CTGV System V1.3: Formal Specification

Aligned with Python Implementation

Begnomar dos Santos Porto
ORCID: 0009-0002-6109-7443

January 21, 2026
Implementation Version: V1.3

Abstract

This document provides a precise mathematical formalism for the CTGV (Versatile Geometric Topological Computing) system, directly mapped to the Python implementation V1.3. Every equation corresponds to specific code segments, ensuring complete alignment between theory and implementation. The formalism explicitly describes shape-dependent rules, field interference mechanisms, and the Temporal Binding Arbiter (TBA) as implemented in the codebase.

1 System Overview

The CTGV system is implemented as a **Computational Dynamical Adaptive System (CDAS)** with the following core components:

- **Gebit**: Fundamental geometric unit with shape and field
- **CTGVEngine**: Propagation engine with shape-specific rules
- **TemporalBindingArbiter**: Ambiguity resolution mechanism
- **GeometricDataModeler**: Data-to-topology converter

2 Geometric Units (Gebits)

2.1 Formal Definition

A Gebit G_i is a 5-tuple:

$$G_i = (\text{shape}_i, x_i(t), \vec{f}_i(t), H_i(t), C_i)$$

where:

- $\text{shape}_i \in \mathcal{S} = \{\text{ORIGIN}, \text{FLOW}, \text{DECISOR}, \dots\}$ (Line 36-47)
- $x_i(t) \in \mathbb{R}_{\geq 0}$: scalar state (activation)
- $\vec{f}_i(t) \in \mathbb{R}^d$: vector field (Line 71-97)
- $H_i(t) = [x_i(t-\tau), \dots, x_i(t)]$: history buffer (Line 113)
- C_i: geometric constraints (Line 50-68)

2.2 Implementation Correspondence

```
    class Gebit:
    def __init__(self, shape: Shape, intensity: float = 1.0,
    label: str = None, dimensions: int = 3):
    self.shape = shape
    self.intensity = intensity
    self.state = 0.0  # x_i(t)
    self.field = VectorField(dimensions)  # \vec{f}_i(t)
    self.connections: Dict['Gebit', float] = {}
    self.history: List[float] = []  # H_i(t)
```

```

```

Listing 1: Gebit Class Definition (Lines 99-184)

3 Field Interference Model

3.1 Mathematical Formulation

The interference between gebits i and j is:

$$\mathcal{I}_{ij}(t) = w_{ij} \cdot (\vec{f}_i \cdot \vec{f}_j) \cdot \cos(\phi_i - \phi_j) \cdot c_i$$

where:

- w_{ij}: topological weight from connections dictionary
- $\vec{f}_i \cdot \vec{f}_j$: dot product of vector fields
- ϕ_i, ϕ_j: field phases
- c_i: coherence factor (1.0 by default)

3.2 Code Implementation

```
    def interfere(self, other: 'VectorField') -> float:
    dot_product = np.dot(self.vector, other.vector)
    phase_diff = abs(self.phase - other.phase)
    return dot_product * math.cos(phase_diff) * self.coherence

```

Listing 2: Interference Calculation (Lines 81-90)

4 Shape-Dependent Update Rules

The core propagation equation for gebit i is:

$$x_i(t+1) = \mathcal{R}_{\text{shape}_i}\left(x_i(t), \sum_{j \in N(i)} \mathcal{I}_{ij}(t), H_i(t)\right)$$

where $\mathcal{R}_{\text{shape}}$ is shape-specific.

4.1 DECISOR Rule (Lines 287-291)

$$\mathcal{R}_{\text{DECISOR}}(x, I, H) = \min\left((x + I)^{\gamma(k)} \cdot d, 1\right)$$

with:

$$\gamma(k) = 0.5 + 0.2 \cdot \log(1 + k), \quad k = |N(i)|$$
$$d = \text{DECAY}[\text{shape}] \quad \text{(from line 57)}$$

Code:

```
    if node.shape == Shape.DECISOR:
    exponent = 0.5 + 0.2 * math.log1p(n_neighbors)
    return min(math.pow(s + input_sum, exponent) * d, 1.0)

```

4.2 RESONATOR Rule (Lines 292-296)

$$\mathcal{R}_{\text{RESONATOR}}(x, I, H) = \frac{(x + I) \cdot d \cdot r}{1 + |(x + I) \cdot d \cdot r|}$$

where:

$$r = 1 + 0.05 \cdot \min(\text{len}(H), 20), \quad d = \text{DECAY}[\text{RESONATOR}] = 1.05$$

Code:

```
    elif node.shape == Shape.RESONATOR:
    resonance = 1.0 + 0.05 * min(node.activation_count, 20)
    new_val = (s + input_sum) * d * resonance
    return new_val / (1.0 + abs(new_val))
```

4.3 LOOP Rule (Lines 297-301)

$$\mathcal{R}_{\text{LOOP}}(x, I, H) = (0.7x + 0.3I + 0.1 \cdot \text{avg}(H_{[-10:]})) \cdot d$$

with $d = 0.98$.

Code:

```
    elif node.shape == Shape.LOOP:
    feedback = node.history[-10:] if len(node.history) >= 10 else node.history
    avg_feedback = np.mean(feedback) if feedback else 0.0
    return (s * 0.7 + input_sum * 0.3 + avg_feedback * 0.1) * d
```

4.4 INHIBITOR Rule (Lines 302-304)

$$\mathcal{R}_{\text{INHIBITOR}}(x, I, H) = \max(0, (x + I) \cdot d \cdot 0.3)$$

with $d = 0.40$.

Code:

```
    elif node.shape == Shape.INHIBITOR:
    return max(0, (s + input_sum) * d * 0.3)
```

5 Global Coherence Metric

5.1 Mathematical Definition

The global coherence $\kappa(t)$ is computed as:

$$\kappa(t) = 0.6 \cdot \left(1 - \frac{\sigma_x}{\mu_x}\right) + 0.4 \cdot \left(1 - \frac{\sigma_k}{\mu_k}\right)$$

where:

- σ_x, μ_x: standard deviation and mean of active states
- σ_k, μ_k: standard deviation and mean of node degrees

5.2 Implementation (Lines 406-419)

```
    def _calculate_coherence(self, nodes: List[Gebit]) -> float:
    all_nodes = self._collect_all_nodes(nodes)

    # State coherence
    states = [n.state for n in all_nodes if n.state > 0]
    state_coh = 1.0 - (np.std(states) / max(np.mean(states), 1e-10))

    # Topological coherence
    degrees = [len(n.connections) for n in all_nodes]
```

```
    degree_coh = 1.0 - (np.std(degrees) / max(np.mean(degrees), 1e-10))

    return (state_coh * 0.6 + degree_coh * 0.4)

```

6 Temporal Binding Arbiter (TBA)

6.1 Narrative Strength Calculation

For source s_k, narrative strength $\Psi(s_k)$ is:

$$\Psi(s_k) = 0.4 \cdot R(s_k) + 0.4 \cdot C(s_k) + 0.2 \cdot S(s_k)$$

where:

$$R(s_k) = \frac{\text{active nodes reachable from } s_k}{\text{total reachable nodes}} \quad \text{(reach)}$$
$$C(s_k) = 1 - \frac{\sigma(\text{states})}{\mu(\text{states})} \quad \text{(coherence)}$$
$$S(s_k) = 1 - \sigma(\text{last 5 history entries}) \quad \text{(stability)}$$

6.2 Implementation (Lines 491-515)

```
    def _calculate_narrative_strength(self, source: Gebit,
    propagation_result: Dict) -> float:
    final_states = propagation_result['final_states']

    # Reach
    active_count = sum(1 for s in final_states.values()
    if s > self.engine.threshold)
    reach = active_count / max(len(final_states), 1)

    # Internal coherence
    states = [s for s in final_states.values() if s > 0]
    if len(states) > 1:
    coherence = 1.0 - (np.std(states) / max(np.mean(states), 1e-10))
    else:
    coherence = 1.0 if states else 0.0

    # Temporal stability
    if len(source.history) >= 5:
    stability = 1.0 - np.std(source.history[-5:])
    else:
    stability = 0.5

    return reach * 0.4 + coherence * 0.4 + stability * 0.2

```

6.3 Ambiguity Calculation

Ambiguity $\mathcal{A}$ is defined as:

$$\mathcal{A} = \frac{\Psi_{(2)}}{\Psi_{(1)}}$$

where $\Psi_{(1)} \geq \Psi_{(2)}$ are the two highest narrative strengths.

Code (Lines 517-532):

```
    def _calculate_ambiguity(self) -> float:
    if not self.narrative_strengths:
    return 1.0

    strengths = list(self.narrative_strengths.values())
    if len(strengths) < 2:
    return 0.0

    max_strength = max(strengths)
```

```
    second_strength = sorted(strengths, reverse=True)[1]

    if max_strength < 0.001:
    return 1.0

    ambiguity = second_strength / max_strength
    return min(ambiguity, 1.0)

```

7 Propagation Algorithm

7.1 Formal Specification

The propagation algorithm follows this iterative process:

1. Initialize active set $A = \{\text{start nodes}\}$
2. For each node $i \in A$:
   (a) Compute total input: $I_i = \sum_{j \in N(i)} \mathcal{I}_{ij}(t)$
   (b) Update state: $x_i(t+1) = \mathcal{R}_{\text{shape}_i}(x_i(t), I_i, H_i(t))$
   (c) Update vector field: $\vec{f}_i(t+1) = \sum_{j \in N(i)} w_{ij} \vec{f}_j(t)$
   (d) If $\Delta x_i = |x_i(t+1) - x_i(t)| >$ threshold, add neighbors to next active set
3. $A \leftarrow$ next active set
4. Repeat until convergence or maximum iterations

7.2 Implementation Correspondence (Lines 196-276)

The main propagation loop implements exactly this algorithm with:

- Convergence threshold: `convergence_threshold` parameter
- Maximum iterations: `max_iterations` parameter
- Adaptive rewiring every 10 iterations (if enabled)

8 Geometric Data Modeling

8.1 Pattern Encoding

Given a 2D pattern $P \in [0,1]^{m \times n}$, the encoding function $\mathcal{E}$ maps:

$$\mathcal{E}(P[y,x]) = G_{y,x} = (\text{shape}(P[y,x]), P[y,x], \vec{0}, [], C_{\text{shape}})$$

where $\text{shape}(v)$ uses the mapping:

$$\text{shape}(v) = \arg\min_{s \in \mathcal{S}} |v - \text{key}(s)|$$

Code (Lines 553-605): Uses default mapping $\{0.0 : \text{INHIBITOR}, 0.3 : \text{FLOW}, 0.6 : \text{AMPLIFIER}, 0.9 : \text{ORIGIN}\}$.

8.2 Connection Weights

Connection weight between adjacent gebits $G_{y,x}$ and $G_{y',x'}$:

$$w = 1 - |P[y,x] - P[y',x']|$$

9 System Parameters

9.1 DECAY Constants (Lines 50-57)

Shape	DECAY Value	Code Reference
ORIGIN	1.00	`DECAY[Shape.ORIGIN]`
FLOW	0.98	`DECAY[Shape.FLOW]`
DECISOR	0.90	`DECAY[Shape.DECISOR]`
RESONATOR	1.05	`DECAY[Shape.RESONATOR]`
INHIBITOR	0.40	`DECAY[Shape.INHIBITOR]`
LOOP	0.98	`DECAY[Shape.LOOP]`

9.2 Geometric Constraints (Lines 59-68)

Shape	Max Connections	Allow Cycles	Symmetric Required
FLOW	2	False	False
DECISOR	3	True	True
RESONATOR	4	True	True
LOOP	2	True	True

10 Experimental Results

From the demonstration output:

1. **Simple Network**: Coherence = 0.769, showing moderate structural organization
2. **TBA Test**: Ambiguity remains at 1.000, indicating equal narrative strengths
3. **Pattern Processing**: Mean difference = 0.5239, showing significant transformation
4. **Feedback Loop**: States propagate but limited to immediate neighbors

11 Conclusion

This formal specification provides a complete mathematical description of the CTGV V1.3 system, with direct references to the Python implementation. The formalism demonstrates that:

1. Every mathematical equation has a corresponding code implementation
2. Parameters are explicitly defined and match the code
3. The system's behavior is fully specified by the shape-dependent rules
4. Coherence and ambiguity metrics are computable and measurable

The CTGV system represents a novel computational paradigm where geometric topology, field interference, and adaptive dynamics interact to produce emergent computational behaviors. The precise alignment between this formalism and the implementation ensures that theoretical predictions can be experimentally verified through simulation.