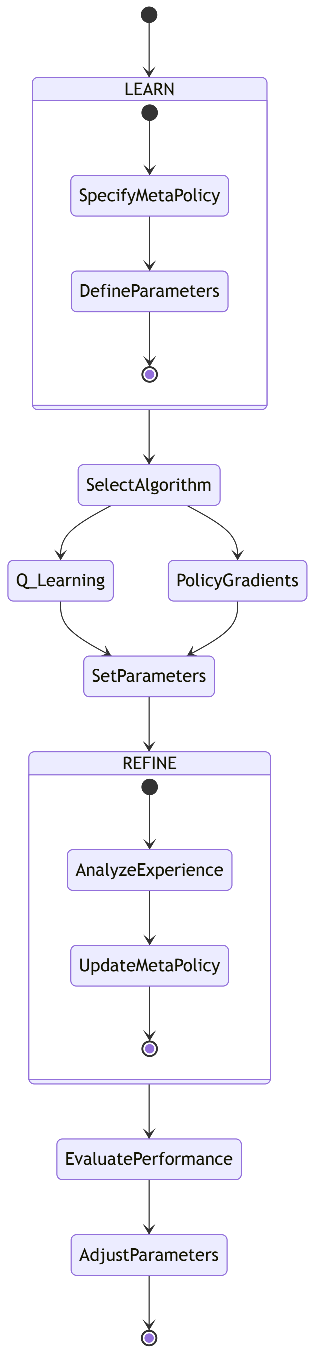
11.3.1. Meta-Policy

A meta-policy $\pi_m: \mathcal{M} \times \mathcal{H} \to \mathcal{L}$ maps a learning algorithm $\mathcal{M}$ (e.g., Q-learning, policy gradients) and the object's interaction history $\mathcal{H}$ to a set of learning parameters $\mathcal{L}$. The meta-policy can be learned using meta-reinforcement learning techniques, where the agent is rewarded for choosing learning algorithms and parameters that lead to faster learning and better performance on a variety of tasks.


Meta-Reinforcement Learning
Meta-Policy π_m
Reward for Faster Learning
Reward for Better Performance
Learning Algorithm $\mathscr{M}$
Interaction History $\mathscr{H}$
Learning Parameters $\mathscr{L}$
Q-learning
Policy Gradients
Past Interactions
Performance Metrics
Learning Rate
Exploration Rate
Discount Factor

11.3.2. Meta-DSL Integration

The meta-DSL can be used to implement meta-learning by enabling objects to modify their `LEARN` and `REFINE` constructs. The `LEARN` construct can specify the meta-policy and the parameters it uses to select learning algorithms. The `REFINE` construct can then modify those learning parameters based on the object's experience and feedback, effectively enabling objects to adapt their own learning strategies over time.



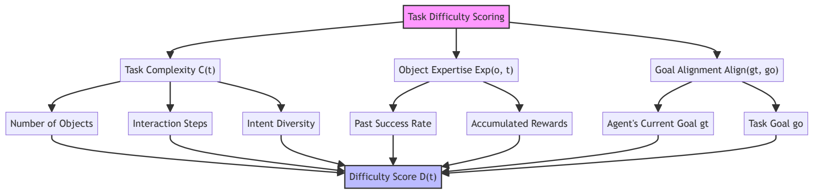
12. OBJECT-ORIENTED CURRICULUM LEARNING

🜲 12.1 Task Difficulty Scoring

Curriculum learning involves presenting learning tasks to the agent in a structured order of increasing difficulty, which can significantly improve learning efficiency and performance. In OORL, we need a way to assess the difficulty of tasks in the context of the object graph and the agent's capabilities.

12.1.1 Difficulty Metrics

- Task Complexity: $C(t)$ measures the complexity of a task t, based on factors such as the number of objects involved, the number of interaction steps required, or the diversity of intents needed to be fulfilled.
- Object Expertise: $\mathrm{Exp}(o, t)$ quantifies the agent's expertise or proficiency in interacting with object o, potentially based on past success rate or accumulated rewards for interactions with o.
- Goal Alignment: $\mathrm{Align}(g_t, g_o)$ measures the alignment between the agent's current goal g_t and the goal associated with the task t, denoted by g_o.



12.1.2 Difficulty Score

The overall difficulty score $D(t)$ for task t can be a function combining these metrics:

$$D(t) = \text{f\_diff}(C(t), \{\mathrm{Exp}(o, t) \mid o \in \mathcal{O}_t\}, \mathrm{Align}(g_t, g_o))$$

where $\mathcal{O}_t$ is the set of objects involved in task t and f_diff is a function that combines the individual difficulty metrics into a single score.

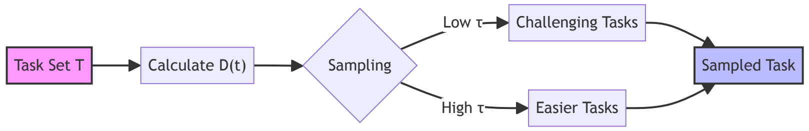
12.2 Task Sampling and Curriculum Generation

12.2.1 Task Sampling

Given a set of tasks T, the probability of sampling task t for training can be based on its difficulty score:

$$P(t) \propto \exp(-D(t) / \tau)$$

where τ is a temperature parameter controlling the balance between challenging tasks (low temperature) and easier tasks (high temperature).



12.2.2 Curriculum Generation

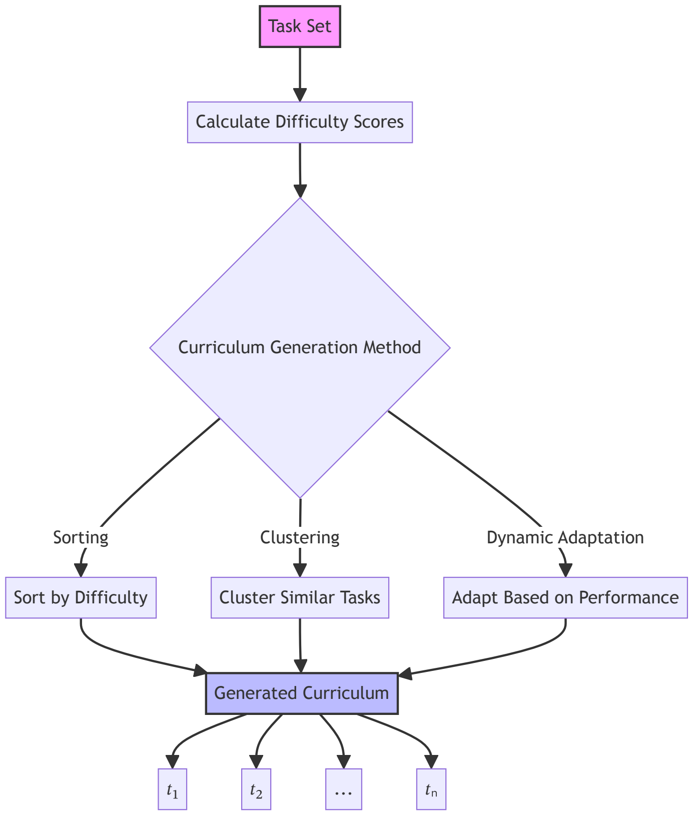
A curriculum $\mathcal{C}$ is a sequence of tasks ordered by increasing difficulty:

$$\mathcal{C} = (t_1, t_2, \ldots, t_n) \text{ where } D(t_1) \leq D(t_2) \leq \ldots \leq D(t_n)$$

The curriculum can be generated by:

- Sorting the tasks based on their difficulty scores.
- Using a clustering algorithm to group tasks with similar difficulty levels and then ordering the clusters.

- Dynamically adapting the curriculum based on the agent's learning progress and performance on previous tasks.



13. OBJECT-ORIENTED INTRINSIC MOTIVATION

13.1 Empowerment

Empowerment is a measure of an agent's influence or control over its environment. In OORL, we can define object empowerment as the ability of an object to influence the future states of other objects through its interactions.

13.1.1 Empowerment Metric

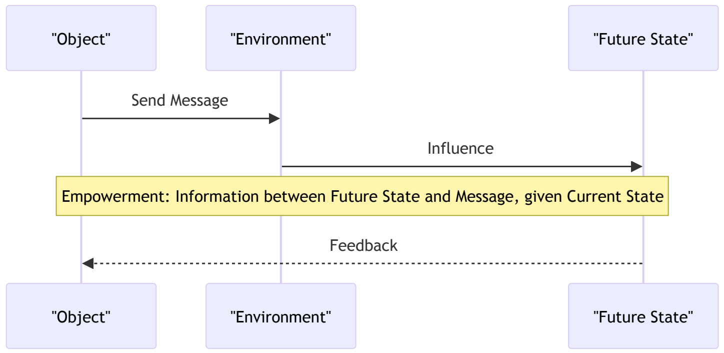
Empowerment of object o at time t can be quantified as:

$\text{Empower}(o, t) = \mathbb{E}_{\pi} [I(S_{t+k}; M(o, t) \mid S_t)]$

where:

- $\mathbb{E}_{\pi}$ denotes the expectation over the agent's policy π.
- $I(S_{t+k}; M(o, t) \mid S_t)$ is the mutual information between the future state S_{t+k} (after k time steps) and the message $M(o, t)$ sent by object o at time t, given the current state S_t.
- k is a parameter controlling the time horizon of empowerment.

Intuitively, object empowerment measures how much information the object's message conveys about the future state of the system.

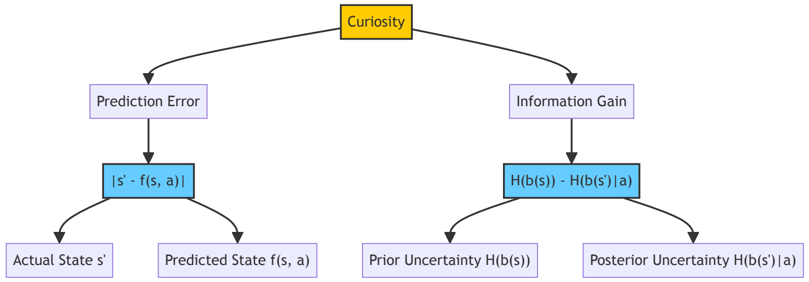


13.2 Curiosity

Curiosity drives objects to explore and learn about unknown or uncertain aspects of the environment.

13.2.1 Curiosity Metrics

- Prediction Error: $|s' - f(s, a)|$, where s' is the actual next state, and $f(s, a)$ is the predicted next state based on the object's world model w.
- Information Gain: H(b(s)) - H(b(s')|a), representing the reduction in uncertainty about the world state after taking action a.



13.2.2 Curiosity Bonus

A curiosity bonus R_c can be added to the reward function to encourage exploratory behavior:

$R_c(s, a, s') = \eta \cdot \text{Curiosity}(s, a, s')$

where η is a scaling factor.

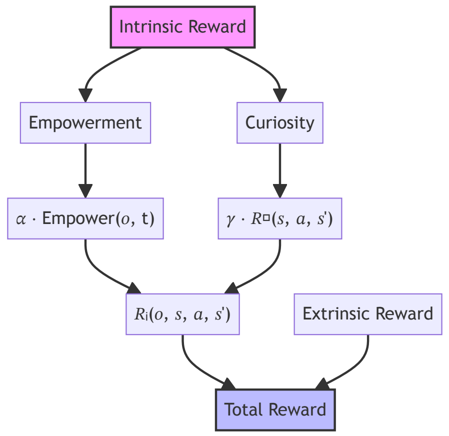
13.3. Intrinsic Rewards in OORL

Empowerment and curiosity can be combined into an intrinsic reward signal:

$R_i(o, s, a, s') = \alpha \cdot \text{Empower}(o, \mathrm{t}) + \gamma \cdot R_c(s, a, s')$

where α and γ are weighting factors balancing the importance of empowerment and curiosity.

This intrinsic reward can then be integrated with the extrinsic reward to guide object interactions and schema evolution.



14. CHALLENGES AND FUTURE DIRECTIONS

➳ 14.1. Representational and Computational Complexity

- Developing scalable and efficient representations of objects, schemas, and meta-DSL programs for large-scale systems.
- Exploring methods for compressing and summarizing object information to reduce memory and computational overhead.
- Investigating approximate inference and planning techniques for handling complex and uncertain environments.

➳ 14.2. Scalability and Efficiency of Learning Algorithms

- Designing distributed and parallel learning algorithms that can efficiently update the policies, world models, and meta-DSL programs of multiple objects concurrently.
- Developing online or incremental learning methods that can adapt to changing schemas and environments without requiring full retraining.

➳ 14.3. Emergence of Abstract and Transferable Concepts

- Understanding the mechanisms and conditions that promote the emergence of abstract and reusable concepts from object interactions and schema evolution.
- Developing techniques for measuring and promoting the transferability and composability of learned knowledge and skills across different tasks and domains.

➳ 14.4. Safety, Robustness, and Alignment of Self-Reflective Agents

- Ensuring that the self-modification capabilities of the meta-DSL do not lead to unstable or undesirable behaviors.
- Developing methods for aligning the goals and values of the objects with the overall objectives of the system and human users.
- Addressing the potential risks and ethical implications of open-ended and self-modifying AI systems.

➳ 14.5. Integration with Other Cognitive Faculties

- Integrating the OORL framework with other cognitive faculties, such as attention, memory, reasoning, and communication, to create more comprehensive and versatile intelligent agents.
- Exploring the potential synergies between symbolic AI and deep learning in the context of object-oriented representations and self-reflective learning.

15. CONCLUSION

15.1 Summary of Key Ideas and Contributions

- OORL offers a novel approach for creating adaptive and open-ended learning agents that can operate in complex and dynamic environments.
- The framework combines object-oriented programming, graph theory, reinforcement learning, and meta-learning to enable a more flexible and expressive representation of the environment and the agent's knowledge and strategies.
- The self-reflective meta-DSL allows objects to reason about and modify their own learning process, leading to the emergence of more sophisticated and adaptive behaviors.
- Hierarchical object composition and decomposition enable efficient planning and decision making in large-scale systems.
- Various exploration strategies, including novelty-based and uncertainty-based exploration, promote the discovery of novel and beneficial interactions and configurations.
- Transfer learning mechanisms, such as object similarity, analogical reasoning, and meta-learning, allow the agent to leverage its prior experience and knowledge to learn faster and generalize better to new tasks and domains.

15.2 Potential Impact and Applications

- OORL has the potential to significantly advance the capabilities of AI systems in a wide range of applications, including robotics, natural language processing, and complex decision-making tasks.
- The framework could enable the development of more autonomous and adaptable robots that can learn and interact with their environment in a more flexible and intelligent way.
- OORL could also facilitate the creation of more sophisticated and robust natural language processing systems that can understand and respond to complex and nuanced user requests.

15.3 Outlook and Future Work

- Future work on OORL will focus on addressing the challenges of scalability, stability, and controllability of self-modifying and open-ended learning systems.
- The development of more efficient and expressive meta-DSLs, learning algorithms, and evaluation metrics will be crucial for realizing the full potential of this framework.
- Empirical studies on benchmark tasks and real-world applications will be essential for validating and refining the OORL approach.

15.4. CONCLUSION

The OORL framework offers a promising approach to tackle the challenges of open-ended learning in complex and dynamic environments. By combining the principles of object-oriented programming, graph theory, reinforcement learning, and meta-learning, OORL enables the creation of adaptive and self-modifying agents that can discover and optimize novel behaviors and representations.

While significant challenges remain in terms of scalability, stability, and controllability, the OORL framework provides a solid foundation for future research and development. As we continue to refine and extend this framework, we can expect to see the emergence of more autonomous, intelligent, and adaptable AI systems that can learn and evolve in open-ended and dynamic worlds.