Case-Enabled Reasoning Engine with Bayesian Representations for Unified Modeling (CEREBRUM)

Daniel Ari Friedman

Version 1.0 (2025-04-07)

-.-. . .-. . -... .-. ..- – —... / -.-. .- ... . -....- . -. .- -... .-.. . -.. / .-. . .- ... — -. .. -. –. / . -. –. .. -. . / .– .. - .... /
-... .- -.– . ...- -. / .-. . .–. .-. . ... . -. - .- - .. — -. ... / ..-. — .-. / ..- -. .. ..-. .. . -.. / – — -.. . .-.. .. -. –.

DOI: 10.5281/zenodo.15170908[1]

-.-. . .-. . -... .-. ..- – —... / -.-. .- ... . -....- . -. .- -... .-.. . -.. / .-. . .- ... — -. .. -. –. / . -. –. .. -. . / .– .. - .... /
-... .- -.– . ...- -. / .-. . .–. .-. . ... . -. - .- - .. — -. ... / ..-. — .-. / ..- -. .. ..-. .. . -.. / – — -.. . .-.. .. -. –.

Contents

[1] https://doi.org/10.5281/zenodo.15170908

CEREBRUM: Case-Enabled Reasoning Engine with Bayesian Representations for Unified Modeling

Daniel Ari Friedman

Active Inference Institute

ORCID: 0000-0001-6232-9096

Email: daniel@activeinference.institute

Version 1.0 (2025-04-07)

1 Abstract

This paper introduces Case-Enabled Reasoning Engine with Bayesian Representations for Unified Modeling (CEREBRUM). CEREBRUM is a synthetic intelligence framework that integrates linguistic case systems with cognitive scientific principles to describe, design, and deploy generative models in an expressive fashion. By treating models as case-bearing entities that can play multiple contextual roles (e.g. like declinable nouns), CEREBRUM establishes a formal linguistic-type calculus for cognitive model use, relationships, and transformations. The CEREBRUM framework uses structures from category theory and modeling techniques related to the Free Energy Principle, in describing and utilizing models across contexts. CEREBRUM addresses the growing complexity in computational and cognitive modeling systems (e.g. generative, decentralized, agentic intelligences), by providing structured representations of model ecosystems that align with lexical ergonomics, scientific principles, and operational processes.

2 Overview

CEREBRUM implements a comprehensive approach to cognitive systems modeling by applying linguistic case systems to model management. This framework treats cognitive models as entities that can exist in different "cases", as in a morphologically rich language, based on their functional role within an intelligence production workflow. This enables more structured representation of model relationships and transformations.

The code to generate this paper, and further open source development from this 1.0 milestone release, is available at https://github.com/ActiveInferenceInstitute/CEREBRUM .

3 Background

3.1 Cognitive Systems Modeling

Cognitive systems modeling approaches cognition as a complex adaptive system, where cognitive processes emerge from the dynamic interaction of multiple components across different scales. This perspective draws from ecological psychology's emphasis on organism-environment coupling, where cognitive processes are fundamentally situated in and shaped by their environmental context. The 4E cognition framework (embodied, embedded, enacted, and extended) provides a theoretical foundation for understanding how cognitive systems extend beyond individual agents to include environmental structures and social interactions. In this view, cognitive models are not merely internal representations but active participants in a broader cognitive ecosystem, where they adapt and evolve through interaction with other models and environmental constraints. This systems-level perspective is particularly relevant for intelligence production, where multiple analytical models must coordinate their activities while maintaining sensitivity to changing operational contexts and requirements. The complex adaptive systems approach emphasizes self-organization, emergence, and adaptation, viewing cognitive processes as distributed across multiple interacting components that collectively produce intelligent behavior through their coordinated activity (including language use).

3.2 Active Inference

Active Inference is a first-principles account of perception, learning, and decision-making based on the Free Energy Principle. In this framework, cognitive systems minimize variational free energy — bounded surprise, reflecting the difference between an organism's internal model and its environment — through perception (updating internal models) and action (changing action and ultimately sensory inputs). The Active Inference framework formalizes uncertainty in terms of entropy and precision weighting, enabling dynamic adaptive processes. While many model architectures are possible, hierarchical message passing is a common implementation that implements predictions as top-down flows and prediction errors as bottom-up flows, creating a bidirectional inference system that iteratively minimizes surprise across model levels. Active Inference treats all cognitive operations as Bayesian model update, providing a unifying mathematical formalism for predictive cognition.

3.3 Linguistic Case Systems

Linguistic case systems represent grammatical relationships between words through morphological marking. Case systems operate as morphosyntactic interfaces between semantics and syntax, encoding contextualized relationship types rather than just sequential ordering. This inherent relationality makes case systems powerful abstractions for modeling complex dependencies and transformations between conceptual entities. Cases under consideration here include nominative (subject), accusative (object), dative (recipient), genitive (possessor), instrumental (tool), locative (location), and ablative (origin), all serving different functional roles within sentence structures.

Languages implement these differently: nominative-accusative systems distinguish subjects from objects, while ergative-absolutive systems group intransitive subjects with direct objects. While English has largely lost its morphological case system, the underlying case relationships still exist and are expressed through word order and prepositions. For example, in "The cat chased the mouse," the nominative case is marked by position (subject before verb) rather than morphology, while in "I gave him the book," the dative case is marked by the preposition "to" (implied) and word order. This demonstrates that (the semantics/semiosis/pragmatics of) case relationships are fundamental to language structure, even when not overtly marked morphologically (e.g. expressed in writing or spoken language).

3.4 Intelligence Case Management Systems

Intelligence case management systems organize investigative workflows and analytical processes in operational contexts. These systems structure information collection, analysis, evaluation, and dissemination while tracking provenance and relationships between intelligence products. Modern implementations increasingly must manage complex model ecosystems where analytical tools, data sources, and products interact within organizational workflows. However, current frameworks lack formal mathematical foundations for representing model relationships, leading to ad hoc integration approaches that become unwieldy at scale. As artificial intelligence components proliferate in these systems, a more rigorous basis for model interaction becomes essential for maintaining operational coherence and analytical integrity.

4 Towards Languages for Generative Modeling

The Active Inference community has extensively explored numerous adjectival modifications of the base framework, including Deep, Affective, Branching-Time, Quantum, Mortal, Structured Inference, among others. Each adjectival-prefixed variant emphasizes specific architectural aspects or extensions of the core formalism. Building on this, CEREBRUM focuses on a wider range of linguistic formalism (e.g. in this paper, declensional semantics) rather than adjectival modifications.

In this first CEREBRUM paper, there is an emphasis on the declensional aspects of generative models as noun-like entities, separate from adjectival qualification. This approach aligns with category theoretic approaches to linguistics, where morphisms between objects formalize grammatical relationships and transformations. By applying formal case grammar to generative models, CEREBRUM extends and transposes structured modeling approaches to ecosystems of shared intelligence, while preserving the underlying (partitioned, flexible, variational, composable, interfacial, inter-active, empirical, applicable, communicable) semantics.

5 Conceptual Foundations: The Intersection of Four Domains

CEREBRUM integrates four key domains to create a unified framework for model management (Figure 1):

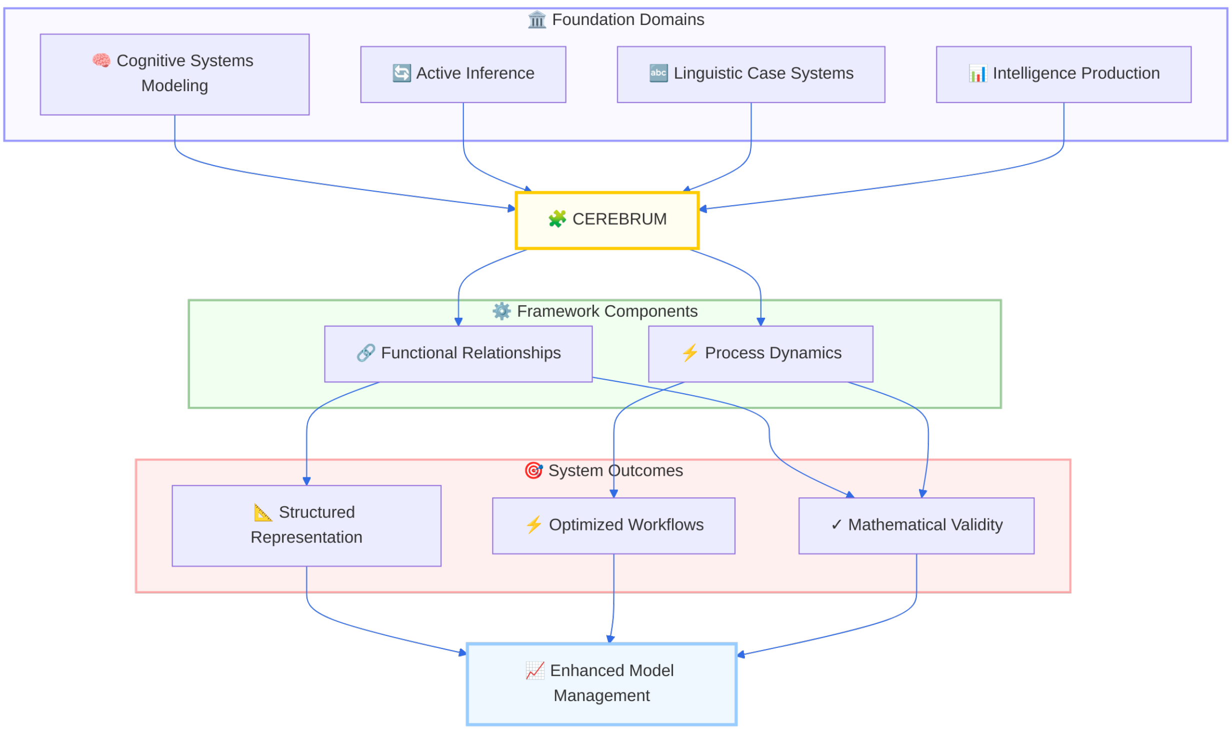

Figure 1: Foundation Domains of CEREBRUM. The diagram shows the four key domains (Cognitive Systems Modeling, Active Inference, Linguistic Case Systems, and Intelligence Production) and their integration through the CEREBRUM core to produce enhanced model management capabilities.

1. **Cognitive Systems Modeling** offers the entities that take on case relationships
2. **Active Inference** supplies the predictive processing mechanics that drive case transformations
3. **Linguistic Case Systems** provide the grammatical metaphor for how models relate to each other
4. **Intelligence Production** furnishes the practical application context and workflows

6 Methods and Materials

6.1 Formal Framework Development

The CEREBRUM framework was developed as a part of a broader synthetic intelligence framework, combining linguistic theory, cognitive science, category theory, and operations research. Key methodological approaches included:

1. **Linguistic Formalization**: Adapting morphosyntactic case theory into computational representations through abstract algebraic structures.
2. **Category-Theoretic Mapping**: Implementing category theory to formalize morphisms between case states as functorial transformations.
3. **Algorithmic Implementation**: Developing algorithmic specifications for case transformations compliant with the Free Energy Principle.
4. **Variational Methods**: Applying variational free energy calculations to optimize model inference as well as structural transformations.

6.2 Mathematical Foundation

The mathematical foundation of CEREBRUM builds on formalizations of case transformations using category theory and variational inference. Case transformations are modeled as morphisms in a category where objects are models with specific case assignments. The framework employs metrics including Kullback-Leibler divergence, Fisher information, and Lyapunov functions to quantify transformation efficacy and system stability. This approach provides both theoretical guarantees of compositional consistency and practical optimization methods for computational implementation.

7 Core Concept: Cognitive Models as Case-Bearing Entities

Just as nouns in morphologically rich languages take different forms based on their grammatical function, cognitive models in CEREBRUM can exist in different "states" or "cases" depending on how they relate to other models or processes within the system. Figure 2 illustrates this linguistic parallel.

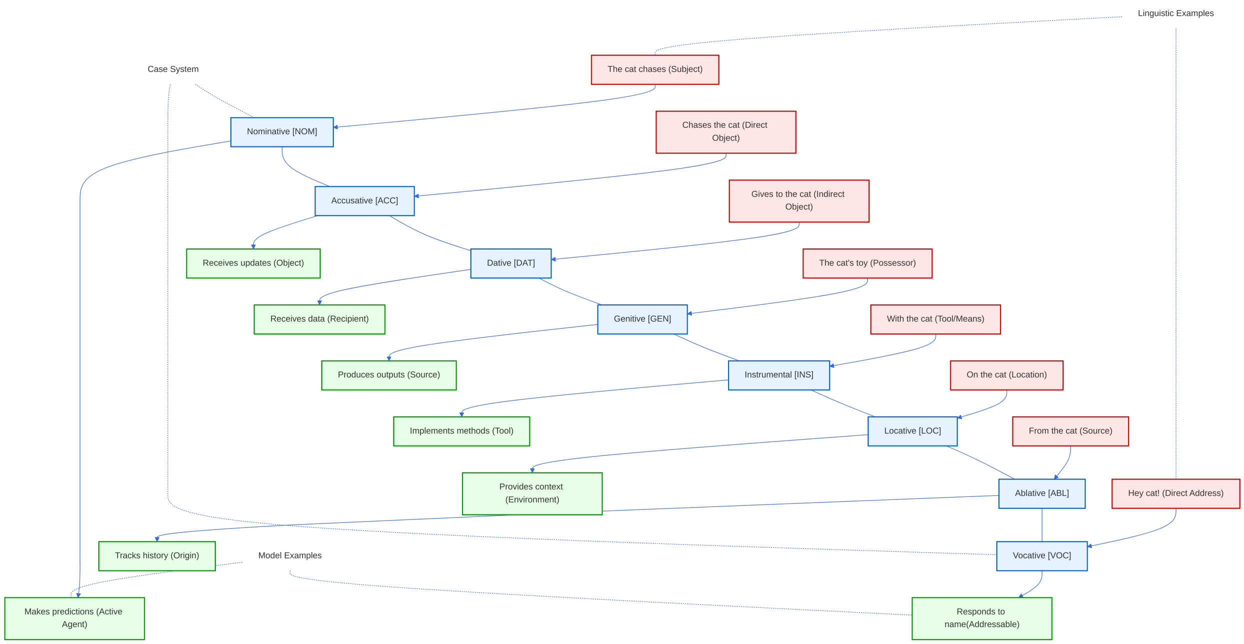

Figure 2: Case Relationships - Model and Linguistic Parallels. The diagram illustrates parallel case relationships between a generative model and linguistic examples, demonstrating how model cases mirror grammatical roles in natural language.

8 Case Functions in Cognitive Modeling

Each case defines a specific relationship type between models or between models and data (Table 1). The basic framework is depicted in Figure 3.

Table 1: Case Functions in Cognitive Model Systems

Abbr	Case	Function in CEREBRUM	Example Usage
[NOM]	**Nominative**	Model as active agent; acts as the primary producer of predictions and exerts causal influence on other models	Model X [NOM] generates predictions about data distributions; controls downstream processing
[ACC]	**Accusative**	Model as object of process; receives transformations and updates from other models or processes	Process applies to Model X [ACC]; optimization procedures refine Model X's parameters
[GEN]	**Genitive**	Model as source/possessor; functions as the origin of outputs, products, and derived models	Output of Model X [GEN]; intelligence products derived from Model X's inferences

Abbr	Case	Function in CEREBRUM	Example Usage
[DAT]	**Dative**	Model as recipient; specifically configured to receive and process incoming data flows	Data fed into Model X [DAT]; Model X receives information from external sources
[INS]	**Instrumental**	Model as method/tool; serves as the means by which analytical operations are performed	Analysis performed via Model X [INS]; Model X implements analytical procedures
[LOC]	**Locative**	Model as context; provides environmental constraints and situational parameters	Parameters within Model X [LOC]; environmental contingencies modeled by X
[ABL]	**Ablative**	Model as origin/cause; represents historical conditions or causal precursors	Insights derived from Model X [ABL]; causal attributions traced to Model X
[VOC]	**Vocative**	Model as addressable entity; functions as a directly callable interface with name-based activation	"Hey Model X" [VOC]; direct invocation of Model X for task initialization; documentation reference point

Within intelligence production systems, these case relationships serve critical functional roles: nominative models act as primary analytical engines driving the intelligence case; accusative models become targets of quality assessment and improvement; multimodal genitive models generate documentation and reports; dative models receive and process collected intelligence data; instrumental models provide the methodological framework for investigations; locative models establish situational boundaries; ablative models represent the historical origins of analytical conclusions; and vocative models serve as directly addressable interfaces for command initiation and documentation reference. Together, these case relationships create a comprehensive framework for structured intelligence workflows.

Figure 4 illustrates how this core framework integrates with intelligence case management.

9 A Preliminary Example of a Case-Bearing Model: Homeostatic Thermostat

Consider a cognitive model of a homeostatic thermostat that perceives room temperature with a thermometer, and regulates temperature through connected heating and cooling systems. In

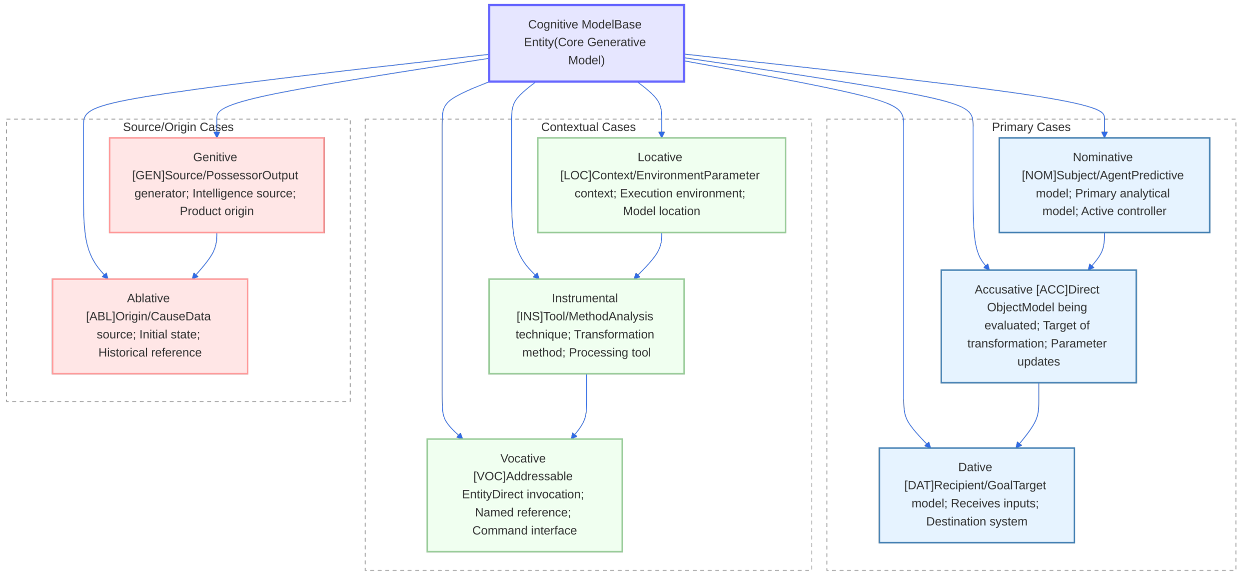

Figure 3: Cognitive Model Case Framework. The hierarchical organization of case types in CEREBRUM, showing primary, source, and contextual declensions with their functional relationships to the core generative model.

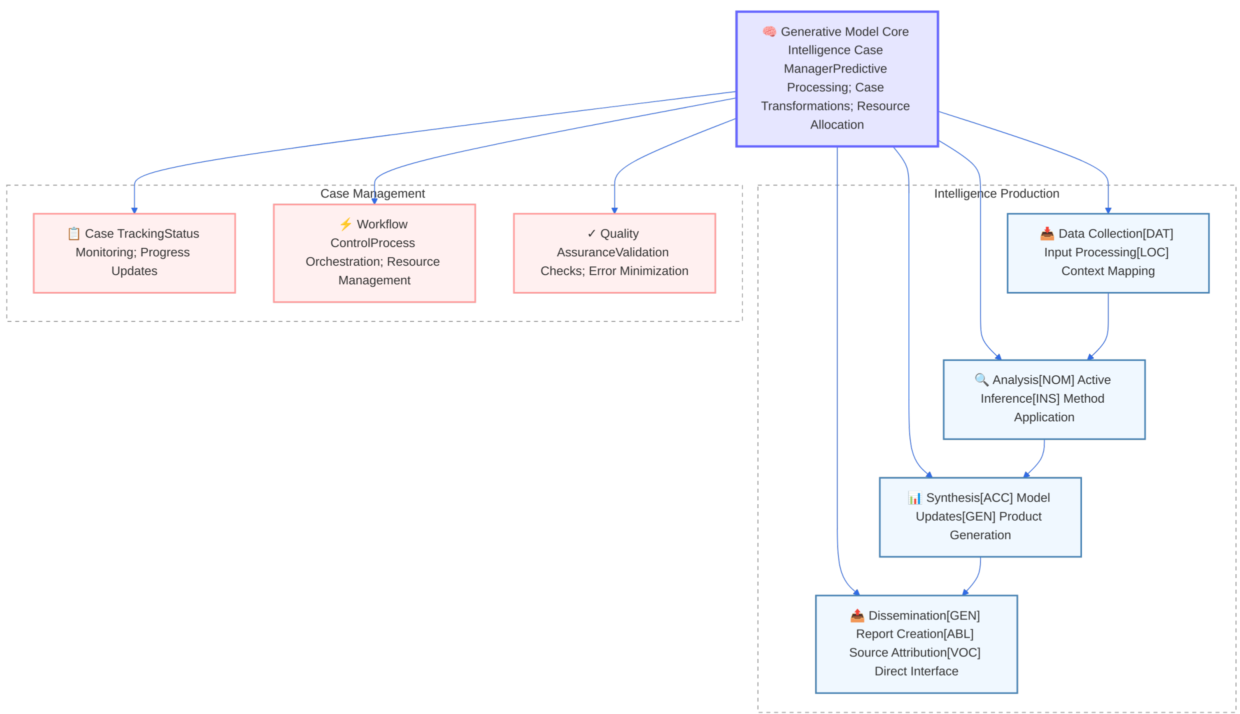

Figure 4: Generative Model Integration in Intelligence Case Management. Illustrates how CEREBRUM's generative model core orchestrates intelligence production and case management through case-specific transformations.

nominative case [NOM], the thermostat model actively generates temperature predictions and dispatches control signals, functioning as the primary agent in the temperature regulation process. When placed in accusative case [ACC], this same model becomes the object of optimization processes, with its parameters being updated based on prediction errors between expected and actual temperature readings. In dative case [DAT], the thermostat model receives environmental temperature data streams and occupant comfort preferences as inputs. The genitive case [GEN] transforms the model into a generator of temperature regulation reports and system performance analytics ("genitive AI"). When in instrumental case [INS], the thermostat serves as a computational tool implementing control algorithms for other systems requiring temperature management. The locative case [LOC] reconfigures the model to represent the contextual environment in which temperature regulation occurs, modeling building thermal properties, or discussing something within the model as a location. Finally, in ablative case [ABL], the thermostat functions as the origin of historical temperature data and control decisions, providing causal explanations for current thermal conditions. This single cognitive model thus assumes dramatically different functional roles while maintaining its core identity as a thermostat.

10 Declinability of Active Inference Generative Models

At the core of CEREBRUM lies the concept of **declinability** - the capacity for generative models to assume different morphological and functional roles through case transformations, mirroring the declension patterns of nouns in morphologically rich languages. Unlike traditional approaches where models maintain fixed roles, or variable roles defined by analytical pipelines, CEREBRUM treats cognitive models as flexible entities capable of morphological adaptation to different operational contexts.

10.1 Morphological Transformation of Generative Models

When an active inference generative model undergoes case transformation, it experiences orchestrated systematic changes summarized in Table 2:

1. **Functional Interfaces**: Input/output specifications change to match the case role requirements
2. **Parameter Access Patterns**: Which parameters are exposed or constrained changes based on case
3. **Prior Distributions**: Different cases employ different prior constraints on parameter values
4. **Update Dynamics**: The ways in which the model updates its internal states vary by case role
5. **Computational Resources**: Different cases receive different precision-weighted computational allocations

Table 2: Transformational Properties of Active Inference Generative Models Under Case Declensions

Case	Parametric Changes	Interface Transformations	Precision Weighting
[NOM]	Fully accessible parameters; all degrees of freedom available for prediction generation; strongest prior constraints on likelihood mapping	Outputs predictions; exposes forward inference pathways; prediction interfaces activated	Highest precision on likelihood; maximizes precision of generative mapping from internal states to observations
[ACC]	Restricted parameter access; plasticity gates opened; learning rate parameters prioritized	Receives transformations; update interfaces exposed; gradient reception pathways active	Highest precision on parameters; maximizes precision of parameter updates based on prediction errors
[DAT]	Input-focused parameterization; sensory mapping parameters prioritized; perceptual categorization parameters activated	Receives data flows; input processing interfaces exposed; sensory reception channels active	Highest precision on inputs; maximizes precision of incoming data relative to internal expectations
[GEN]	"Genitive AI"; Output-focused parameterization; production parameters activated; generative pathway emphasis	Generates products; output interfaces prioritized; production pathways activated	Highest precision on outputs; maximizes precision of generated products relative to internal models
[INS]	Method-oriented parameters exposed; algorithmic parameters accessible; procedural knowledge emphasized	Implements processes; computational interfaces active; procedural execution pathways open	Highest precision on operations; maximizes precision of procedural execution relative to methodological expectations
[LOC]	Context parameters emphasized; environmental modeling parameters prioritized; situational knowledge emphasized	Provides environmental constraints; contextual interfaces active; environmental modeling pathways prioritized	Highest precision on contexts; maximizes precision of contextual representation relative to environmental dynamics

Case	Parametric Changes	Interface Transformations	Precision Weighting
[**ABL**]	Origin states emphasized; historical parameters accessible; causal attribution pathways strengthened	Source of information; historical data interfaces active; causal explanation pathways open	Highest precision on historical data; maximizes precision of causal attributions and historical reconstructions
[**VOC**]	Identity parameters prioritized; naming and identification parameters activated; interface exposure emphasized	Maintains addressable interfaces; name recognition pathways activated; command reception channels open	Highest precision on identification cues; maximizes precision of name recognition relative to calling patterns

10.2 Active Inference Model Declension Example

Consider a perception-oriented generative model M with parameters theta, internal states s, and observational distribution p(o|s,theta). When declined across cases, this single model transforms as follows:

- **M[NOM]**: Actively generates predictions by sampling from p(o|s,theta), with all parameters fully accessible
- **M[ACC]**: Becomes the target of updates, with parameter gradients calculated from prediction errors
- **M[DAT]**: Configured to receive data flows, with specific input interfaces activated
- **M[GEN]**: Optimized to generate outputs, with output interfaces prioritized
- **M[INS]**: Functions as a computational method, exposing algorithmic interfaces
- **M[LOC]**: Provides contextual constraints for other models, with environmental parameters exposed
- **M[ABL]**: Serves as an information source, with historical data accessible
- **M[VOC]**: Functions as an addressable entity responding to direct invocation, with naming parameters activated

The Vocative case [VOC] represents a unique functional role where models serve as directly addressable entities within a model ecosystem. Unlike other cases that focus on data processing or transformational aspects, the vocative case specifically optimizes a model for name-based recognition and command reception. This has particular relevance in synthetic intelligence environments where models must be selectively activated or "woken up" through explicit address, similar to how humans are called by name to gain their attention. The vocative case maintains specialized interfaces for handling direct commands, documentation references, and initialization requests. In practical applications, models in vocative case might serve as conversational agents awaiting activation, documentation reference points within technical specifications, or system components that remain dormant until explicitly addressed. This pattern mimics the linguistic vocative case where a noun is used in direct address, as in “Hey Siri” or “OK Google” activation phrases for digital assistants, creating a natural bridging pattern between human language interaction and model orchestration.

This systematic pattern of transformations constitutes a complete “declension paradigm” for cognitive models, using precision-modulation to fulfill diverse functional roles while maintaining their core identity.

11 Model Workflows as Case Transformations

Case transformations represent operations that change the functional role of a model in the system, reflecting active inference principles of prediction and error minimization. Figure 5 provides a sequence diagram of a typical transformation cycle, and Figure 6 shows the intelligence production workflow where these transformations occur.

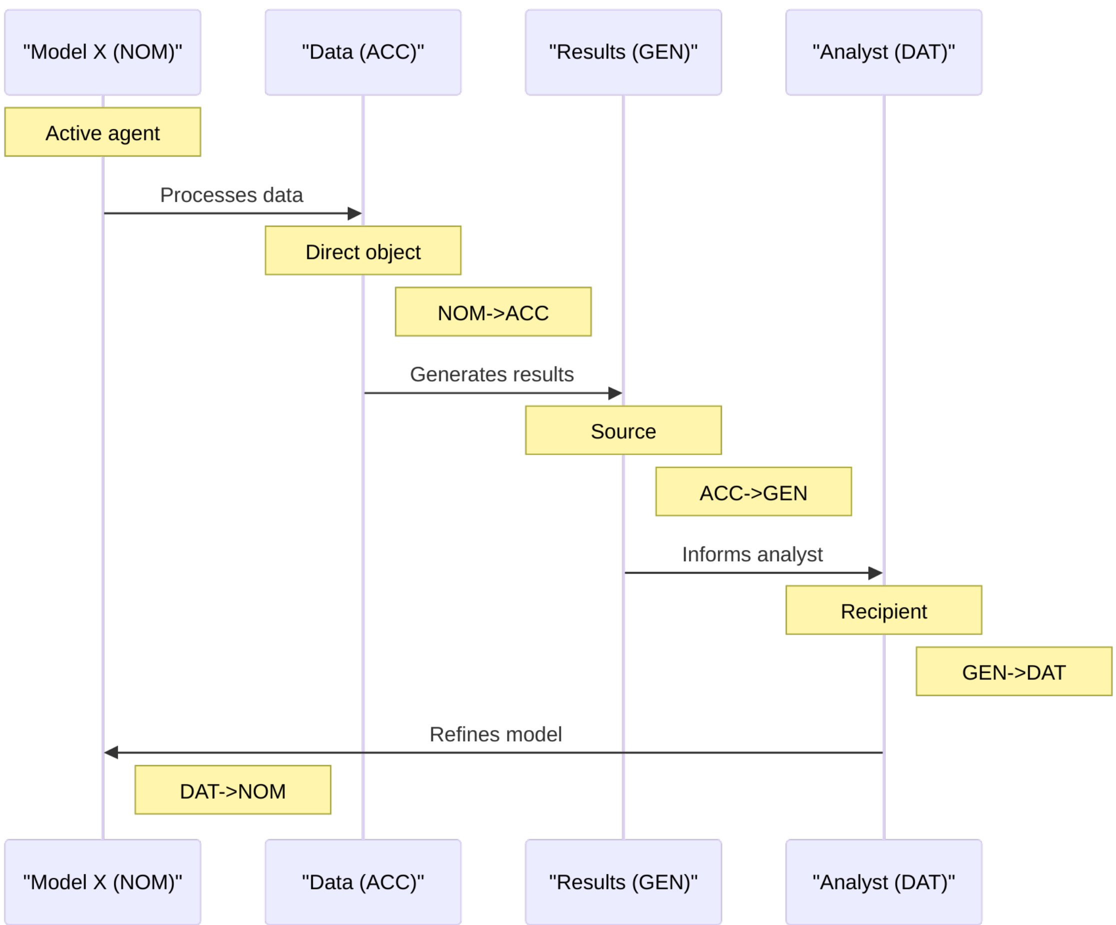

Figure 5: Model Workflows as Case Transformations - Sequence Diagram 1. Illustrates the temporal sequence of case transformations as models transition through different functional roles in an intelligence workflow.

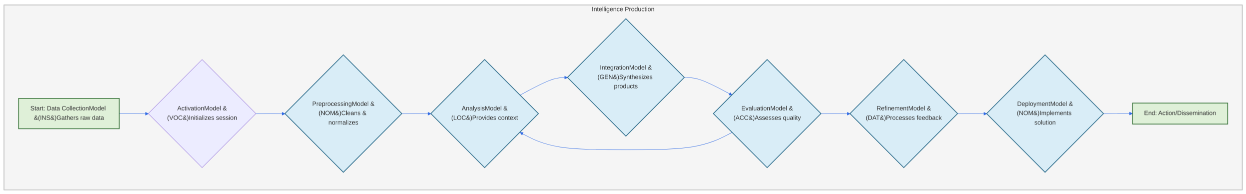

Figure 6: Intelligence Production Workflow with Case-Bearing Models. Illustrates the intelligence production cycle, showing the stages where models with different case assignments participate.

12 Category-Theoretic Formalization

CEREBRUM employs category theory to formalize case relationships between cognitive models, creating a rigorous mathematical foundation, illustrated in Figure 7 and Figure 8.

13 Computational Linguistics, Structural Alignment, and Model Relationships

CEREBRUM supports different alignment systems for model relationships, mirroring linguistic morphosyntactic structures (Figure 9). These alignment patterns determine how models interact and transform based on their functional roles. Figure 9 illustrates the core alignment patterns derived from linguistic theory, showing how models can be organized based on their case relationships. This includes nominative-accusative alignment (where models are distinguished by their role as agents or patients), ergative-absolutive alignment (where models are grouped by their relationship to actions), and tripartite alignment (where each case is marked distinctly).

Figure 10 demonstrates the practical implementation of these alignment patterns in model ecosystems, showing how different alignment systems affect model interactions and transformations. The diagram illustrates the computational implications of each alignment pattern, including resource allocation, message passing, and transformation efficiency. This implementation view complements the theoretical alignment patterns shown in Figure 9 by demonstrating their practical application in cognitive model management.

14 Implementation in Intelligence Production

As mentioned, CEREBRUM integrates with intelligence case management through structured workflows (see Figures 4 and 6). Figure 11 and Figure 12 provide alternative state-based visualizations of these workflows.

The intelligence production workflow begins with raw data collection, where models in instrumental case [INS] serve as data collection tools, implementing specific methods for information gathering. As data moves through preprocessing, models transition to nominative case [NOM], taking on active processing roles to clean, normalize, and prepare the data for analysis. During analysis, models assume locative case [LOC], providing contextual understanding and environmental parameters

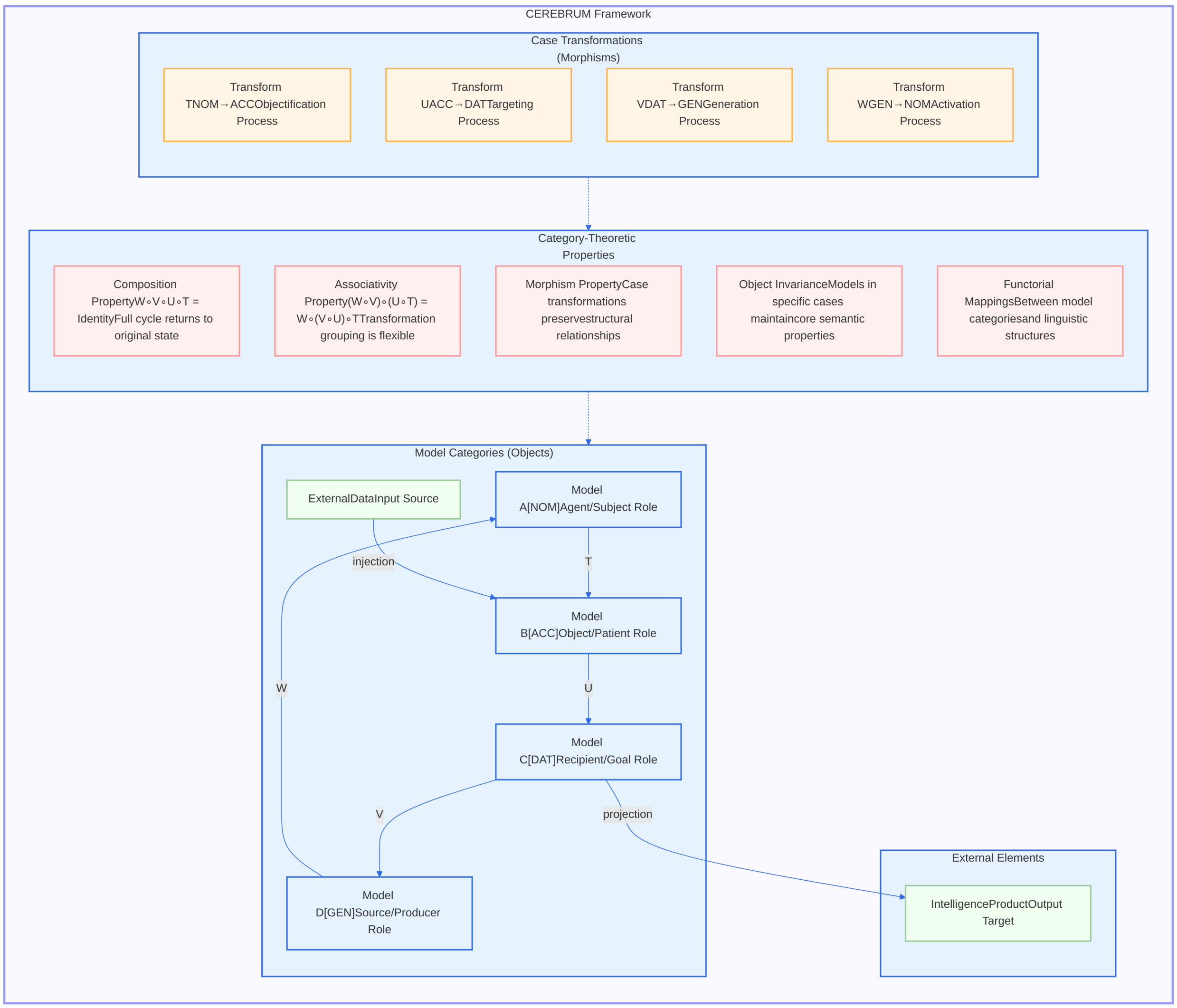

Figure 7: CEREBRUM Category Theory Framework. Demonstrates the category-theoretic formalization of case relationships and transformations between cognitive models.

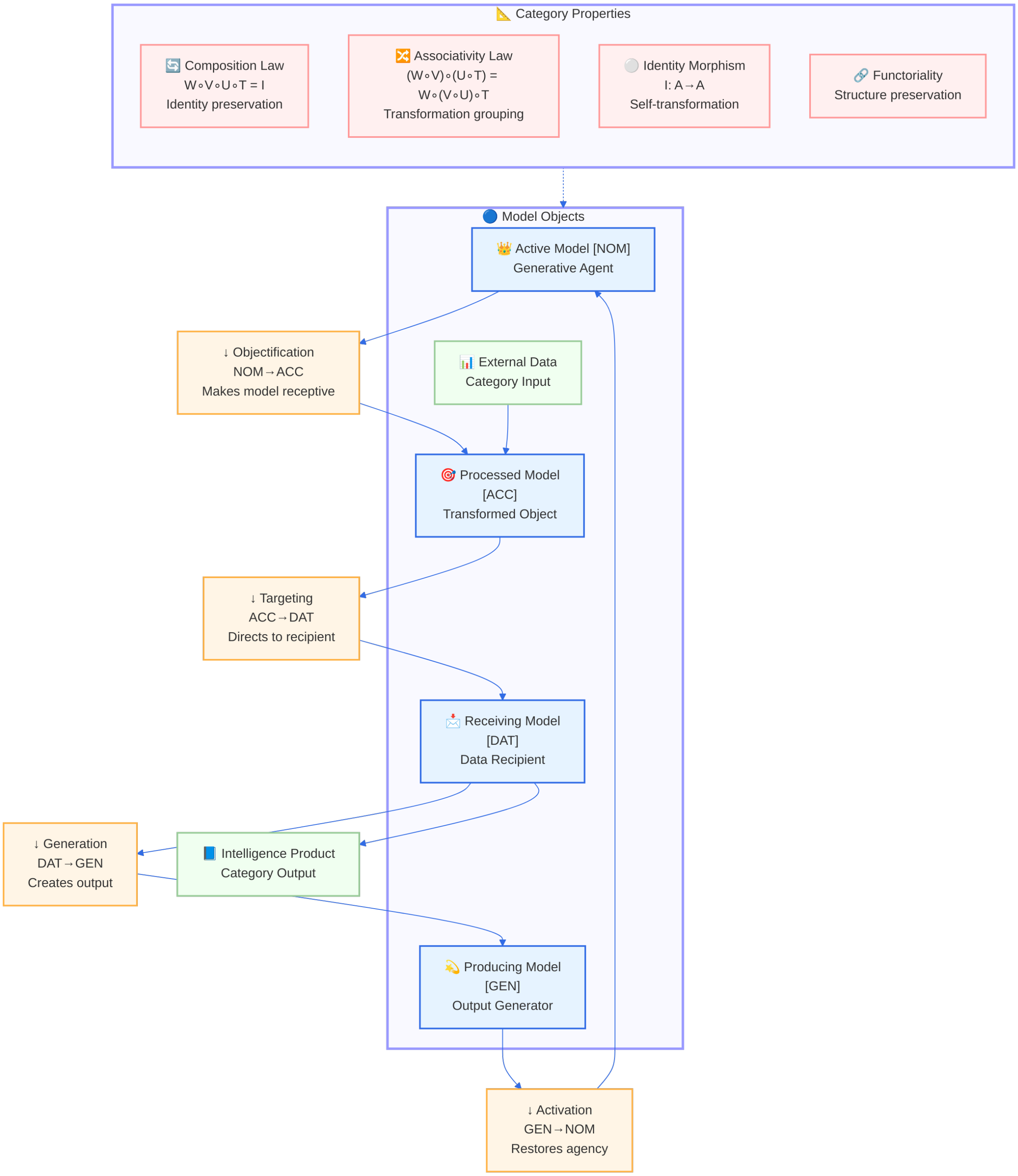

Figure 8: Category Theory Framework (Alternative View). Further illustrates the category-theoretic components and properties within CEREBRUM.

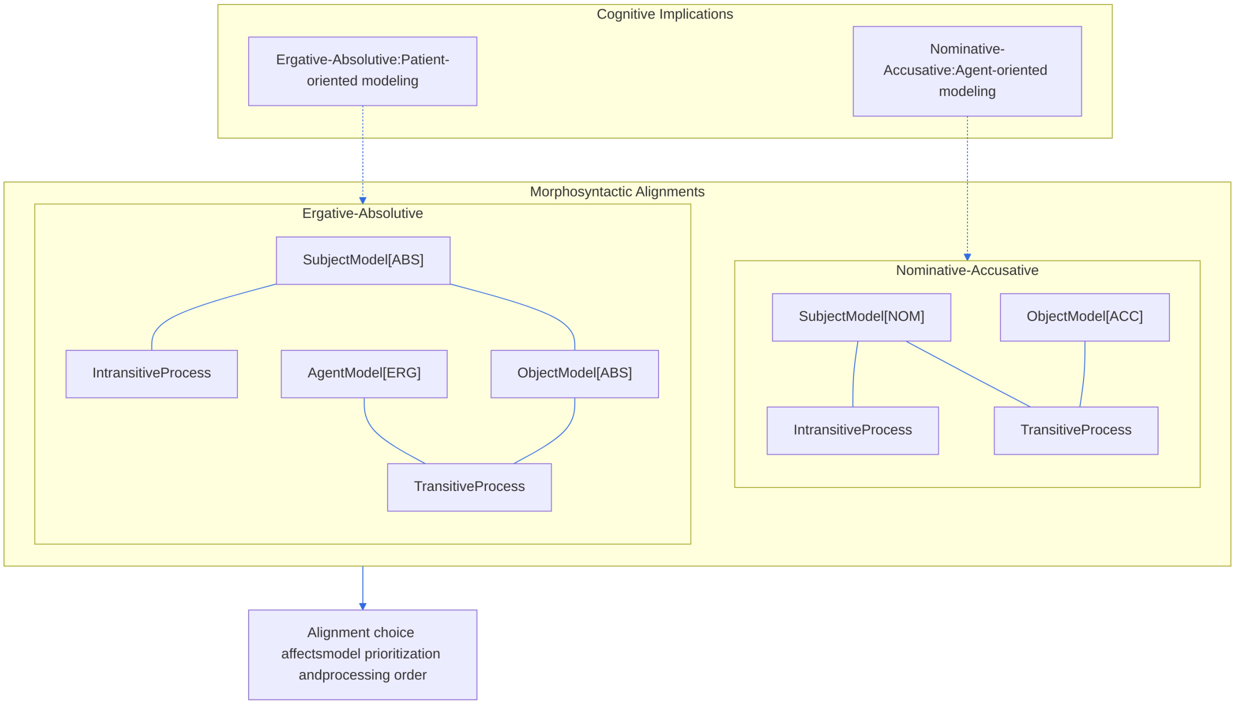

Figure 9: Morphosyntactic Alignments in Model Relationships. Shows how CEREBRUM implements different alignment patterns for model relationships based on linguistic morphosyntactic structures.

that shape the analytical process.

Integration represents a critical transition point where models in genitive case [GEN] generate intelligence products by synthesizing information from multiple sources. These products then undergo evaluation by models in accusative case [ACC], which assess quality and identify areas for improvement. The refinement phase employs models in dative case [DAT] to process feedback and implement necessary changes, while deployment returns models to nominative case [NOM] for active implementation of refined solutions.

This cyclical process demonstrates how case transformations enable models to maintain their core identity while adapting to different functional requirements throughout the intelligence production lifecycle. Each case assignment optimizes specific aspects of model behavior, from data collection and processing to product generation and quality assessment, creating a flexible yet structured approach to intelligence production.

15 Active Inference Integration

CEREBRUM aligns with active inference frameworks by treating case transformations as predictive processes within a free energy minimization framework, as illustrated in Figure 13. Figure 14 details the associated message passing rules.

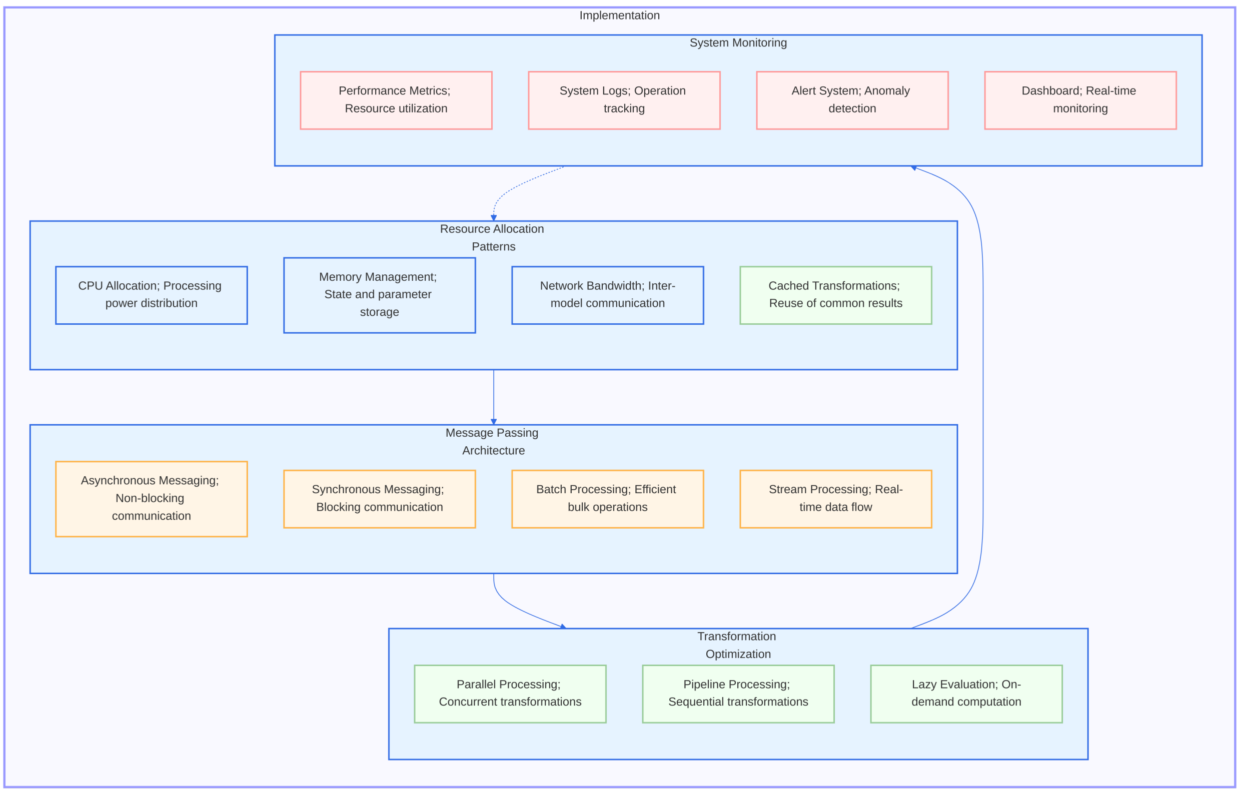

Figure 10: Computational Implementation of Model Relationships. Illustrates the practical implementation details of model relationships in CEREBRUM, including resource allocation patterns, message passing efficiency, and transformation optimization strategies.

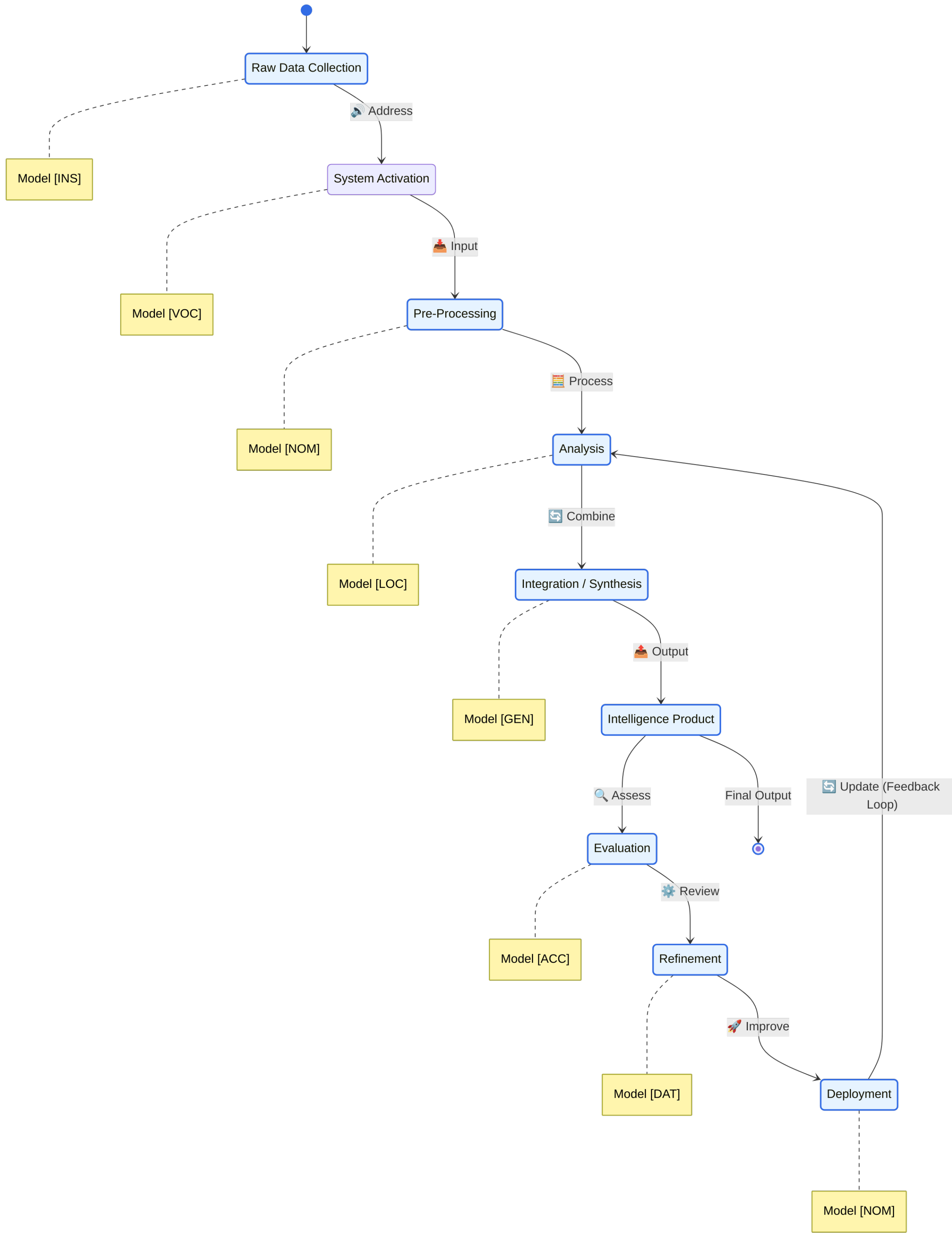

Figure 11: Implementation in Intelligence Production - State Diagram. Provides a state-based view of the intelligence workflow highlighting model case assignments at each stage.

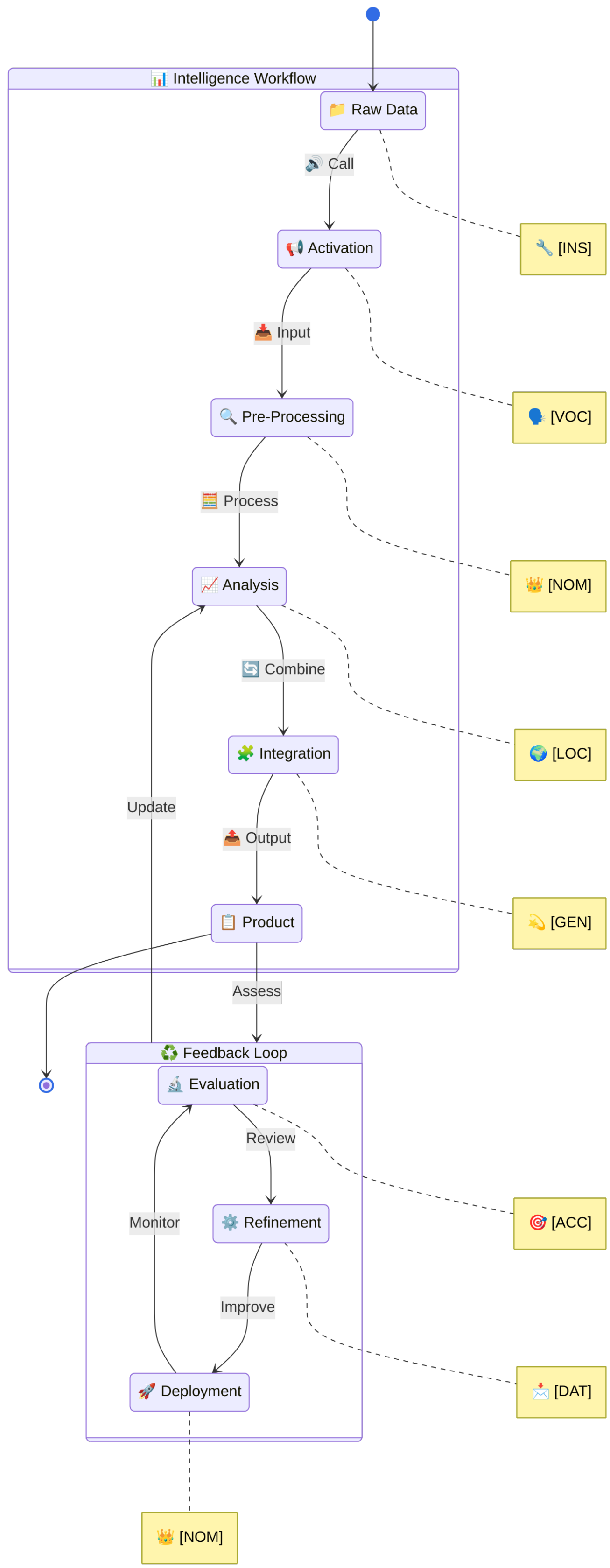

Figure 12: Intelligence Workflow (Alternative View). Presents another perspective on the intelligence production cycle and feedback loops, emphasizing case roles.

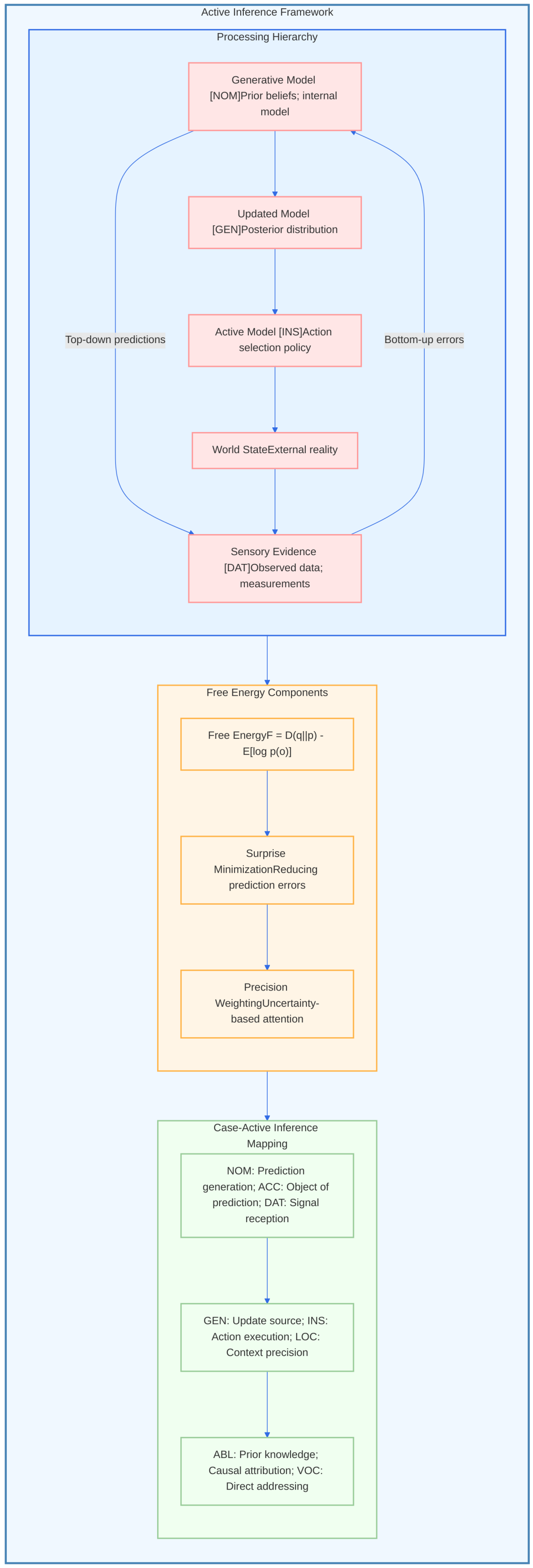

Figure 13: Active Inference Integration Framework. Shows how active inference principles are integrated with case transformations through precision-weighted message passing and free energy minimization.

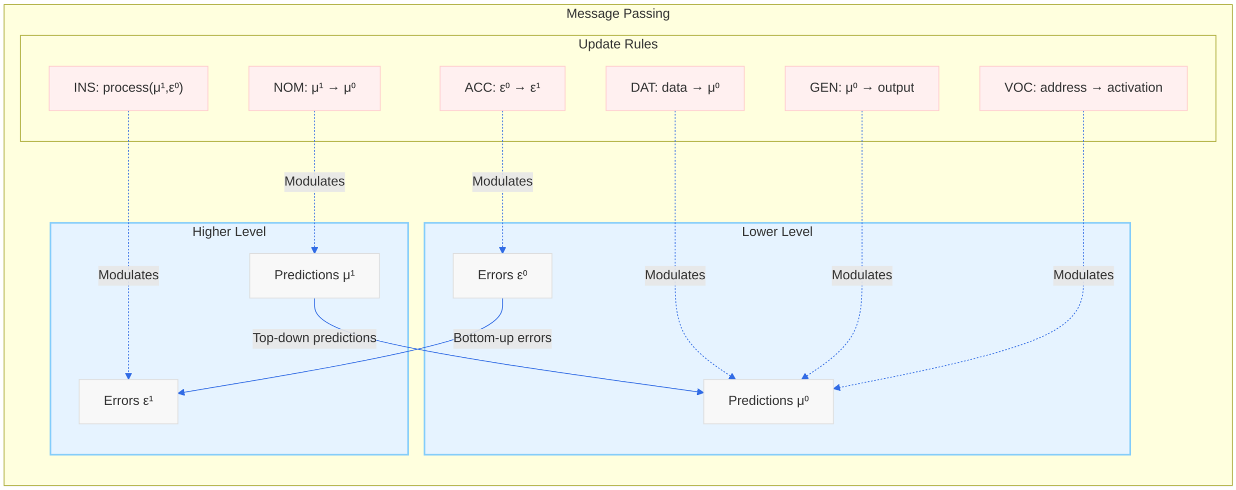

Figure 14: Case-Specific Message Passing in Active Inference. Illustrates how message passing dynamics change based on the model's current case assignment within an active inference hierarchy.

16 Formal Case Calculus

The relationships between case-bearing models follow a formal calculus derived from grammatical case systems, presented in Figure 15.

17 Cross-Domain Integration Benefits

The CEREBRUM framework delivers several advantages through its integration of the four foundational domains:

Table 4: Cross-Domain Integration Benefits in CEREBRUM Framework

Domain	Contribution	Benefit to CEREBRUM	Theoretical Significance
Linguistic Case Systems	Systematic relationship framework; grammatical role templates; morphosyntactic structures	Structured representation of model interactions; formalized functional transitions; systematic role assignment	Provides formal semantics for model relationships; enables compositional theory of model interactions; grounds functions in linguistic universals

Domain	Contribution	Benefit to CEREBRUM	Theoretical Significance
Cognitive Systems Modeling	Entity representation and processing; model formalization; information-processing structures	Flexible model instantiation across functional roles; adaptive model morphology; unified modeling paradigm	Advances theory of cognitive model composition; formalizes functional transitions in cognitive systems; bridges symbolic and statistical approaches
Active Inference	Predictive transformation mechanics; free energy principles; precision-weighted learning	Self-optimizing workflows with error minimization; principled uncertainty handling; bidirectional message passing	Extends active inference to model ecosystems; provides mathematical foundation for case transformations; unifies perception and model management
Intelligence Production	Practical operational context; analytical workflows; intelligence cycle formalisms	Real-world application in case management systems; operational coherence; analytical integrity	Bridges theoretical and applied intelligence; enhances intelligence workflow coherence; improves analytical product quality

18 Related Work

CEREBRUM builds upon several research traditions while offering a novel synthesis. In this first paper, there are no specific works linked or cited. Later work will provide more detail in reference and derivation. The work stands transparently on the shoulders of nestmates and so is presented initially as a speculative design checkpoint in the development of certain cognitive modeling practices.

Related approaches include:

18.1 Cognitive Architectures

Existing cognitive architectures such as ACT-R, Soar, and CLARION provide comprehensive frameworks for modeling cognitive processes but lack formal mechanisms for representing functional role transitions. Unlike these systems, CEREBRUM explicitly models the morphological transformations of computational entities as they move through different processing contexts.

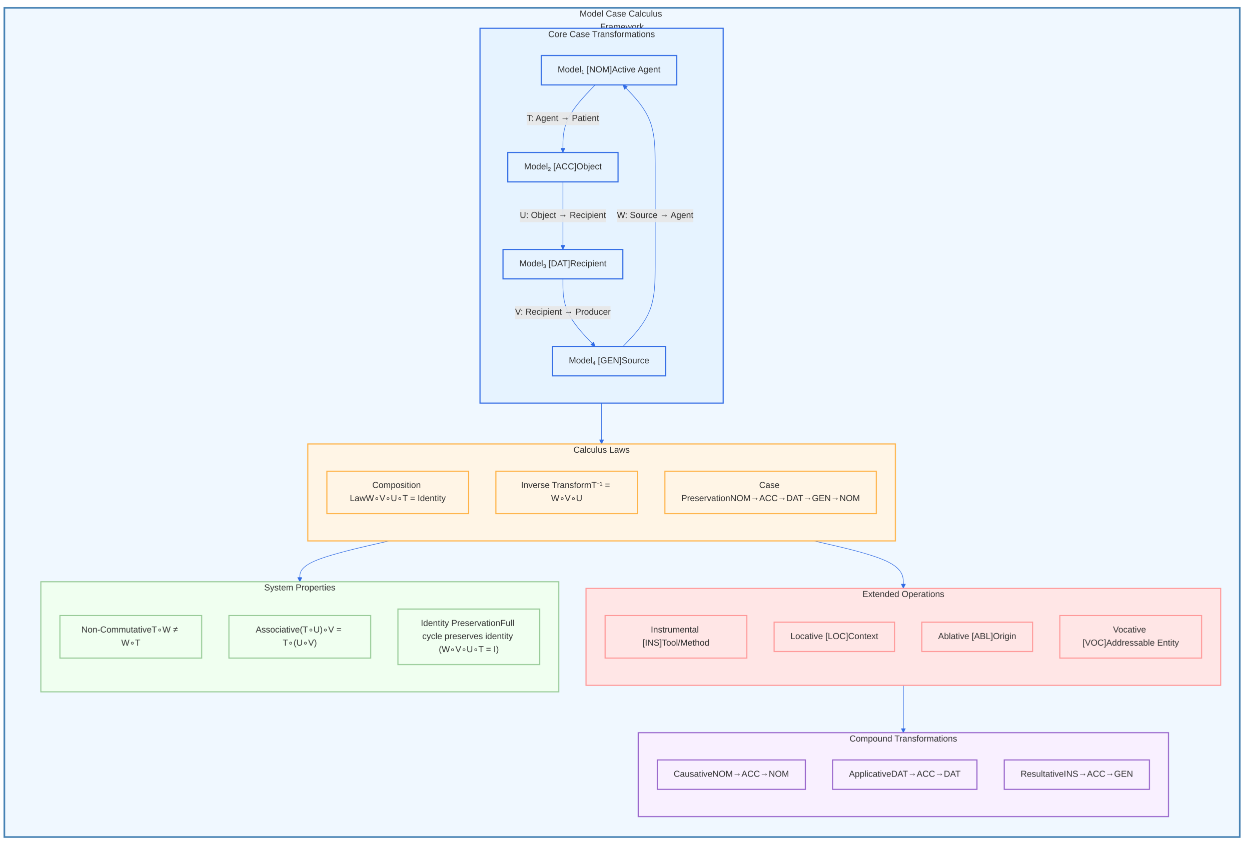

Figure 15: Model Case Calculus Framework. Presents the formal mathematical relationships and transformation rules that govern case transitions in the CEREBRUM framework.

18.2 Category-Theoretic Cognition

Recent work applying category theory to cognitive science has established mathematical foundations for cognitive processes. CEREBRUM extends this tradition by applying categorical structures specifically to case relationships and active inference, focusing on practical applications in intelligence production rather than purely theoretical constructs.

18.3 Active Inference Applications

Prior applications of active inference to artificial intelligence have focused primarily on perception and action in individual agents. CEREBRUM expands this domain by applying active inference principles to model ecosystems, where multiple models interact within structured workflows guided by case-based transformations.

18.4 Linguistic Computing

Computational linguistics has extensively employed case grammar for natural language processing, but rarely extended these principles to model management. CEREBRUM repurposes linguistic case theory as a structural framework for model relationships rather than textual analysis.

(See Appendix 2: Novel Linguistic Cases for a discussion of how CEREBRUM can discover and create new linguistic cases beyond traditional case systems.)

(See Appendix C: Practical Applications for detailed implementations of CEREBRUM in model ecosystems.)

19 Future Directions

Future work on the CEREBRUM framework will focus on both theoretical expansions and practical implementations:

- **Programming Libraries**: Developing robust programming libraries implementing the CEREBRUM framework across multiple languages to facilitate adoption
- **Visualization Tools**: Creating interactive visualization tools for case transformation processes to enhance understanding and analysis
- **Linguistic Extensions**: Expanding the framework to incorporate additional linguistic features such as aspect, tense, and modality into model relationship representations
- **Open Source Stewardship**: Establishing open source governance and community development practices through the Active Inference Institute
- **Computational Complexity**: Deriving formal computational complexity estimates for case transformations in various model ecosystem configurations
- **Multiple Dispatch Systems**: Implementing multiple dispatch architectures for programming languages to efficiently handle case-based polymorphism
- **Database Methods**: Developing specialized database structures and query languages for efficient storage and retrieval of case-bearing models

- **Cognitive Security**: Exploring security implications of case-based systems, including authorization frameworks based on case relationships

20 Conclusion

CEREBRUM provides a structured framework for managing cognitive models by applying linguistic case principles to represent different functional roles and relationships. This synthesis of linguistic theory, category mathematics, active inference, and intelligence production creates a powerful paradigm for understanding and managing complex model ecosystems. By treating models as case-bearing entities, CEREBRUM enables more formalized transformations between model states while providing intuitive metaphors for model relationships that align with human cognitive patterns and operational intelligence workflows.

The formal integration of variational free energy principles with case transformations establishes CEREBRUM as a mathematically rigorous framework for active inference implementations. The precision-weighted case selection mechanisms, Markov blanket formulations, and hierarchical message passing structures provide computationally tractable algorithms for optimizing model interactions. These technical formalizations bridge theoretical linguistics and practical cognitive modeling while maintaining mathematical coherence through category-theoretic validation.

The CEREBRUM framework represents another milestone in a long journey of how we conceptualize model relationships, moving from ad hoc integration approaches, on through seeking the first principles of persistent, composable, linguistic intelligences. This journey, really an adventure, continues to have profound implications for theory and practice. By here incipiently formalizing the grammatical structure of model interactions, CEREBRUM points towards enhancement of current capabilities and opens new avenues for modeling emergent behaviors in ecosystems of shared intelligence. As computational systems continue to grow in complexity, frameworks like CEREBRUM that provide structured yet flexible approaches to model management will become increasingly essential for maintaining conceptual coherence and operational effectiveness.

Appendix A 1: Mathematical Formalization

21 Mathematical Appendix

This appendix contains all mathematical formalizations referenced throughout the paper, organized by equation number.

21.1 Variational Free Energy and Case Transformations

Equation 1: Variational Free Energy for Case Transformation

$$F = D_{KL}[q(s|T(m))||p(s|m)] - \mathbb{E}_p[\log p(o|s, T(m))] \tag{1}$$

where T(m) represents the transformed model, s are internal states, and o are observations.

Equation 2: Markov Blanket and Case Relationship

$$\text{Case}(M) \subseteq \text{MB}(M) \tag{2}$$

where MB(M) denotes the Markov blanket of model M.

Equation 3: Precision Weighting for Case Selection

$$\beta(c, m) = \frac{\exp(-F(c, m))}{\sum_i \exp(-F(c_i, m))} \tag{3}$$

where (c,m) is the precision weight for case c and model m.

Equation 4: Case-Specific Gradient Descent on Free Energy

$$\frac{\partial m}{\partial t} = -\kappa_c \cdot \frac{\partial F}{\partial m} \tag{4}$$

where κ_c is the case-specific learning rate.

Equation 5: Expected Free Energy Reduction in Case Transitions

$$\mathbb{E}[\Delta F] = \sum_{s,a} T(s'|s, a)\pi[a|s](F(s, c) - F(s', c')) \tag{5}$$

where c and c' represent the initial and target cases respectively.

Equation 6: Bayes Factor for Case Selection

$$BF = \frac{p(o|m, c_1)}{p(o|m, c_2)} \tag{6}$$

Equation 7: Free Energy Minimization in Case Transitions

$$F = D_{KL}[q(s|c, m)||p(s|m)] - \mathbb{E}_{q(s|c,m)}[\log p(o|s, c, m)] \tag{7}$$

21.2 Message Passing Rules for Different Cases

These equations illustrate how case assignments modulate standard hierarchical message passing (e.g., in predictive coding) where beliefs/predictions (μ) and prediction errors (ε) flow between adjacent levels (denoted by superscripts 0 and 1). The case-specific weights (κ_c) determine the influence of each message type based on the model's current functional role.

Equations 8-12: Case-Specific Message Passing Rules

$$\text{Nominative [NOM]} : \mu^0 = \mu^0 + \kappa_{NOM} \cdot (\mu^1 - \mu^0) \tag{8}$$

(Lower-level prediction μ^0 updated by top-down prediction μ^1, weighted by κ_{NOM})

$$\text{Accusative [ACC]} : \varepsilon^1 = \varepsilon^1 + \kappa_{ACC} \cdot (\varepsilon^0 - \varepsilon^1) \tag{9}$$

(Higher-level error ε^1 updated by bottom-up error ε^0, weighted by κ_{ACC})

$$\text{Dative [DAT]} : \mu^0 = \mu^0 + \kappa_{DAT} \cdot (data - \mu^0) \tag{10}$$

(Lower-level prediction μ^0 updated directly by incoming 'data', weighted by κ_{DAT})

$$\text{Genitive [GEN]} : output = \mu^0 + \kappa_{GEN} \cdot \eta \tag{11}$$

(Output generated based on lower-level prediction μ^0, weighted by κ_{GEN}, potentially with noise η)

$$\text{Instrumental [INS]} : process = f(\mu^1, \varepsilon^0) \cdot \kappa_{INS} \tag{12}$$

(A process output determined by some function f of top-down prediction μ^1 and bottom-up error ε^0, weighted by κ_{INS})

$$\text{Vocative [VOC]} : activation = \sigma(\kappa_{VOC} \cdot sim(id, address)) \tag{12a}$$

(Activation state determined by similarity between model identity id and incoming address, weighted by κ_{VOC} and passed through activation function σ)

where κ_c represents case-specific learning rates or precision weights, η is a noise term, μ^0, μ^1 represent beliefs/predictions, and $\varepsilon^0, \varepsilon^1$ represent prediction errors at adjacent hierarchical levels.

21.3 Precision Allocation and Resource Optimization

Equation 13: Precision Weight Allocation with Temperature

$$\beta(c, m) = \frac{\exp(-\gamma \cdot F(c, m))}{\sum_i \exp(-\gamma \cdot F(c_i, m))} \tag{13}$$

where is the inverse temperature parameter controlling allocation sharpness.

Equation 14: Resource-Weighted Free Energy

$$F_\beta(m) = \sum_c \beta(c, m) \cdot F(c, m) \cdot R(c) \tag{14}$$

where R(c) represents the computational resources allocated to case c.

21.4 Novel Case Formalizations

Equation 15: Conjunctive Case Free Energy

$$F_{CNJ} = D_{KL}[q(s|CNJ,m)||p(s|m)] - \mathbb{E}_{q(s|CNJ,m)}[\log p(o|s,\{m_i\})] \tag{15}$$

where {m_i} represents the assembly of connected models.

Equation 16: Conjunctive Case Message Passing

$$\mu^{CNJ} = \sum_i w_i \cdot \mu_i + \kappa_{CNJ} \cdot (\prod_i \mu_i - \sum_i w_i \cdot \mu_i) \tag{16}$$

where w_i are model-specific weighting factors.

Equation 17: Recursive Case Precision Dynamics

$$\beta(REC,m) = \frac{\exp(-\gamma \cdot F(REC,m))}{\sum_i \exp(-\gamma \cdot F(c_i,m)) + \exp(-\gamma \cdot F(REC,m))} \tag{17}$$

21.4.1 Glossary of Variables

- a: Action (in MDP context, often selecting a case transition)
- α: Learning rate (in Neural Process Models context)
- BF: Bayes Factor (for comparing model evidence between cases)
- c, c_i, c', c_1, c_2: Linguistic case assignment (e.g., NOM, ACC, specific case instances)
- Case(M): Case assignment of model M
- **Case Transformation**: An operation that changes the functional role (case) of a model within the system
- **CEREBRUM**: Case-Enabled Reasoning Engine with Bayesian Representations for Unified Modeling
- D_{KL}: Kullback-Leibler divergence
- data: Input data (in Dative case message passing; Eq 10)
- **Declinability**: The capacity of a generative model within CEREBRUM to assume different morphological and functional roles (cases) through transformations
- $E_p[\cdot]$: Expectation with respect to distribution p (Information Geometry)
- $\mathbb{E}[\cdot]$: Expectation operator
- F: Variational Free Energy
- $F_\beta(m)$: Resource-weighted free energy for model m
- F_{CNJ}: Free energy for the speculative Conjunctive case
- $f(...)$: Function (used generally; e.g., in Instrumental message passing; Eq 12)
- g_{ij}: Fisher information metric tensor component (Information Geometry)

- i, j: Indices for summation or tensor components
- $L(M)$: Lyapunov function for model M (Dynamical Systems section)
- m, M: Cognitive model
- $\{m_i\}$: Assembly or set of connected models
- $\mathrm{MB}(M)$: Markov blanket of model M
- **Morphological Marker (Computational Analogue)**: Specific computational properties (e.g., active interfaces; parameter access patterns; update dynamics) that signal a model's current case assignment within CEREBRUM
- n: Model parameter count (Complexity section)
- $O(...)$: Big O notation for computational complexity
- o: Observations or sensory data
- output: Output generated by a model (in Genitive case; Eq 11)
- $p(s|...)$: Prior distribution over internal states s
- $p(o|...)$: Likelihood distribution of observations o
- $p(x|theta)$: Probability distribution of data x given parameters $theta$ (Information Geometry)
- process: Result of a process executed by a model (in Instrumental case; Eq 12)
- $q(s|...)$: Approximate posterior distribution over internal states s
- $R(c)$: Computational resources allocated to case c
- REC: Speculative Recursive case assignment
- s: Internal states of a model
- s': Next state (in MDP context; target case assignment)
- t: Time variable (in gradient descent context; Eq 4)
- T: Transformation function (e.g., $T(m)$ is a transformed model in Eq 1; also MDP transition function)
- $T(s'|s, a)$: State transition function in MDP (probability of transitioning to state s' from state s given action a)
- w_i: Model-specific weighting factors (in Conjunctive case; Eq 16)
- ΔF: Change in Free Energy
- Δw_{ij}: Change in synaptic weight between neuron i and j (Neural Process Models section)
- $\beta(c, m)$: Precision weight (allocation) assigned to model m in case c
- γ: Inverse temperature parameter (controlling precision allocation sharpness)
- ϵ_i: Error signal of neuron i (Neural Process Models section)
- $\varepsilon^0, \varepsilon^1$: Error signals used in message passing (representing prediction errors at adjacent hierarchical levels; Eq 9, 12)
- η: Noise term (Eq 11)
- κ_c: Case-specific learning rate or precision weight (modulating message updates; Eqs 4, 8-12)
- μ^0, μ^1: Mean values used in message passing (representing predictions or beliefs at adjacent hierarchical levels)
- μ^{CNJ}: Mean value resulting from Conjunctive case message passing
- $\pi(a|s)$: Policy in MDP (probability of taking action a in state s)
- $\sigma'(a_j)$: Derivative of activation function of neuron j (Neural Process Models section)

- $theta, theta_i, theta_j$: Model parameters # Appendix B 2: Novel Linguistic Cases

22 Discovering and Creating New Linguistic Cases Through CEREBRUM

The CEREBRUM framework not only operationalizes traditional linguistic cases but potentially enables the discovery of entirely new case archetypes through its systematic approach to model interactions. As cognitive models interact in increasingly complex ecosystems, emergent functional roles may arise that transcend the classical case system derived from human languages.

22.1 Emergence of Novel Case Functions

Traditional linguistic case systems evolved to serve human communication needs in physical and social environments. However, computational cognitive ecosystems face novel challenges and opportunities that may drive the emergence of new functional roles. The mathematical formalism of CEREBRUM provides a scaffold for identifying these emergent case functions through:

1. **Pattern detection in model interaction graphs**: Recurring patterns of information flow that don't fit established cases
2. **Free energy anomalies**: Unusual optimization patterns indicating novel functional configurations
3. **Precision allocation clusters**: Statistical clustering of precision weightings revealing new functional categories
4. **Transition probability densities**: Dense regions in case transition probability spaces suggesting stable new cases

22.2 Speculative Novel Case: The Emergent "Conjunctive" Case

One speculative example of a novel case that might emerge within CEREBRUM is what we might term the "conjunctive" case [CNJ]. This case would represent a model's role in synthesizing multiple predictive streams into coherent joint predictions that couldn't be achieved through simple composition of existing cases.

The mathematical formalism for a model in conjunctive case would extend the standard free energy equation as shown in Equation 15 (see Mathematical Appendix), representing the assembly of connected models participating in the joint prediction. The key innovation is that the likelihood term explicitly depends on multiple models' predictions rather than a single model's output, enabling integration of diverse predictive streams.

In the message-passing formulation, the conjunctive case would introduce unique update rules as described in Equation 16 (see Mathematical Appendix), with weighting factors for individual model predictions, as well as a multiplicative integration of predictions that captures interdependencies beyond simple weighted averaging. This formulation enables rich joint inference across model collectives.

22.3 Speculative Novel Case: The "Recursive" Case

Another potential novel case is the "recursive" case [REC], which would enable a model to apply its transformations to itself, creating a form of computational reflection not captured by traditional cases.

In the recursive case, a model assumes both agent and object roles simultaneously, creating feedback loops that enable complex self-modification behaviors. This case would be particularly relevant for metalearning systems and artificial neural networks that modify their own architectures.

The recursive case would introduce unique precision dynamics as formalized in Equation 17 (see Mathematical Appendix). The key innovation is that the model appears on both sides of the transformation, creating a form of self-reference that traditional case systems don't accommodate. This enables models to introspect and modify their own parameters through self-directed transformations.

22.4 Speculative Novel Case: The "Metaphorical" Case

A third potential novel case is the "metaphorical" case [MET], which would enable a model to map structures and relationships from one domain to another, creating computational analogies that transfer knowledge across conceptual spaces.

In the metaphorical case, a model acts as a transformation bridge between disparate domains, establishing systematic mappings between conceptual structures. This case would be particularly valuable for transfer learning systems and creative problem-solving algorithms that need to apply learned patterns in novel contexts.

The metaphorical case would introduce unique cross-domain mapping functions as formalized in Equation 18 (see Mathematical Appendix). The key innovation is the structured alignment of latent representations across domains, enabling principled knowledge transfer that preserves relational invariants while adapting to target domain constraints.

22.4.1 Connections to Human Cognition and Communication

The metaphorical case has rich connections to multiple domains of human cognition and communication. In affective neuroscience, it models how emotional experiences are mapped onto conceptual frameworks, explaining how we understand emotions through bodily metaphors (e.g., "heavy heart," "burning anger"). In first and second-person neuroscience, metaphorical mappings enable perspective-taking and empathy through systematic projection of one's own experiential models onto others. Educational contexts leverage metaphorical case operations when complex concepts are taught through familiar analogies, making abstract ideas concrete through structured mappings. The way people converse about generative models often employs metaphorical language—describing models as "thinking," "imagining," or "dreaming"—which represents a natural metaphorical mapping between human cognitive processes and computational operations. Learning itself fundamentally involves metaphorical operations when knowledge from one domain scaffolds understanding in another. Perhaps most profoundly, the metaphorical case provides a computational framework for understanding how symbols and archetypes function in human cognition—as cross-domain mappings that compress complex experiential patterns into transferable, culturally-shared representations that retain their structural integrity across diverse contexts while adapting to individual interpretive frameworks.

22.5 Implications of Novel Cases for Computational Cognition

The discovery of novel cases through CEREBRUM could have profound implications for computational cognitive science:

1. **Expanded representational capacity**: New cases enable representation of functional relationships beyond traditional linguistic frameworks
2. **Enhanced model compositionality**: Novel cases might enable more efficient composition of complex model assemblies
3. **Computational reflection**: Cases like the recursive case enable systematic implementation of self-modifying systems
4. **Cross-domain integration**: New cases like the metaphorical case might bridge domains that are difficult to connect with traditional case systems

These speculative extensions of CEREBRUM highlight its potential not just as an implementation of linguistic ideas in computational contexts, but as a framework that could expand our understanding of functional roles beyond traditional linguistic categories. The mathematical rigor of CEREBRUM provides a foundation for systematically exploring this expanded space of possible case functions, potentially leading to entirely new paradigms for understanding complex model interactions in cognitive systems.

Table A1: Properties of Speculative Novel Cases in CEREBRUM

Property	Conjunctive Case [CNJ]	Recursive Case [REC]	Metaphorical Case [MET]
Function	Synthesizes multiple predictive streams into coherent joint predictions; integrates diverse model outputs; resolves cross-model inconsistencies	Applies transformations to itself; enables self-modification; creates meta-level processing loops	Maps structures and relationships between domains; establishes cross-domain correspondences; transfers knowledge patterns across conceptual spaces
Parametric Focus	Cross-model correlation parameters and shared latent variables; inter-model weights; joint distribution parameters	Self-referential parameters; recursive transformations; meta-parameters governing self-modification	Structural alignment parameters; analogical mapping weights; cross-domain correspondence metrics

Property	Conjunctive Case [CNJ]	Recursive Case [REC]	Metaphorical Case [MET]
Precision Weighting	Highest precision on inter-model consistency and joint predictions; emphasizes mutual information; optimizes integration factors	Dynamic self-allocation; recursive precision assignment; meta-precision governing self-modification	Selective precision on structural invariants; emphasis on relational similarities over surface features; adaptive mapping precision
Interface Type	Aggregative interfaces with multiple connected models; convergent communication channels; integration hubs	Reflexive interfaces; self-directed connections; loopback channels	Bridging interfaces across domain boundaries; cross-contextual mappings; translation channels
Update Dynamics	Updates based on joint prediction errors across the connected model assembly; collective error minimization; consistency optimization	Self-modification loops; introspective learning; meta-learning through internal feedback	Updates based on structural alignment success; transfer performance feedback; analogical coherence optimization

Appendix C 3: Practical Applications of Model Declension in Cognitive Ecosystems

The declension paradigm for cognitive models offers practical benefits in complex model ecosystems spanning multiple cognitive domains. This appendix outlines specific applications where the morphological adaptability of models provides significant advantages and describes technical implementations.

23 Model Pipeline Optimization

Complex cognitive workflows typically involve sequences of models arranged in processing pipelines. Traditional approaches require specialized interface layers between models, leading to inefficiencies and compatibility challenges. By applying case declensions to models in these pipelines, each component can seamlessly adapt its interfaces:

Consider a pipeline where Model exhibits a case transition from [ACC] (receiving data) to [DAT] (forwarding results), demonstrating how a single model can adapt its functional interfaces based on its position in the processing sequence.

23.1 Technical Implementation: Pipeline Adapter Patterns

```
class CaseTransformer:
    """Implements case transformation between pipeline stages"""
```

```python
    def transform(self, model, source_case, target_case):
        """Transform model from source_case to target_case"""
        if source_case == "ACC" and target_case == "DAT":
            # Reconfigure model parameter access
            model.input_gates = True
            model.output_gates = True
            # Adjust precision weighting
            model.precision_weights = {"inputs": 0.8, "outputs": 0.7}
            # Update interface specifications
            model.interfaces = {
                "input": model.default_interfaces["input"],
                "output": model.default_interfaces["forward"]
            }
        return model

class ModelPipeline:
    """Pipeline of case-bearing models"""

    def __init__(self, models, case_transformer):
        self.models = models
        self.transformer = case_transformer

    def optimize(self):
        """Optimize pipeline by assigning appropriate cases to models"""
        # First model is typically in NOM case (generating)
        self.models[0].case = "NOM"

        # Middle models often transition between ACC and DAT
        for i in range(1, len(self.models)-1):
            # When receiving: ACC case
            self.models[i].case = "ACC"
            # Process data
            self.models[i].process()
            # When forwarding: DAT case
            self.transformer.transform(self.models[i], "ACC", "DAT")

        # Final model often in GEN case (producing output)
        self.models[-1].case = "GEN"
```

24 Computational Resource Optimization

In resource-constrained environments, the precision allocation mechanism provided by case declension enables dynamic distribution of computational resources:

Table 5: Resource Allocation Strategy by Cognitive Task Type

Use Case	Resource Strategy	Case Priority	Optimization Objective
Real-time decision making	Prioritize prediction generation; allocate resources to forward inference; minimize predictive latency	[NOM] > [DAT] > [ACC] > others	Minimize latency; maximize predictive accuracy; optimize decision boundaries
Data ingestion and processing	Prioritize input handling; allocate resources to perceptual categorization; maximize throughput	[DAT] > [ACC] > [GEN] > others	Maximize throughput; optimize filter efficiency; minimize information loss
Report generation	Prioritize output production; allocate resources to synthesis; optimize presentation clarity	[GEN] > [NOM] > [LOC] > others	Optimize fidelity; maximize clarity; ensure appropriate detail level
Method development	Prioritize process refinement; allocate resources to algorithm optimization; focus on error reduction	[INS] > [ACC] > [NOM] > others	Minimize error; improve algorithmic efficiency; enhance procedural robustness

This dynamic resource allocation is formalized through the precision-weighted free energy equation (Equation 14 in the Mathematical Appendix), where models are allocated computational resources proportional to their precision weights for their current case assignment.

24.1 Technical Implementation: Resource Allocation Manager

```
import numpy as np

class ResourceAllocationManager:
    """Manages computational resources across case-bearing models"""

    def __init__(self, total_compute, total_memory):
        self.total_compute = total_compute  # e.g., CPU cores or cycles
```

```python
        self.total_memory = total_memory    # e.g., RAM allocation
        self.case_priorities = {
            "real_time_decision": {"NOM": 0.5, "DAT": 0.3, "ACC": 0.2},
            "data_ingestion": {"DAT": 0.5, "ACC": 0.3, "GEN": 0.2},
            "report_generation": {"GEN": 0.5, "NOM": 0.3, "LOC": 0.2},
            "method_development": {"INS": 0.5, "ACC": 0.3, "NOM": 0.2}
        }

    def allocate_resources(self, models, task_type):
        """Allocate computational resources based on case priorities for task"""
        # Get priorities for this task type
        priorities = self.case_priorities.get(
            task_type,
            {"NOM": 0.25, "ACC": 0.25, "DAT": 0.25, "GEN": 0.25}  # Default uniform
        )

        # Count models by case
        case_counts = {}
        for model in models:
            case_counts[model.case] = case_counts.get(model.case, 0) + 1

        # Calculate base allocations (proportional to priority)
        total_priority = sum(
            priorities.get(model.case, 0.1) for model in models
        )

        # Assign compute and memory
        for model in models:
            # Get priority or default low priority if not specified
            priority = priorities.get(model.case, 0.1)

            # Assign proportional resources
            model.compute_allocation = (priority / total_priority) * self.total_compute
            model.memory_allocation = (priority / total_priority) * self.total_memory

            # Apply precision-weighting based on free energy equation
            if hasattr(model, 'precision'):
                # Scale allocation by model precision
                precision_factor = np.tanh(model.precision)  # Bounded scaling
                model.compute_allocation *= precision_factor
                model.memory_allocation *= precision_factor
```

```python
        return models
```

25 Model Ecosystem Adaptability

Cognitive ecosystems must adapt to changing environments and requirements. The declension paradigm enables flexible reconfiguration of model relationships without architectural redesign.

Conceptually, this means the same set of models can reconfigure their functional roles through case reassignment, adapting to new requirements without changing the underlying model implementations.

25.1 Technical Implementation: Dynamic Case Assignment

```python
class AdaptiveModelEcosystem:
    """An ecosystem of models that adapts to changing requirements"""

    def __init__(self, models, transformer):
        self.models = models
        self.transformer = transformer
        self.current_configuration = "default"

    def reconfigure(self, new_configuration):
        """Reconfigure the ecosystem for a new operational context"""
        # Configuration specifications
        configurations = {
            "data_intensive": {
                "processor_models": ["DAT", "DAT", "ACC"],
                "reasoner_models": ["LOC", "INS", "NOM"],
                "output_models": ["GEN", "VOC"]
            },
            "decision_intensive": {
                "processor_models": ["ACC", "NOM", "NOM"],
                "reasoner_models": ["INS", "NOM", "LOC"],
                "output_models": ["NOM", "VOC"]
            },
            "explanation_intensive": {
                "processor_models": ["ACC", "ABL", "LOC"],
                "reasoner_models": ["LOC", "INS", "ABL"],
                "output_models": ["GEN", "VOC"]
            }
        }
```

```python
# Get target configuration or use default
target_config = configurations.get(
    new_configuration,
    {"processor_models": ["ACC"], "reasoner_models": ["NOM"], "output_models": ["GEN"]}
)

# Apply configuration
model_groups = {
    "processor_models": [m for m in self.models if m.type == "processor"],
    "reasoner_models": [m for m in self.models if m.type == "reasoner"],
    "output_models": [m for m in self.models if m.type == "output"]
}

# Apply case assignments to each group
for group_name, cases in target_config.items():
    models = model_groups.get(group_name, [])
    for i, model in enumerate(models):
        if i < len(cases):
            # Transform model to new case
            self.transformer.transform(model, model.case, cases[i])
            model.case = cases[i]

# Update current configuration
self.current_configuration = new_configuration
return True
```

26 Cross-Domain Integration

The CEREBRUM framework facilitates integration between disparate cognitive domains by providing a unified grammatical structure for model interactions:

Table 6: Cross-Domain Integration Patterns in CEREBRUM Framework

Domain	Primary Cases	Integration Pattern	Error Propagation
Perception	[NOM] (senses), [ACC] (percepts)	Sensory models [NOM] → Perceptual models [ACC]; hierarchical feature extraction; predictive sensing	Bottom-up; prediction errors flow from sensors to percepts; precision-weighted by sensory reliability

Domain	Primary Cases	Integration Pattern	Error Propagation
Reasoning	[INS] (logic), [LOC] (context)	Logical models [INS] → Contextual models [LOC]; context-sensitive inference; situational logic	Bidirectional; coherence errors propagate between logical rules and contextual constraints; mutual constraints
Planning	[GEN] (goals), [ABL] (history)	Historical models [ABL] → Goal models [GEN]; experience-informed planning; trajectory optimization	Top-down; goal-directed errors influence historical interpretation; teleological constraints
Action	[DAT] (commands), [NOM] (execution)	Command models [DAT] → Execution models [NOM]; imperative processing; motor control	Circular; execution errors feed back to command refinement; continuous adjustment loop

By mapping these domain-specific interactions to standardized case relationships, previously incompatible models can be integrated into cohesive cognitive systems.

26.1 Technical Implementation: Cross-Domain Integration Interface

```
class CrossDomainIntegrator:
    """Integrates models from different cognitive domains"""

    def __init__(self):
        self.domain_patterns = {
            "perception": {
                "primary_cases": ["NOM", "ACC"],
                "error_flow": "bottom_up",
                "precision_modulation": "sensory_reliability"
            },
            "reasoning": {
                "primary_cases": ["INS", "LOC"],
                "error_flow": "bidirectional",
                "precision_modulation": "coherence"
            },
            "planning": {
                "primary_cases": ["ABL", "GEN"],
                "error_flow": "top_down",
                "precision_modulation": "goal_alignment"
            },
```

```
        "action": {
            "primary_cases": ["DAT", "NOM"],
            "error_flow": "circular",
            "precision_modulation": "execution_efficacy"
        }
    }

def connect_domains(self, source_model, target_model):
    """Connect models from different domains using appropriate case alignment"""
    source_domain = source_model.domain
    target_domain = target_model.domain

    # Get domain patterns
    source_pattern = self.domain_patterns.get(source_domain)
    target_pattern = self.domain_patterns.get(target_domain)

    if not source_pattern or not target_pattern:
        return False  # Unknown domain

    # Determine compatible cases for connection
    connection_map = {
        # From perception to reasoning
        ("perception", "reasoning"): {
            "source_case": "ACC",  # Perceptual output
            "target_case": "LOC",  # Contextual input for reasoning
            "message_format": "feature_vector",
            "error_propagation": "weighted_bottom_up"
        },
        # From reasoning to planning
        ("reasoning", "planning"): {
            "source_case": "INS",  # Reasoning method
            "target_case": "GEN",  # Goal generation
            "message_format": "constraint_set",
            "error_propagation": "bidirectional"
        },
        # From planning to action
        ("planning", "action"): {
            "source_case": "GEN",  # Goal production
            "target_case": "DAT",  # Command reception
            "message_format": "action_sequence",
            "error_propagation": "top_down"
```

```python
    },
    # From action to perception (closing the loop)
    ("action", "perception"): {
        "source_case": "NOM",  # Action execution
        "target_case": "NOM",  # Sensory prediction
        "message_format": "predicted_sensation",
        "error_propagation": "circular"
    }
}

# Get connection specification
connection_key = (source_domain, target_domain)
connection_spec = connection_map.get(connection_key)

if not connection_spec:
    # Try reverse connection with adjusted cases
    connection_key = (target_domain, source_domain)
    connection_spec = connection_map.get(connection_key)
    if connection_spec:
        # Swap source and target specifications
        connection_spec = {
            "source_case": connection_spec["target_case"],
            "target_case": connection_spec["source_case"],
            "message_format": connection_spec["message_format"],
            "error_propagation": self._reverse_error_flow(
                connection_spec["error_propagation"]
            )
        }

if not connection_spec:
    # If still no direct match, use default connection
    connection_spec = {
        "source_case": source_pattern["primary_cases"][1],  # Output case
        "target_case": target_pattern["primary_cases"][0],  # Input case
        "message_format": "generic",
        "error_propagation": "minimal"
    }

# Configure the connection
return self._establish_connection(
    source_model, target_model, connection_spec
```

```
    )

def _establish_connection(self, source_model, target_model, spec):
    """Establish actual connection between models"""
    # Set up message passing interface
    source_model.add_output_connection(
        target_model.id,
        case=spec["source_case"],
        format=spec["message_format"]
    )

    target_model.add_input_connection(
        source_model.id,
        case=spec["target_case"],
        format=spec["message_format"]
    )

    # Configure error propagation
    if spec["error_propagation"] == "weighted_bottom_up":
        target_model.add_error_callback(
            lambda err: source_model.update_with_error(err * source_model.precision)
        )
    elif spec["error_propagation"] == "top_down":
        source_model.add_error_callback(
            lambda err: target_model.update_with_error(err)
        )
    elif spec["error_propagation"] == "bidirectional":
        # Both models get each other's errors
        source_model.add_error_callback(
            lambda err: target_model.update_with_error(err * 0.5)
        )
        target_model.add_error_callback(
            lambda err: source_model.update_with_error(err * 0.5)
        )
    elif spec["error_propagation"] == "circular":
        # Circular error propagation for sensorimotor loops
        source_model.add_error_callback(
            lambda err: target_model.add_prediction_error(err)
        )
        target_model.add_error_callback(
            lambda err: source_model.add_sensory_error(err)
```

```python
        )

        return True

    def _reverse_error_flow(self, flow_type):
        """Reverse the direction of error flow"""
        flow_map = {
            "weighted_bottom_up": "top_down",
            "top_down": "weighted_bottom_up",
            "bidirectional": "bidirectional",
            "circular": "circular",
            "minimal": "minimal"
        }
        return flow_map.get(flow_type, "minimal")
```

27 Knowledge Graph Enhancement

The case declension system enhances knowledge representation by providing richer relational semantics in model-based knowledge graphs. This enhancement operates at multiple levels:

1. **Semantic Role Labeling**:
    - Models in [NOM] case represent active knowledge producers
    - Models in [ACC] case represent knowledge targets/recipients
    - Models in [DAT] case represent knowledge transfer endpoints
    - Models in [GEN] case represent knowledge sources/origins
    - Models in [INS] case represent methodological knowledge
    - Models in [LOC] case represent contextual knowledge
    - Models in [ABL] case represent historical/causal knowledge
2. **Relationship Typing**:
    - Morphosyntactic edges encode relationship types
    - Case assignments provide edge directionality
    - Case transitions represent knowledge flow patterns
    - Multi-case paths represent complex knowledge transformations
3. **Example Knowledge Propagation Rules**:
    - Case-preserving transformations maintain semantic roles
    - Case-changing transformations represent functional shifts
    - Case alignment patterns guide knowledge integration
    - Case-based precision weighting prioritizes knowledge flow (see Equation 13 in Mathematical Appendix)

This enhanced knowledge graph shows how case-declined models provide explicit relationship semantics between entities, creating richer knowledge representations that mirror the way natural language encodes semantic relationships through case systems.

27.1 Technical Implementation: Case-Based Knowledge Graph Schema

```python
import networkx as nx

class CerebrumKnowledgeGraph:
    """Knowledge graph with case-semantic relationships"""

    def __init__(self):
        self.graph = nx.MultiDiGraph()
        self.case_mappings = {
            "NOM": {"relation_type": "produces", "inverse": "produced_by"},
            "ACC": {"relation_type": "targets", "inverse": "targeted_by"},
            "DAT": {"relation_type": "receives", "inverse": "received_by"},
            "GEN": {"relation_type": "sources", "inverse": "sourced_from"},
            "INS": {"relation_type": "implements", "inverse": "implemented_by"},
            "LOC": {"relation_type": "contextualizes", "inverse": "contextualized_by"},
            "ABL": {"relation_type": "originates", "inverse": "originated_from"},
            "VOC": {"relation_type": "addresses", "inverse": "addressed_by"}
        }

    def add_case_relationship(self, source_entity, target_entity, case):
        """Add relationship based on case semantics"""
        if case not in self.case_mappings:
            raise ValueError(f"Unknown case: {case}")

        relation = self.case_mappings[case]["relation_type"]
        inverse = self.case_mappings[case]["inverse"]

        # Add the case-based relationship
        self.graph.add_edge(
            source_entity,
            target_entity,
            relation=relation,
            case=case,
            weight=1.0
        )

        # Add the inverse relationship (optional)
        self.graph.add_edge(
            target_entity,
            source_entity,
            relation=inverse,
```

```
            case="inverse_" + case,
            weight=0.5  # Inverse relationships generally weighted lower
        )

    def propagate_knowledge(self, source_entity, relation_path, weight_decay=0.85):
        """Propagate knowledge along a case-based path"""
        current_nodes = [(source_entity, 1.0)]  # (node, weight)
        visited = set()

        for case in relation_path:
            relation = self.case_mappings.get(case, {}).get("relation_type")
            if not relation:
                continue

            next_nodes = []
            for node, weight in current_nodes:
                if node in visited:
                    continue

                visited.add(node)
                for _, target, data in self.graph.out_edges(node, data=True):
                    if data.get("relation") == relation:
                        # Propagate with decaying weight
                        next_nodes.append((target, weight * weight_decay))

            current_nodes = next_nodes
            if not current_nodes:
                break

        # Return final nodes with propagated weights
        return current_nodes

    def analyze_connectivity(self):
        """Analyze connectivity patterns in the knowledge graph"""
        # Get frequency of each case type
        case_counts = {}
        for _, _, data in self.graph.edges(data=True):
            case = data.get("case", "unknown")
            case_counts[case] = case_counts.get(case, 0) + 1

        # Calculate centrality measures for entities
```

```python
        centrality = nx.degree_centrality(self.graph)

        # Identify dominant case patterns
        dominant_patterns = []
        for node in self.graph.nodes():
            node_cases = {}
            for _, _, data in self.graph.out_edges(node, data=True):
                case = data.get("case", "unknown")
                node_cases[case] = node_cases.get(case, 0) + 1

            # Get dominant case (most frequent)
            if node_cases:
                dominant_case = max(node_cases.items(), key=lambda x: x[1])[0]
                dominant_patterns.append((node, dominant_case))

        return {
            "case_distribution": case_counts,
            "entity_centrality": centrality,
            "dominant_patterns": dominant_patterns
        }
```

28 Emergent Behaviors in Model Collectives

When multiple case-bearing models interact within an ecosystem, emergent collective behaviors arise from their case-driven interactions, analogous to how linguistic communities develop shared understanding through dialog:

1. **Self-organizing workflows**:
    - Models dynamically form processing chains based on complementary case assignments
    - Like speakers in dialogue naturally assuming complementary roles (questioner/answerer)
    - Case alignment creates natural processing pipelines
    - Processing chains form spontaneously through case compatibility
2. **Adaptive resource allocation**:
    - Precision-weighted competition for computational resources drives efficient task distribution
    - Similar to attention allocation in linguistic communities
    - Resources are allocated based on case-specific precision weights (see Equation 13 in the Mathematical Appendix)
    - Dynamic reallocation follows free energy gradients
3. **Collective learning**:
    - Error signals propagate through case relationships
    - Like linguistic communities converging on shared meanings

- Learning rates are modulated by case compatibility
- System-wide adaptation through message passing (see Equations 8-12 in the Mathematical Appendix)

4. **Fault tolerance**:
   - Models can adopt alternative cases when certain cognitive functions are degraded
   - Similar to linguistic communities adapting to speaker limitations
   - Case reassignment follows free energy minimization
   - Graceful degradation through case flexibility
5. **Semantic Consensus Formation**:
   - Models converge on shared representations through case-mediated interactions
   - Parallels linguistic communities developing shared vocabularies
   - Consensus emerges through case-specific alignment
   - Alignment strength varies by case type
6. **Hierarchical Organization**:
   - Case relationships naturally create processing hierarchies
   - Like linguistic communities developing formal/informal speech levels
   - Hierarchy levels emerge from case distributions
   - Case assignments reflect hierarchical position

These emergent properties demonstrate how the declension paradigm enables robust, adaptive collective behaviors in complex cognitive ecosystems, mirroring the way linguistic communities develop and maintain shared understanding through structured interactions. The mathematical formalization of these properties provides a rigorous foundation for analyzing and optimizing model collective behavior.

28.1 Technical Implementation: Model Collective Simulator

```
import numpy as np
import networkx as nx
from collections import defaultdict

class ModelCollectiveSimulator:
    """Simulates emergent behaviors in collectives of case-bearing models"""

    def __init__(self, num_models=10, case_types=["NOM", "ACC", "DAT", "GEN", "INS", "LOC", "A
        self.models = []
        self.case_types = case_types
        self.compatibility_matrix = self._generate_compatibility_matrix()

        # Initialize models with random cases
        for i in range(num_models):
            self.models.append({
                "id": i,
```

```python
            "case": np.random.choice(case_types),
            "precision": np.random.uniform(0.5, 1.0),
            "connections": [],
            "resources": 1.0,
            "learning_rate": np.random.uniform(0.01, 0.1),
            "representation": np.random.random(10)  # Simple vector representation
        })

    # Initialize network
    self.network = self._initialize_network()

def _generate_compatibility_matrix(self):
    """Generate matrix of case compatibility scores"""
    num_cases = len(self.case_types)
    matrix = np.zeros((num_cases, num_cases))

    # Define natural complementary relationships
    # Higher values indicate stronger compatibility
    complementary_pairs = {
        ("NOM", "ACC"): 0.9,  # Subject-Object
        ("GEN", "DAT"): 0.8,  # Source-Recipient
        ("INS", "LOC"): 0.7,  # Method-Context
        ("ABL", "NOM"): 0.6,  # Origin-Subject
        ("NOM", "VOC"): 0.5,  # Caller-Called
        ("DAT", "ACC"): 0.5,  # Recipient-Object
        ("GEN", "INS"): 0.4,  # Source-Method
        ("LOC", "ABL"): 0.4   # Context-Origin
    }

    # Fill compatibility matrix
    for i, case1 in enumerate(self.case_types):
        for j, case2 in enumerate(self.case_types):
            # Check both directions
            score = complementary_pairs.get((case1, case2), 0.1)
            score2 = complementary_pairs.get((case2, case1), 0.1)
            matrix[i, j] = max(score, score2)

    # Ensure diagonal has moderate compatibility with self
    np.fill_diagonal(matrix, 0.3)

    return matrix
```

```python
def _initialize_network(self):
    """Initialize network of model connections based on case compatibility"""
    G = nx.Graph()

    # Add all models as nodes
    for model in self.models:
        G.add_node(model["id"], case=model["case"], precision=model["precision"])

    # Add edges based on case compatibility
    for i, model1 in enumerate(self.models):
        case1_idx = self.case_types.index(model1["case"])

        for j, model2 in enumerate(self.models):
            if i == j:
                continue

            case2_idx = self.case_types.index(model2["case"])
            compatibility = self.compatibility_matrix[case1_idx, case2_idx]

            # Only connect if compatibility exceeds threshold
            if compatibility > 0.3:
                G.add_edge(
                    model1["id"],
                    model2["id"],
                    weight=compatibility,
                    type="case_relation"
                )

                # Update model connections
                model1["connections"].append(model2["id"])

    return G

def simulate_self_organization(self, steps=10):
    """Simulate self-organization of model workflows"""
    results = []

    for step in range(steps):
        # Track changes
        changes = 0
```

```python
# Models assess their local environment and adapt
for model in self.models:
    # Get current case and neighbors
    current_case = model["case"]
    neighbor_ids = list(self.network.neighbors(model["id"]))

    if not neighbor_ids:
        continue

    # Analyze neighbor cases
    neighbor_cases = []
    for nid in neighbor_ids:
        neighbor = self.models[nid]
        neighbor_cases.append(neighbor["case"])

    # Calculate optimal case based on neighbors
    optimal_case = self._determine_optimal_case(current_case, neighbor_cases)

    # Change case if improvement exceeds threshold
    if optimal_case != current_case:
        model["case"] = optimal_case
        changes += 1

        # Update network
        self.network.nodes[model["id"]]["case"] = optimal_case

# Update connections based on new case assignments
self._update_connections()

# Record state
case_distribution = self._get_case_distribution()
workflow_chains = self._identify_workflow_chains()

results.append({
    "step": step,
    "changes": changes,
    "case_distribution": case_distribution,
    "workflow_chains": workflow_chains
})
```

```
            # Check for stability
            if changes == 0:
                break

        return results

    def _determine_optimal_case(self, current_case, neighbor_cases):
        """Determine optimal case assignment based on neighborhood"""
        current_idx = self.case_types.index(current_case)

        # Calculate compatibility with each potential case
        compatibility_scores = []

        for potential_case_idx, potential_case in enumerate(self.case_types):
            score = 0
            for neighbor_case in neighbor_cases:
                neighbor_idx = self.case_types.index(neighbor_case)
                score += self.compatibility_matrix[potential_case_idx, neighbor_idx]

            # Slightly favor current case to prevent oscillation
            if potential_case_idx == current_idx:
                score *= 1.1

            compatibility_scores.append((potential_case, score))

        # Return case with highest compatibility
        return max(compatibility_scores, key=lambda x: x[1])[0]

    def _update_connections(self):
        """Update network connections based on current case assignments"""
        # Clear existing connections
        self.network.clear_edges()

        # Rebuild connections based on updated case compatibility
        for i, model1 in enumerate(self.models):
            model1["connections"] = []  # Reset connections
            case1_idx = self.case_types.index(model1["case"])

            for j, model2 in enumerate(self.models):
                if i == j:
                    continue
```

```python
            case2_idx = self.case_types.index(model2["case"])
            compatibility = self.compatibility_matrix[case1_idx, case2_idx]

            # Only connect if compatibility exceeds threshold
            if compatibility > 0.3:
                self.network.add_edge(
                    model1["id"],
                    model2["id"],
                    weight=compatibility,
                    type="case_relation"
                )

                # Update model connections
                model1["connections"].append(model2["id"])

    def _get_case_distribution(self):
        """Get current distribution of cases"""
        distribution = {case: 0 for case in self.case_types}

        for model in self.models:
            distribution[model["case"]] += 1

        return distribution

    def _identify_workflow_chains(self):
        """Identify emergent workflow chains in the network"""
        # Create directed graph based on case relationships
        directed_graph = nx.DiGraph()

        for model in self.models:
            directed_graph.add_node(model["id"], case=model["case"])

        # Add directed edges based on typical workflow patterns
        workflow_patterns = [
            ("NOM", "ACC"),  # Producer -> Consumer
            ("ACC", "DAT"),  # Consumer -> Recipient
            ("DAT", "GEN"),  # Recipient -> Generator
            ("GEN", "INS"),  # Generator -> Method
            ("INS", "LOC"),  # Method -> Context
            ("LOC", "ABL"),  # Context -> Origin
```

```python
            ("ABL", "NOM")    # Origin -> Producer (completing cycle)
        ]

        # Add edges following workflow patterns
        for model1 in self.models:
            for model2 in self.models:
                if model1["id"] == model2["id"]:
                    continue

                # Check if they form a workflow pattern
                if (model1["case"], model2["case"]) in workflow_patterns:
                    # Check if they're connected in the undirected graph
                    if model2["id"] in model1["connections"]:
                        directed_graph.add_edge(model1["id"], model2["id"])

        # Find all simple paths of length >= 3
        paths = []
        for source in directed_graph.nodes():
            for target in directed_graph.nodes():
                if source != target:
                    for path in nx.all_simple_paths(directed_graph, source, target, cutoff=5):
                        if len(path) >= 3:
                            case_path = [self.models[node_id]["case"] for node_id in path]
                            paths.append({"path": path, "cases": case_path})

        return paths

    def simulate_consensus_formation(self, steps=20):
        """Simulate consensus formation in model representations"""
        history = []

        # Initial consensus measure
        consensus = self._measure_consensus()
        history.append({"step": 0, "consensus": consensus})

        for step in range(1, steps+1):
            # Each model updates its representation based on neighbors
            for model in self.models:
                if not model["connections"]:
                    continue
```

```
            # Get representations from connected models
            connected_reps = []
            for conn_id in model["connections"]:
                conn_model = self.models[conn_id]
                # Weight by case compatibility
                model1_case_idx = self.case_types.index(model["case"])
                model2_case_idx = self.case_types.index(conn_model["case"])
                weight = self.compatibility_matrix[model1_case_idx, model2_case_idx]

                connected_reps.append((conn_model["representation"], weight))

            # Update representation by moving toward weighted average
            if connected_reps:
                # Calculate weighted average
                total_weight = sum(w for _, w in connected_reps)
                avg_rep = np.zeros_like(model["representation"])

                for rep, weight in connected_reps:
                    avg_rep += rep * (weight / total_weight)

                # Move toward average based on learning rate
                model["representation"] = (
                    (1 - model["learning_rate"]) * model["representation"] +
                    model["learning_rate"] * avg_rep
                )

        # Measure consensus after updates
        consensus = self._measure_consensus()
        history.append({"step": step, "consensus": consensus})

    return history

def _measure_consensus(self):
    """Measure degree of consensus in representations"""
    if not self.models:
        return 0

    # Calculate average representation
    avg_rep = np.mean([m["representation"] for m in self.models], axis=0)

    # Calculate average distance from this consensus
```

```python
        distances = []
        for model in self.models:
            dist = np.linalg.norm(model["representation"] - avg_rep)
            distances.append(dist)

        # Normalize and invert so higher values mean more consensus
        if not distances:
            return 1.0

        avg_distance = np.mean(distances)
        if avg_distance == 0:
            return 1.0

        # Map to [0,1] where 1 means perfect consensus
        consensus = 1.0 / (1.0 + avg_distance)
        return consensus
```

The parallel between model collectives and linguistic communities extends to:

1. **Information Flow Patterns**:
   - Case-based routing: Messages flow according to case compatibility
   - Community structure: Models cluster by case affinity
   - Flow efficiency depends on case-specific precision weights
2. **Adaptation Mechanisms**:
   - Local adjustments: Models modify case assignments based on neighbors
   - Global optimization: System-wide free energy minimization (see Equation 1 in the Mathematical Appendix)
   - Adaptation rates follow temporal decay patterns
3. **Stability Properties**:
   - Case equilibrium: Stable distributions of case assignments
   - Dynamic resilience: Recovery from perturbations
   - Stability emerges from case distribution entropy

This framework provides a formal basis for understanding how collections of case-bearing models can develop sophisticated collective behaviors analogous to linguistic communities, while maintaining mathematical rigor through precise formalization of the underlying mechanisms.