Morphonic Modal Languages: Schema Reference and Implementation Guide

Complete Technical Specifications for Λ𝒪ψ, Λ𝒪γ, Λ𝒪χ Compilers and Runtimes

Abstract

This document provides complete schema specifications, validation rules, and implementation guidelines for the three morphonic modal control languages. It serves as the authoritative reference for compiler developers, runtime implementers, and system integrators.

1. Phonetic Language (Λ𝒪ψ) Specification

1.1 Core Schema (`phon.schema.json`)

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Morphonic Phonetic Control Language",
  "type": "object",
  "required": ["utterance", "context"],
  "properties": {
    "utterance": {
      "type": "string",
      "description": "Natural language command sequence",
      "pattern": "^[A-Z][A-Za-z\\s]+$"
    },
    "tokens": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/token"
      }
    },
    "rhythm": {
      "type": "array",
      "description": "Timing weights for each token",
      "items": {
        "type": "number",
        "minimum": 0.0,
        "maximum": 2.0
      }
    },
    "stress_pattern": {
      "type": "array",
      "description": "Emphasis markers",
      "items": {
        "type": "integer",
        "enum": [0, 1, 2]
```

```
      }
    },
    "context": {
      "$ref": "#/definitions/execution_context"
    }
  },
  "definitions": {
    "token": {
      "type": "object",
      "required": ["word", "type"],
      "properties": {
        "word": { "type": "string" },
        "type": {
          "enum": ["VERB", "NOUN", "MODIFIER", "CLOSURE", "OBSERVATION"]
        },
        "morphonic_op": {
          "enum": ["obs", "close", "conj", "chamber", "fire", "dual"]
        }
      }
    },
    "execution_context": {
      "type": "object",
      "properties": {
        "lattice_dim": { "type": "integer", "minimum": 2 },
        "symmetry_group": {
          "enum": ["D4", "D8", "D12", "D24", "E8"]
        },
        "precision": { "type": "number", "default": 1e-6 }
      }
    }
  }
}
```

1.2 Token Mapping Table

Word	Type	Morphonic Op	Lambda	Notes
OBSERVE	VERB	obs	$\lambda x.\,\mathrm{obs}(x)$	Collapse to eigenstate
CLOSE	VERB	close	$\lambda x.\,\mathrm{close}(x)$	Find equilibrium
FIRE	VERB	fire	$\lambda x.\,\mathrm{fire}(x)$	Trigger boundary
BALANCE	VERB	conj	$\lambda x.\,\mathrm{conj}(x, \sigma(x))$	Conjugate pair
CHAMBER	NOUN	chamber	$\lambda x.\,\mathrm{chamber}(x)$	Identify partition
STATE	NOUN	-	Identity	Input/output state
DUAL	MODIFIER	dual	$\lambda x.\,\sigma(x)$	Return beam

1.3 Phonetic Parsing Algorithm

```
def parse_phonetic(utterance, context):
    """
    Parse phonetic utterance into lambda expression.

    Args:
        utterance: Natural language command string
        context: Execution context (lattice, symmetry, etc.)

    Returns:
        Compiled lambda expression
    """
    # Step 1: Tokenize
    tokens = tokenize(utterance)

    # Step 2: Identify morphonic operations
    ops = [token_to_op(t) for t in tokens]

    # Step 3: Build syntax tree
    tree = build_tree(ops)

    # Step 4: Type check
    typed_tree = type_check(tree, context)

    # Step 5: Optimize (closure identification)
    optimized = optimize_closures(typed_tree)

    # Step 6: Compile to lambda
    lambda_expr = compile_to_lambda(optimized)

    return lambda_expr
```

1.4 Example Phonetic Programs

Program 1: Simple Observation

```
{
  "utterance": "OBSERVE state CLOSE",
  "rhythm": [1.0, 0.8, 1.2],
  "context": {
    "lattice_dim": 8,
    "symmetry_group": "D8"
  }
}
```

Compiles to:

```
λ state. close(obs(state))
```

Program 2: Conjugate Balance

```
{
  "utterance": "BALANCE charge CONSERVE energy",
  "rhythm": [1.2, 0.9, 1.1, 0.8],
  "context": {
    "lattice_dim": 4,
    "symmetry_group": "D4"
  }
}
```

Compiles to:

```
λ (charge, energy). close(conj(charge, energy))
```

2. Glyphic Language (Λ𝒪γ) Specification

2.1 Core Schema (`glyph.schema.json`)

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Morphonic Glyphic Control Language",
  "type": "object",
  "required": ["sequence"],
  "properties": {
    "sequence": {
      "type": "array",
      "description": "Ordered list of glyphs",
      "items": {
        "$ref": "#/definitions/glyph"
      }
    },
    "parallel_groups": {
      "type": "array",
      "description": "Glyphs to execute in parallel",
      "items": {
        "type": "array",
        "items": { "$ref": "#/definitions/glyph" }
      }
    },
    "layout": {
      "$ref": "#/definitions/geometric_layout"
    }
  },
  "definitions": {
    "glyph": {
      "type": "object",
      "required": ["symbol", "operation"],
      "properties": {
        "symbol": {
          "type": "string",
          "enum": ["◉", "⬡", "⚡", "⊚", "⟷", "∇", "△", "☯"]
        },
```

```
        "operation": {
          "enum": ["obs", "chamber", "fire", "close", "conj", "grad", "dual", "balance"]
        },
        "parameters": {
          "type": "object",
          "additionalProperties": true
        },
        "position": {
          "type": "object",
          "properties": {
            "x": { "type": "number" },
            "y": { "type": "number" }
          }
        }
      }
    },
    "geometric_layout": {
      "type": "object",
      "properties": {
        "type": {
          "enum": ["linear", "circular", "lattice", "tree"]
        },
        "connections": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "from": { "type": "integer" },
              "to": { "type": "integer" },
              "type": {
                "enum": ["sequential", "parallel", "conjugate"]
              }
            }
          }
        }
      }
    }
  }
}
```

2.2 Glyph Symbol Reference

Symbol	Unicode	Operation	Geometry	Lambda
◉	U+25C9	Observe	Point	obs(x)
⬡	U+2B21	Chamber	Hexagon	chamber(x)
⚡	U+26A1	Fire	Lightning	fire(x)
⊚	U+229A	Close	Circled dot	close(x)
⟷	U+27F7	Conjugate	Bidirectional	conj(x,y)
∇	U+2207	Gradient	Nabla	grad(x)
△	U+25B3	Dual	Triangle	dual(x)

Symbol	Unicode	Operation	Geometry	Lambda
☯	U+262F	Balance	Yin-yang	balance(x)

2.3 Glyphic Composition Rules

Rule 1: Sequential Composition

```
G1 → G2 → G3
```

Compiles to:

```
λ x. G3(G2(G1(x)))
```

Rule 2: Parallel Composition

```
G1 ∥ G2
```

Compiles to:

```
λ x. (G1(x), G2(x))
```

Rule 3: Conjugate Composition

```
G1 ⟷ G2
```

Compiles to:

```
λ x. conj(G1(x), G2(dual(x)))
```

2.4 Example Glyphic Programs

Program 1: Observation-Chamber-Closure

```
{
  "sequence": [
    { "symbol": "◉", "operation": "obs" },
    { "symbol": "⬡", "operation": "chamber", "parameters": { "dim": 8 } },
    { "symbol": "◎", "operation": "close" }
  ],
  "layout": {
    "type": "linear"
  }
}
```

Visual representation:

```
◉ → ⬡ → ⊚
```

Compiles to:

```
λ state. close(chamber_8D(obs(state)))
```

Program 2: Conjugate Balance

```
{
  "parallel_groups": [
    [
      { "symbol": "◉", "operation": "obs" },
      { "symbol": "△", "operation": "dual" }
    ]
  ],
  "sequence": [
    { "symbol": "⟷", "operation": "conj" },
    { "symbol": "☯", "operation": "balance" }
  ]
}
```

Visual representation:

```
◉ ‖ △
  ⟷
  ☯
```

Compiles to:

```
λ state. balance(conj(obs(state), dual(state)))
```

3. Chromatic Language (Λ𝒞χ) Specification

3.1 Core Schema (`chrom.schema.json`)

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Morphonic Chromatic Control Language",
  "type": "object",
  "required": ["channels", "decisions"],
  "properties": {
    "channels": {
      "type": "object",
      "description": "Color-typed computation channels",
      "patternProperties": {
```

```
        "^(red|blue|green|yellow|magenta|cyan)$": {
          "$ref": "#/definitions/channel_spec"
        }
      }
    },
    "decisions": {
      "type": "array",
      "description": "Decision points and routing",
      "items": {
        "$ref": "#/definitions/decision"
      }
    },
    "merge_policy": {
      "enum": ["parallel", "sequential", "priority", "weighted"]
    }
  },
  "definitions": {
    "channel_spec": {
      "type": "object",
      "required": ["operation"],
      "properties": {
        "operation": {
          "type": "string",
          "description": "Lambda expression for this channel"
        },
        "priority": {
          "type": "integer",
          "minimum": 0,
          "maximum": 10
        },
        "dependencies": {
          "type": "array",
          "items": { "type": "string" }
        }
      }
    },
    "decision": {
      "type": "object",
      "required": ["condition", "true_channel", "false_channel"],
      "properties": {
        "condition": {
          "type": "string",
          "description": "Boolean lambda expression"
        },
        "true_channel": {
          "enum": ["red", "blue", "green", "yellow", "magenta", "cyan"]
        },
        "false_channel": {
          "enum": ["red", "blue", "green", "yellow", "magenta", "cyan"]
        }
      }
    }
  }
}
```

3.2 Channel Color Semantics

Color	Priority	Semantics	Typical Use
Red	High (9-10)	Forward beam, primary computation	Main algorithm path
Blue	Medium (5-7)	Return beam, verification	Conjugate checking
Green	Low (1-3)	Equilibrium, closure states	Final output
Yellow	High (8-9)	Warning, boundary conditions	Error handling
Magenta	Medium (4-6)	Cross-channel communication	Data sharing
Cyan	Variable	Observation, measurement	State inspection

3.3 Decision Routing

```
def route_decision(condition, channels, state):
    """
    Route computation based on chromatic decision.

    Args:
        condition: Boolean lambda expression
        channels: Dict of color → channel_spec
        state: Current computation state

    Returns:
        Routed channel result
    """
    if eval_condition(condition, state):
        return execute_channel(channels[true_channel], state)
    else:
        return execute_channel(channels[false_channel], state)
```

3.4 Example Chromatic Programs

Program 1: Parallel Red/Blue Conjugate

```
{
  "channels": {
    "red": {
      "operation": "obs(state)",
      "priority": 10
    },
    "blue": {
      "operation": "conj(dual(state))",
      "priority": 8
    },
    "green": {
      "operation": "close(merge(red, blue))",
      "priority": 2
    }
  },
```

```
  "merge_policy": "parallel"
}
```

Compiles to:

```
λ state.
  let red_result = obs(state) in
  let blue_result = conj(dual(state)) in
  close(merge(red_result, blue_result))
```

Program 2: Conditional Decision Routing

```
{
  "channels": {
    "red": { "operation": "fire(state)" },
    "yellow": { "operation": "warn(state)" },
    "green": { "operation": "close(state)" }
  },
  "decisions": [
    {
      "condition": "threshold(state) &gt; 0.9",
      "true_channel": "red",
      "false_channel": "yellow"
    }
  ]
}
```

Compiles to:

```
λ state.
  if threshold(state) &gt; 0.9 then
    fire(state)
  else
    warn(state)
```

4. Phi (φ) Energy Model Specification

4.1 Core Schema (`phi_model.schema.json`)

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Morphonic Energy Model",
  "type": "object",
  "required": ["operations", "total_energy"],
  "properties": {
    "operations": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/operation_cost"
```

```
      }
    },
    "total_energy": {
      "type": "number",
      "description": "Total energy in units of k_B * T"
    },
    "entropy_change": {
      "type": "number",
      "description": "Net entropy change in nats"
    }
  },
  "definitions": {
    "operation_cost": {
      "type": "object",
      "required": ["operation", "cost"],
      "properties": {
        "operation": {
          "enum": ["obs", "close", "conj", "chamber", "fire", "dual"]
        },
        "cost": {
          "type": "number",
          "description": "Energy cost in k_B * T units"
        },
        "reversible": {
          "type": "boolean"
        },
        "entropy_delta": {
          "type": "number"
        }
      }
    }
  }
}
```

4.2 Energy Cost Table

Operation	Reversible	Cost (k_B T units)	Entropy Δ (nats)
obs	No	ln(n)	-ln(n)
close	Yes (if converges)	0	0
conj	Yes	0	0
chamber	Yes	0	0
fire	No	ln(2)	-ln(2)
dual	Yes	0	0

where n = number of states being collapsed

4.3 Energy Validation

```
def validate_energy(program, expected_max):
    """
    Verify program energy consumption.

    Args:
        program: Compiled lambda expression
        expected_max: Maximum allowed energy (k_B T units)

    Returns:
        (valid: bool, actual_cost: float, breakdown: dict)
    """
    cost_breakdown = {}
    total = 0.0

    for op in extract_operations(program):
        op_cost = get_operation_cost(op)
        cost_breakdown[op] = op_cost
        total += op_cost

    valid = (total &lt;= expected_max)

    return valid, total, cost_breakdown
```

5. Run Configuration Specification

5.1 Schema (`run_config.schema.json`)

```
input:
  type: State | Observable | Chamber
  dimension: integer
  lattice: D4 | D8 | D12 | E8
  initial_value: array[float]

execution:
  mode: sequential | parallel | distributed
  precision: float (default 1e-6)
  max_iterations: integer (default 1000)
  convergence_threshold: float

pipeline:
  - operation: string
    parameters: object
    validation: string

output:
  type: State | Observable | Chamber
  format: json | binary | tensor
  validation:
```

```
    - check: string
      tolerance: float
```

5.2 Example Run Configurations

Configuration 1: Simple Observation

```
input:
  type: State
  dimension: 8
  lattice: D8
  initial_value: [1, 0, 0, 0, 0, 0, 0, 0]

pipeline:
  - operation: obs
  - operation: close

output:
  type: Observable
  validation:
    - check: eigenstate
      tolerance: 1e-6
```

Configuration 2: Complex Chromatic Flow

```
input:
  type: State
  dimension: 24
  lattice: D24

execution:
  mode: parallel
  precision: 1e-9

pipeline:
  - operation: chromatic
    parameters:
      channels:
        red: obs
        blue: conj
        green: close
  - operation: merge
    parameters:
      policy: weighted

output:
  type: Chamber
  format: tensor
```

6. Expectations and Validation

6.1 Schema (`expectations.schema.json`)

```
{
  "expected_closure": {
    "type": "object",
    "properties": {
      "dimension": { "type": "integer" },
      "eigenvalue": { "type": "number" },
      "tolerance": { "type": "number" }
    }
  },
  "energy_bound": {
    "type": "object",
    "properties": {
      "max_cost": { "type": "string" },
      "operations": {
        "type": "array",
        "items": { "type": "string" }
      }
    }
  },
  "convergence": {
    "type": "object",
    "properties": {
      "max_iterations": { "type": "integer" },
      "threshold": { "type": "number" }
    }
  }
}
```

6.2 Validation Rules

1. **Type Safety:** Output type matches declared type
2. **Energy Bound:** Total φ ≤ expected maximum
3. **Convergence:** close() reaches fixed point within max_iterations
4. **Conjugate Symmetry:** conj(x, dual(x)) is self-dual
5. **Chamber Closure:** All chambers are closed under conjugation

7. Compiler Implementation Guide

7.1 Compilation Stages

```
Source (Phon/Glyph/Chrom)
    ↓
Lexical Analysis
    ↓
```

```
Parsing (AST generation)
    ↓
Type Checking
    ↓
Optimization (closure identification)
    ↓
Lambda Generation
    ↓
Energy Analysis (φ-model)
    ↓
Code Generation (executable)
```

7.2 Optimization Passes

1. **Closure Identification:** Detect fixed-point structures
2. **Conjugate Merging:** Combine conjugate pairs
3. **Dead Chamber Elimination:** Remove unused partitions
4. **Energy Minimization:** Prefer reversible operations

7.3 Runtime Integration

```
class MorphonicRuntime:
    def __init__(self, lattice_dim, symmetry_group):
        self.lattice = initialize_lattice(lattice_dim)
        self.symmetry = load_group(symmetry_group)
        self.phi_tracker = EnergyTracker()

    def execute(self, program, input_state):
        # Validate input
        validate_state(input_state, self.lattice)

        # Execute compiled lambda
        result = eval_lambda(program, input_state)

        # Track energy
        energy_cost = self.phi_tracker.compute(program)

        # Validate output
        validate_expectations(result, energy_cost)

        return result, energy_cost
```

8. Conclusion

This specification provides complete schemas, implementation guidelines, and validation rules for all three morphonic modal control languages. Compiler developers should:

1. Implement parsers conforming to these schemas
2. Generate lambda expressions following the typing rules

3. Integrate φ-model energy tracking
4. Validate against expectations schemas

All schemas are provided as JSON files in the accompanying materials.