Society of Meta-Agents: The SOUL Motivation Framework for Specific and Objective Understanding Logic

Keyvan M. Sadeghi

April 22, 2025

1 Literature Review

We have identified six practical schools of agent motivation:

- **Intrinsic Motivation in RL**: Curiosity and surprise as auxiliary rewards to drive exploration in sparse environments (Schmidhuber, 2010; Pathak et al., 2017).
- **Information-Theoretic Drives**: Maximizing mutual information, novelty, and empowerment signals (Klyubin et al., 2005; Salge et al., 2014; Goertzel, 2024).
- **Competence-Progress Models**: Rewarding measurable learning progress toward self-generated subgoals to form a self-curriculum (Oudeyer & Kaplan, 2007).
- **Homeostatic/Drive-Reduction**: Minimizing internal variables (e.g., prediction error) via cybernetic drives (Hull, 1943; Friston, 2010).
- **Cognitive Architectures**: Embedding scalar drives into symbolic cycles (Soar's operator preferences; ACT-R's Expected Value of Control; Goertzel, 2024) for precise control updates.
- **Developmental-Robotics Hybrids**: Combining maturational constraints with competence progress to emulate developmental curricula (Lungarella et al., 2003).

2 Mathematical Foundations and Architectural Decisions

2.1 The Motivation Vector

At the heart of the SOUL Motivation Framework is the hidden internal state, the Motivation Vector:

$$\mathbf{s}_t = [s_c, s_u, s_h]$$

where s_c is competence, s_u is novelty/surprise, and s_h is homeostasis. This vector is updated after every interaction and drives all agent actions. In code, it is maintained as a Python dataclass and is never exposed to the LLM.

2.2 Motivation Vector Updates

- **Competence Progress:**

$$\Delta_c = p_g(t) - p_g(t-1), \quad s_c \leftarrow \text{proj}_{[0,1]}(s_c + \alpha \Delta_c)$$

Explanation: $p_g(t)$ is the agent's measured performance (competence) at time t. Δ_c is the change in competence since the last step. s_c is updated by adding the scaled change in competence, then projected to stay within $[0, 1]$.

- **Novelty/Surprise:**

$$\text{novel}(t) = 1 - \frac{\mathbf{e}(t) \cdot \mu_{t-1}}{\|\mathbf{e}(t)\| \, \|\mu_{t-1}\|}, \quad s_u \leftarrow \text{proj}_{[0,1]}(s_u + \alpha \, \text{novel}(t))$$

Explanation: $\mathbf{e}(t)$ is the embedding of the current context; μ_{t-1} is the mean embedding of past contexts. This computes the cosine similarity between current and past contexts, subtracts from 1, so higher values mean more novelty. s_u is updated by adding the scaled novelty score, projected to $[0, 1]$. If either norm is zero, set $\text{novel}(t) = 1$.

- **Homeostatic Decay:**

$$s_h \leftarrow (1 - \delta) s_h + \delta$$

Explanation: s_h is the homeostatic drive, which gently decays toward a baseline (e.g., 1) at rate δ. This ensures the agent doesn't get stuck at extremes.

2.3 Meta-Graph and Rule Engine

The agent maintains a symbolic meta-graph G (a directed graph of rewrite rules R and meta-rules M). Each node represents a pattern or rule, and edges encode transformations or relationships. In Python, this is implemented with `networkx.DiGraph`.

Discrete Generative Core: At each turn, the agent predicts a distribution over rules, compares it to observed outcomes, and computes error using KL divergence:

$$e_t = D_{\text{KL}}(q_t \| p_t)$$

Explanation: p_t is the predicted distribution over rules; q_t is the observed distribution. D_{KL} is the Kullback-Leibler divergence, a measure of how one probability distribution diverges from another. e_t is the error signal used for learning and adaptation.

2.4 Thresholded Nudge and Confidence

At each step, the agent computes a confidence score C_t:

$$C_t = f_{\text{match}}(x_t, G, \mathbf{s}_t)$$

Explanation: C_t is a confidence score computed by comparing the current context x_t and the agent's state $\mathbf{s}_t$ to the meta-graph G using a similarity or density function f_{match} (e.g.,

a softmax-weighted sum over matching patterns). The agent compares C_t to a dynamic threshold τ_t to decide whether to intervene.

$$\text{If} \quad C_t \geq \tau_t \quad \text{then nudge; else remain silent (null action)}$$

Note: All functions such as $\text{proj}_{[0,1]}$, f_{match}, Perceive, UpdateMotivation, and HarvestAxiom are defined in the Appendix.

2.5 Perception–Cognition–Action Loop

The agent's operation at each time t is:

$$\begin{aligned}
\text{Perceive:} \quad & y_t = \text{Perceive}(x_t) \\
\text{Update:} \quad & \mathbf{s}_{t+1} = \text{UpdateMotivation}(\mathbf{s}_t, y_t) \\
\text{Record:} \quad & G_{t+1} = \text{UpdateMetaGraph}(G_t, x_t, y_t) \\
\text{Confidence:} \quad & C_{t+1} = f_{\text{match}}(x_{t+1}, G_{t+1}, \mathbf{s}_{t+1}) \\
\text{Action:} \quad & a_{t+1} = \begin{cases} \text{HarvestAxiom}(G_{t+1}, x_{t+1}) & \text{if } C_{t+1} \geq \tau_{t+1} \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}$$

Explanation: a_{t+1} is either a harvested axiom/rule (to be injected as a nudge) or the null action $\varnothing$ (agent remains silent).

2.6 Subgoal Discovery

If s_c stagnates or s_u spikes, the agent auto-discovers new subgoals by clustering novel contexts in G and generating new rules. This enables adaptive exploration.

2.7 Discrete Generative Core and Error Signals

Instincts and policies are encoded as rewrite rules in G. The agent predicts a distribution $p_t(m)$ over outcomes, observes $q_t(m)$, and computes error:

$$e_t = D_{\text{KL}}(q_t \| p_t) = \sum_m q_t(m) \log \frac{q_t(m)}{p_t(m)}$$

Explanation: p_t is the predicted distribution over rules; q_t is the observed distribution. D_{KL} is the Kullback-Leibler divergence, a measure of how one probability distribution diverges from another. e_t is the error signal used for learning and adaptation. where q_t and p_t are distributions with $\text{supp}(q_t) \subseteq \text{supp}(p_t)$, or $p_t(m)$ is regularized (e.g., $p_t(m) \leftarrow \max(p_t(m), \epsilon)$ for small ϵ).

2.8 Intrinsic and Episodic Rewards

The agent receives two types of reward signals:

- **Intrinsic Reward:**

$$r_t^{\text{int}} = -e_t$$

Explanation: Internal reward is negative error (the agent is rewarded for reducing surprise).

- **Episodic Reward:**

$$r_t^{\text{ep}} = \beta \sum_m q_t(m) \log \frac{1}{p_t(m)}, \quad \beta > 0$$

Explanation: Episodic reward is proportional to the negative log-likelihood of predictions, weighted by β.

β is a proportionality constant.

2.9 Meta-Rule Self-Modification

Meta-rules M can edit or restructure the meta-graph itself. The agent searches for local edits that minimize error:

$$m_i \leftarrow \arg \min_{m' \in \mathcal{N}(m_i)} e_t(R, \{M \setminus m_i\} \cup \{m'\})$$

Explanation: The agent searches for a local change (neighbor m') to a meta-rule m_i that minimizes the error e_t. Meta-rules are rules that can rewrite the rule graph itself. where $\mathcal{N}(m_i)$ denotes the neighborhood of m_i in meta-rule space.

2.10 Wasserstein Natural Gradient

Parameterize rule-distribution $p(\xi)$ and update via:

$$\xi_{k+1} = \xi_k - h\, G(\xi_k)^{-1}\, \nabla_\xi F\big(p(\xi_k)\big)$$

Explanation: ξ are the parameters of the rule distribution. $G(\xi_k)$ is the Laplacian (a kind of matrix) over the rule graph, encoding its geometry. This is a gradient descent step that respects the structure of the rule space (a "natural gradient").

2.11 Neural–Symbolic Hybrid and Memory

Continuous predictive-coding nets (vision and motor) run beneath the discrete core, exchanging features/actions. Long-term memory is implemented via vector stores (e.g., `faiss`, `chromadb`) for retrieval and adaptation.

2.12 LLM Pre-Prompting and Naturalization

When the agent nudges, it injects symbolic axioms/rules into the LLM prompt. The LLM is pre-prompted to interpret these in MeTTa or similar syntax and translate their intent into natural language or actions.

2.13 Genetic Mixing and Policy Sharing

Hyperparameters $(\alpha, \delta, \tau_c, \tau_u)$ are encoded as arrays and can be evolved via genetic algorithms (e.g., `DEAP`, `pygad`), enabling agent societies to mix and share policies and metagraphs.

2.14 Concrete Python Mapping

- **Motivation Vector:** Python `dataclass` with fields for s_c, s_u, s_h.
- **Rule Graph:** `networkx.DiGraph` with nodes for rules/meta-rules.
- **Neural Nets:** `torch.nn.Module` or `jax` models for predictive coding.
- **Memory:** `faiss` or `chromadb` vector store.
- **Hyperparameters:** Numpy array or genetic algorithm chromosome.

3 Null Action and Silent Learning

If the confidence C_t does not exceed the threshold τ_t, the agent performs the null action $\varnothing$, i.e., it remains silent and continues to observe, record, and learn without intervening.

4 Implementation Guide

Below are concrete steps, with Python library suggestions.

4.1 Core Data Structures

- Use `numpy` for vector operations and embeddings.
- Use `networkx` to represent the metagraph of rewrite rules and compute Laplacians.
- Store agent state in a simple `dataclass`:

```
from dataclasses import dataclass
@dataclass
class State:
    competence: float = 0.0
    curiosity:  float = 0.0
    stability:  float = 1.0
```

4.2 Novelty Detector

- Implement rolling mean with `collections.deque` and cosine distance via `scipy.spatial.distance.cos`

4.3 Rewrite-Rule Engine

- Model rules as Python objects mapping pattern graphs to outputs.
- Use `networkx` pattern-matching or custom graph algorithms for rule application.
- Derive $p_t(m)$ by sampling or counting rule firings over stochastic selections (e.g., softmax weights in `torch`).

4.4 Information-Theoretic Error

- Compute KL divergence with `scipy.stats.entropy(q, p)`.

4.5 Reward & State Update Loop

1. Collect feedback score $r(t)$ via environment simulation or user rating.
2. Update `State` (Motivation Vector) using the mathematical formulas above, with learning rate `alpha`.
3. Compute confidence C_t and compare to threshold τ_t; if $C_t \geq \tau_t$, harvest and inject a nudge (axiom/rule) from the meta-graph.
4. If nudging, prepend the harvested axiom/rule to the user query and invoke the LLM (e.g., via `openai` or `transformers`), relying on a pre-prompt for interpretation.
5. If not nudging, remain silent and continue to observe, record, and update internal state.

Example: In Python, this loop is implemented as a function that updates the Motivation Vector, computes confidence, and either calls a nudge-injection routine or skips to the next observation.

4.6 Meta-Rule Implementation

- Represent meta-rules as rules over the rule-graph using `networkx`.
- Define neighborhood $\mathcal{N}(m_i)$ of small metagraph edits.
- Apply the meta-rule update formula in your training loop alongside rule edits.

4.7 Natural Gradient Optimization

- Install `pot` (Python Optimal Transport) for Wasserstein solvers.
- Build ground metric matrix (ω_{ij}) from rule-graph distances.
- Construct measure-dependent Laplacian via NetworkX weights.
- Compute parameter Jacobians with `autograd` or manual derivatives.
- Perform updates using `numpy.linalg.pinv` for pseudoinverse.

4.8 Continuous Predictive-Coding Nets

- Use `PyTorch` or `JAX` to implement NGC-style layers:

$$z \leftarrow z + \beta\big(-\gamma z + (E \cdot e) \odot \phi'(z) - e\big)$$

  Explanation: z is the neural state (e.g., latent or hidden activations), β is the update step size, γ is a decay rate, E is a weight matrix, e is the prediction error, and $\phi'(z)$ is the derivative of the activation function. The update combines decay, error-driven feedback, and nonlinearity to iteratively refine z toward reducing prediction error, as in predictive coding networks.

- Leverage `torch.nn.Module` for vision and arm nets.
- Optimize with local Hebbian-like or standard optimizers (SGD) per PC update.

4.9 Genetic Tuning (Optional)

- Represent hyperparameters $(\alpha, \delta, \tau_c, \tau_u)$ as a NumPy array.
- Use `DEAP` or `pygad` for crossover and mutation over logged performance metrics.

4.10 Vector Stores and Long-Term Memory

- Integrate `faiss` or `chromadb` for storing past contexts/embeddings.

Null Action and Silence: *If the confidence C_t does not exceed the threshold τ_t, the agent performs the null action $\varnothing$, i.e., it remains silent and continues to observe, record, and learn without intervening.*

5 Considerations

While the SOUL Motivation Framework offers a mathematically unified approach to agent motivation, several limitations and open questions remain. First, the framework is currently theoretical and lacks empirical validation; its performance and scalability in real-world or large-scale simulated environments are yet to be demonstrated. The symbolic meta-graph and rule-based components, while interpretable, may face challenges in high-dimensional, noisy, or rapidly changing domains where purely neural or end-to-end learning may be more robust.

Alternative paths for future development include hybridizing the SOUL framework with more advanced neural architectures, such as deep reinforcement learning agents or transformer-based world models, to better handle perception and action in complex environments. Additionally, the meta-rule self-modification and genetic mixing mechanisms could be further explored using evolutionary computation or meta-learning techniques, potentially enabling agents to autonomously discover new motivational drives or adapt to novel tasks.

Finally, integration with large language models (LLMs) and other foundation models presents both opportunities and risks. While LLMs can interpret and naturalize symbolic

nudges, ensuring alignment and safety in open-ended interactions remains an open challenge. Careful evaluation and iterative refinement will be required as the framework transitions from theory to practice.

6 Conclusion

This guide unifies seminal research, architectural principles, and practical Python tools to implement SOUL's Motivation Framework. By following it, you'll build agents that self-modify, learn from surprise and progress, and nudge LLMs with precise, adaptive prompts.

The novel contributions of this work are threefold: (1) the explicit formalization of a motivation vector that integrates competence, novelty, and homeostasis in a unified mathematical framework; (2) the introduction of a symbolic meta-graph and meta-rule self-modification mechanism, enabling agents to adaptively restructure their motivational drives and reasoning patterns; and (3) the practical blueprint for integrating symbolic, neural, and genetic components in a modular, extensible architecture. Together, these elements advance the field by providing both a theoretical foundation and a roadmap for future implementation and experimentation.

References

[1] B. Goertzel. Discrete Active Predictive Coding for Goal-Guided Algorithmic Chemistry as a Potential Cognitive Kernel, 2024. arXiv:2412.16547 [cs.AI].

[2] D. Pathak *et al.*, Curiosity-Driven Exploration by Self-Supervised Prediction, 2017.

[3] C. Salge *et al.*, Curiosity-Driven RL Using Information-Gain, 2014.

[4] K. Friston. The Free-Energy Principle: A Unified Brain Theory?, 2010.

[5] J. Schmidhuber. Formal Theory of Creativity, Fun, and Intrinsic Motivation, 2010.

[6] P.-Y. Oudeyer, F. Kaplan, and V. V. Hafner. Intrinsic Motivation Systems for Autonomous Mental Development, 2007.

[7] A. S. Klyubin *et al.*, Empowerment: A Universal Agent-Centric Measure, 2005.

[8] M. Lungarella, G. Metta, R. Pfeifer, and G. Sandini. Developmental robotics: A survey, Connection Science, 15(4), 151-190, 2003.

[9] C. L. Hull. *Principles of Behavior: An Introduction to Behavior Theory.* New York: Appleton-Century, 1943.