MES Documentation

Viktor Winschel

March 26, 2025

Contents

Part I

Home

Part II

Memory Evolutive Systems (MES)

Welcome to the documentation for the Memory Evolutive Systems (MES) package in Julia.

Chapter 1

Overview

Memory Evolutive Systems (MES) is a mathematical framework for modeling complex hierarchical systems that evolve over time. This package provides a Julia implementation of the core concepts and tools for working with MES.

Chapter 2

Features

- **Category Theory Foundations**
  - Create and manipulate categories with objects and morphisms
  - Verify category properties (composition, identity)
  - Work with patterns and calculate colimits
- **Memory Systems**
  - Store and retrieve information in category-based memory
  - Support for variable multiplicity and decay
  - Hierarchical organization of memory components
- **Pattern Synchronization**
  - Create and verify synchronizations between patterns
  - Support for complex pattern matching and evolution
- **Advanced Features**
  - Hierarchical categories with multiple complexity levels
  - Co-regulators for system adaptation
  - Memory components with variable multiplicity

Chapter 3

Installation

To install MES, use Julia's package manager:

```
using Pkg
Pkg.add("MES")
```

Chapter 4

Quick Start

Here's a simple example of creating a category and working with patterns:

```
using MES

# Create a category
objects = ["A", "B", "C"]
morphisms = Dict(
    ("A", "A") => ["id_A"],  # Identity morphism
    ("B", "B") => ["id_B"],
    ("C", "C") => ["id_C"],
    ("A", "B") => ["f"],     # Morphism from A to B
    ("B", "C") => ["g"]      # Morphism from B to C
)
category = create_category(objects, morphisms)

# Create a pattern
pattern = create_pattern(category, ["A", "B"], [("A", "B")])

# Calculate colimit
colimit = calculate_colimit(pattern)
```

Chapter 5

Documentation Structure

- Getting Started - Basic tutorials and examples
- Theory - Mathematical foundations and concepts
- Examples - Practical examples and use cases
- API Reference - Detailed documentation of types and functions
- Papers - Academic references and background

Chapter 6

Contributing

Contributions are welcome! Please see our GitHub repository for:

- Issue tracking
- Feature requests
- Pull requests
- Development guidelines

Chapter 7

License

MES is licensed under the MIT License. See the LICENSE file for details.

Part III

Theory

Chapter 8

Categories

8.1 Categories

Categories are fundamental mathematical structures that represent relationships between objects. In MES, categories provide the foundation for modeling system components and their interactions, as described in Chapter 1 of the 2007 book by Ehresmann and Vanbremeersch.

Basic Concepts

Mathematical Definition

From the 2007 book, Chapter 1:

A category $\mathcal{C}$ consists of:

- A collection of objects: $\text{Ob}(\mathcal{C})$
- A collection of morphisms: $\text{Hom}(\mathcal{C})$
- Composition operation: $\circ : \text{Hom}(B, C) \times \text{Hom}(A, B) \to \text{Hom}(A, C)$
- Identity morphisms: $\text{id}_A : A \to A$

Satisfying the following axioms:

1. **Associativity**: $\forall f : A \to B, g : B \to C, h : C \to D \ (h \circ g) \circ f = h \circ (g \circ f)$
2. **Identity**: $\forall f : A \to B \ f \circ \text{id}_A = f = \text{id}_B \circ f$

Implementation

In MES, categories are implemented using the `Category` type:

```
struct Category
    objects::Vector{String}
    morphisms::Vector{Tuple{String, String, String}}  # (source, target, name)
end
```

Example

Here's a simple example of creating and verifying a category:

```
# Create a simple category with three objects and their morphisms
cat = Category(
    ["A", "B", "C"],
    [
        ("A", "B", "f"),
        ("B", "C", "g"),
        ("A", "C", "h")
    ]
)

# Verify category properties
verify_category(cat)  # Returns true if all axioms are satisfied
```

Category Properties

Mathematical Properties

From the 2007 book, Chapter 1:

Categories in MES satisfy several important properties:

1. **Composition Closure**: $\forall f : A \to B, g : B \to C \in \text{Hom}(\mathcal{C})\ \exists h : A \to C = g \circ f \in \text{Hom}(\mathcal{C})$
2. **Identity Existence**: $\forall A \in \text{Ob}(\mathcal{C})\ \exists \text{id}_A : A \to A \in \text{Hom}(\mathcal{C})$
3. **Composition Associativity**: $\forall f : A \to B, g : B \to C, h : C \to D\ (h \circ g) \circ f = h \circ (g \circ f)$

Implementation

These properties are verified in the code:

```
# Check composition closure
function verify_composition_closure(cat::Category)
    for (src1, tgt1, _) in cat.morphisms
        for (src2, tgt2, _) in cat.morphisms
            if tgt1 == src2
                # Check if composition exists
                has_composition = any(src1 == src && tgt2 == tgt
                    for (src, tgt, _) in cat.morphisms)
                if !has_composition
                    return false
                end
            end
        end
    end
    return true
end

# Check identity existence
function verify_identity_existence(cat::Category)
    for obj in cat.objects
        has_identity = any(src == obj && tgt == obj
```

```
            for (src, tgt, _) in cat.morphisms)
        if !has_identity
            return false
        end
    end
    return true
end
```

Example

Here's an example of a category representing a simple neural network:

```
# Create a neural network category
nn_cat = Category(
    ["Input", "Hidden", "Output"],
    [
        ("Input", "Hidden", "weights1"),
        ("Hidden", "Output", "weights2"),
        ("Input", "Input", "id_input"),
        ("Hidden", "Hidden", "id_hidden"),
        ("Output", "Output", "id_output")
    ]
)

# Verify all properties
@assert verify_category(nn_cat)
@assert verify_composition_closure(nn_cat)
@assert verify_identity_existence(nn_cat)
```

Applications

Neural Networks

From the 2007 book, Chapter 9 (MENS):

Categories can represent neural network architectures:

```
# Create a more complex neural network category
complex_nn = Category(
    ["Input", "Hidden1", "Hidden2", "Output"],
    [
        ("Input", "Hidden1", "w1"),
        ("Hidden1", "Hidden2", "w2"),
        ("Hidden2", "Output", "w3"),
        ("Input", "Input", "id_input"),
        ("Hidden1", "Hidden1", "id_hidden1"),
        ("Hidden2", "Hidden2", "id_hidden2"),
        ("Output", "Output", "id_output")
    ]
)
```

Social Networks

From the 2023 paper:

Categories can model social network relationships:

```
# Create a social network category
social_net = Category(
    ["User", "Post", "Comment"],
    [
        ("User", "Post", "creates"),
        ("Post", "Comment", "has"),
        ("User", "Comment", "writes"),
        ("User", "User", "id_user"),
        ("Post", "Post", "id_post"),
        ("Comment", "Comment", "id_comment")
    ]
)
```

For more detailed information about categories in MES, refer to:

1. Chapter 1 of the 2007 book for comprehensive mathematical foundations
2. The 2023 paper for recent developments and applications
3. The original papers cited in both works for specific mathematical proofs

Further Reading

For more detailed information about category theory in MES, refer to:

- Patterns - Pattern recognition and colimits
- Memory Systems - Memory and system evolution
- Synchronization - System coordination
- Category Examples - Practical examples

Systems Theory and Graphs

A system can be represented as a collection of objects and their relationships. In MES, we model this using graphs and categories.

```
# Create a simple graph representing objects and relations
using MES

# Define objects (vertices)
objects = ["Cell1", "Cell2", "Protein1"]

# Define relations (edges)
relations = [
    ("Cell1", "Cell2", "signals_to"),
    ("Protein1", "Cell1", "regulates")
]

# Create the graph
graph = create_graph(objects, relations)
```

Categories and Functors

A category extends the concept of a graph by adding:

- Identity morphisms for each object
- Composition of morphisms
- Associativity and unit laws

```
# Create a category from the graph
category = create_category(graph)

# Add a composition rule
add_composition!(category, "signals_to", "regulates", "indirect_regulation")

# Verify category laws
verify_category(category)
```

Categories in Systems Theory

Categories provide a powerful framework for modeling systems:

1. **Configuration Categories**

   ```
   # Model a biological system's state
   config = configuration_category(
       objects = ["Cell", "Membrane", "Nucleus"],
       morphisms = ["contains", "surrounds"],
       compositions = [("contains", "surrounds", "protects")]
   )
   ```

2. **Transformations Between States**

   ```
   # Model system evolution
   F = functor(
       source = initial_state,
       target = final_state,
       object_map = Dict("Cell" => "DividedCell"),
       morphism_map = Dict("contains" => "contains_both")
   )
   ```

Category Construction

Categories can be built in several ways:

1. **Via Generators and Relations**

   ```
   # Define basic generators
   generators = category_generators(
       objects = ["A", "B", "C"],
       basic_morphisms = ["f: A→B", "g: B→C"]
   )

   # Add relations
   add_relations!(generators, ["g ∘ f = h"])
   ```

2. **Labelled Categories**

```
# Create a category with labeled morphisms
labeled_cat = labeled_category(
    objects = ["Neuron1", "Neuron2"],
    labels = ["excitatory", "inhibitory"],
    morphisms = [
        ("Neuron1", "Neuron2", "excitatory"),
        ("Neuron2", "Neuron1", "inhibitory")
    ]
)
```

Implementation Details

The MES package provides several types and functions for working with categories:

```
# Basic category type
struct Category{T}
    objects::Set{T}
    morphisms::Dict{Tuple{T,T}, Set{String}}
    compositions::Dict{Tuple{String,String}, String}
end

# Create a category
function create_category(
    objects::Vector{T},
    morphisms::Vector{Tuple{T,T,String}}
) where T
    # Implementation...
end

# Verify category laws
function verify_category(cat::Category)
    # Check identity morphisms
    # Check composition closure
    # Check associativity
    # Check identity laws
end
```

Mathematical Background

For the mathematical foundations of categories, including formal definitions and proofs, see the Category Theory Appendix.

Examples

See the Category Examples section for more practical applications, including:

- Biological system modeling
- Neural network representation
- Social network analysis

Category Properties

Categories in MES satisfy several important properties:

1. **Composition Closure**: $\forall f : A \to B, g : B \to C \in \mathsf{Hom}(\mathcal{C})\ \exists h : A \to C = g \circ f \in \mathsf{Hom}(\mathcal{C})$
2. **Identity Existence**: $\forall A \in \mathsf{Ob}(\mathcal{C})\ \exists \mathsf{id}_A : A \to A \in \mathsf{Hom}(\mathcal{C})$
3. **Composition Associativity**: $\forall f : A \to B, g : B \to C, h : C \to D\ (h \circ g) \circ f = h \circ (g \circ f)$

Chapter 9

Patterns

9.1 Patterns and Colimits

Patterns are essential structures in MES that represent complex configurations of objects and their relationships. They form the basis for understanding how different components of a system interact and evolve.

Basic Concepts

Mathematical Definition

A pattern P in a category $\mathcal{C}$ is defined as: $P = \{O_i, M_j\}$ where:

- $$O_i$$

  are objects in $\mathcal{C}$

- $$M_j$$

  are morphisms between these objects

The pattern recognition process can be expressed mathematically as: $P = \{O_i, M_j\}$ where O_i are objects and M_j are morphism

Implementation

In MES, patterns are implemented using the `Pattern` type:

```
struct Pattern
    objects::Vector{String}
    morphisms::Vector{Tuple{String, String, String}}
end
```

Example

Here's a simple example of creating and working with patterns:

```
# Create a simple pattern with three objects and their morphisms
pattern = Pattern(
    ["A", "B", "C"],
    [
        ("A", "B", "f"),
        ("B", "C", "g"),
        ("A", "C", "h")
    ]
)

# Verify pattern properties
verify_pattern(pattern)  # Returns true if pattern is valid
```

Colimits

Mathematical Definition

The colimit of a pattern P, denoted colim(P), is an object C together with morphisms $\varphi_i : O_i \to C$ satisfying the universal property:

$$\forall X \in \mathcal{C}, \forall \psi_i : O_i \to X \text{ such that } \psi_j \circ M_j = \psi_i$$

$$\exists! \psi : C \to X \text{ such that } \psi \circ \varphi_i = \psi_i$$

Implementation

Colimits are calculated in the code:

```
function calculate_colimit(pattern::Pattern)
    # Create a new object for the colimit
    colimit_obj = "colim_$(join(pattern.objects, "_"))"

    # Create morphisms from each object to the colimit
    colimit_morphisms = [
        (obj, colimit_obj, "to_colimit_$obj")
        for obj in pattern.objects
    ]

    # Return the colimit object and morphisms
    return (colimit_obj, colimit_morphisms)
end

# Verify colimit properties
function verify_colimit(pattern::Pattern, colimit_obj, colimit_morphisms)
    # Check universal property
    for obj in pattern.objects
        has_morphism = any(src == obj && tgt == colimit_obj
```

```
            for (src, tgt, _) in colimit_morphisms)
        if !has_morphism
            return false
        end
    end
    return true
end
```

Example

Here's an example of calculating and verifying a colimit:

```
# Create a pattern representing a simple neural network
nn_pattern = Pattern(
    ["Input", "Hidden", "Output"],
    [
        ("Input", "Hidden", "weights1"),
        ("Hidden", "Output", "weights2")
    ]
)

# Calculate its colimit
colimit_obj, colimit_morphisms = calculate_colimit(nn_pattern)

# Verify the colimit
@assert verify_colimit(nn_pattern, colimit_obj, colimit_morphisms)
```

Applications

Neural Networks

Patterns can represent neural network architectures:

```
# Create a pattern for a more complex neural network
complex_nn_pattern = Pattern(
    ["Input", "Hidden1", "Hidden2", "Output"],
    [
        ("Input", "Hidden1", "w1"),
        ("Hidden1", "Hidden2", "w2"),
        ("Hidden2", "Output", "w3")
    ]
)

# Calculate and verify its colimit
colimit_obj, colimit_morphisms = calculate_colimit(complex_nn_pattern)
@assert verify_colimit(complex_nn_pattern, colimit_obj, colimit_morphisms)
```

Social Networks

Patterns can model social network relationships:

```
# Create a pattern for a social network
social_pattern = Pattern(
    ["User", "Post", "Comment"],
    [
```

```
        ("User", "Post", "creates"),
        ("Post", "Comment", "has"),
        ("User", "Comment", "writes")
    ]
)

# Calculate and verify its colimit
colimit_obj, colimit_morphisms = calculate_colimit(social_pattern)
@assert verify_colimit(social_pattern, colimit_obj, colimit_morphisms)
```

For more detailed information about patterns and colimits in MES, refer to the original papers in the Papers section.

Chapter 10

Memory Systems

10.1 Memory Systems

Memory systems in MES extend the basic category structure with memory capabilities and procedures for processing information.

Basic Concepts

Mathematical Definition

A Memory Evolutive System $\mathcal{M}$ is defined as: $\mathcal{M} = (\mathcal{C}, M, \mathcal{P})$ where:

- $\mathcal{C}$ is a category
- M is a memory system
- $\mathcal{P}$ is a set of procedures

The memory system M is a function: $M : T \times E \to D$ where:

- T is the time domain

- E

  is the event space

- D

  is the data space

Implementation

In MES, memory systems are implemented using the `MemorySystem` type:

```
struct MemorySystem
    category::Category
    memory::Dict{String, Any}
    procedures::Vector{Function}
end
```

Example

Here's a simple example of creating and working with a memory system:

```
# Create a memory system with a simple category
cat = Category(
    ["A", "B"],
    [("A", "B", "f")]
)

# Initialize memory and procedures
memory = Dict{String, Any}()
procedures = [
    function store_data(data)
        memory["data"] = data
    end,
    function retrieve_data()
        return get(memory, "data", nothing)
    end
]

# Create the memory system
mem_sys = MemorySystem(cat, memory, procedures)

# Use the procedures
store_data(mem_sys, "test data")
@assert retrieve_data(mem_sys) == "test data"
```

Memory Operations

Mathematical Properties

The memory operations in MES satisfy several important properties:

1. **Memory Storage**: $\forall t \in T, e \in E, d \in D\ M(t, e) = d$
2. **Memory Retrieval**: $\forall t \in T, e \in E\ \exists d \in D : M(t, e) = d$
3. **Memory Update**: $\forall t_1, t_2 \in T, e \in E\ M(t_2, e) = f(M(t_1, e))$

Implementation

These operations are implemented in the code:

```
function store_data(mem_sys::MemorySystem, key::String, data::Any)
    mem_sys.memory[key] = data
end

function retrieve_data(mem_sys::MemorySystem, key::String)
    return get(mem_sys.memory, key, nothing)
end

function update_data(mem_sys::MemorySystem, key::String, update_fn::Function)
    if haskey(mem_sys.memory, key)
        mem_sys.memory[key] = update_fn(mem_sys.memory[key])
    end
end
```

Example

Here's an example of using memory operations:

```
# Create a memory system for tracking neural network states
nn_mem = MemorySystem(
    Category(
        ["Input", "Hidden", "Output"],
        [("Input", "Hidden", "w1"), ("Hidden", "Output", "w2")]
    ),
    Dict{String, Any}(),
    []
)

# Store network weights
store_data(nn_mem, "weights", Dict(
    "w1" => rand(10, 5),
    "w2" => rand(5, 2)
))

# Update weights with gradient descent
function update_weights(weights, learning_rate=0.01)
    return Dict(k => v .- learning_rate .* rand(size(v))
        for (k, v) in weights)
end

update_data(nn_mem, "weights", w -> update_weights(w))
```

Applications

Neural Networks

Memory systems can track neural network states:

```
# Create a memory system for a complex neural network
complex_nn_mem = MemorySystem(
    Category(
        ["Input", "Hidden1", "Hidden2", "Output"],
        [
            ("Input", "Hidden1", "w1"),
            ("Hidden1", "Hidden2", "w2"),
            ("Hidden2", "Output", "w3")
        ]
    ),
    Dict{String, Any}(),
    []
)

# Store network configuration
store_data(complex_nn_mem, "config", Dict(
    "architecture" => ["Input", "Hidden1", "Hidden2", "Output"],
    "activation" => "relu",
    "optimizer" => "adam"
))

# Store training history
store_data(complex_nn_mem, "history", Dict(
    "loss" => Float64[],
    "accuracy" => Float64[]
))
```

Social Networks

Memory systems can track social network states:

```
# Create a memory system for a social network
social_mem = MemorySystem(
    Category(
        ["User", "Post", "Comment"],
        [
            ("User", "Post", "creates"),
            ("Post", "Comment", "has"),
            ("User", "Comment", "writes")
        ]
    ),
    Dict{String, Any}(),
    []
)

# Store user data
store_data(social_mem, "users", Dict(
    "user1" => Dict(
        "name" => "Alice",
        "posts" => ["post1", "post2"],
        "comments" => ["comment1"]
    )
))

# Store post data
store_data(social_mem, "posts", Dict(
```

```
    "post1" => Dict(
        "content" => "Hello, world!",
        "author" => "user1",
        "comments" => ["comment1"]
    )
))
```

For more detailed information about memory systems in MES, refer to the original papers in the Papers section.

Chapter 11

Synchronization

11.1 Synchronization

Synchronization is a crucial aspect of MES that ensures coordinated evolution of different system components. It enables the harmonious interaction between various parts of a complex system.

Basic Concepts

Mathematical Definition

A synchronization $\mathcal{S}$ between two patterns P_1 and P_2 is defined as: $\mathcal{S} = (P_1, P_2, \varphi)$ where:

- $$P_1$$ and P_2 are patterns
- $$\varphi$$ is a morphism between their colimits

The synchronization process satisfies several mathematical properties:

1. **Compatibility**: $\forall x \in \text{colim}(P_1), y \in \text{colim}(P_2)\ \varphi(x) = y \implies \text{compatible}(x, y)$
2. **Transitivity**: $\forall P_1, P_2, P_3$ if $\mathcal{S}_{12}$ and $\mathcal{S}_{23}$ then $\mathcal{S}_{13}$
3. **Reflexivity**: $\forall P\ \exists \mathcal{S} : P \to P$

Implementation

In MES, synchronization is implemented using the `Synchronization` type:

```
struct Synchronization
    pattern1::Pattern
    pattern2::Pattern
    morphism::Tuple{String, String, String}
end
```

Example

Here's a simple example of creating and working with synchronization:

```
# Create two patterns to synchronize
pattern1 = Pattern(
    ["A", "B"],
    [("A", "B", "f")]
)

pattern2 = Pattern(
    ["X", "Y"],
    [("X", "Y", "g")]
)

# Create a synchronization between the patterns
sync = Synchronization(
    pattern1,
    pattern2,
    ("A", "X", "sync")
)

# Verify synchronization properties
verify_synchronization(sync)  # Returns true if properties are satisfied
```

Synchronization Mechanisms

Implementation

The synchronization mechanisms are implemented in the code:

```
function verify_synchronization(sync::Synchronization)
    # Check compatibility
    for (src1, tgt1, _) in sync.pattern1.morphisms
        for (src2, tgt2, _) in sync.pattern2.morphisms
            if (src1, src2, _) == sync.morphism
                if !compatible(tgt1, tgt2)
                    return false
                end
            end
        end
    end

    # Check transitivity
    if !verify_transitivity(sync)
        return false
    end

    # Check reflexivity
    if !verify_reflexivity(sync)
        return false
    end

    return true
end
```

```
function compatible(x::String, y::String)
    # Implementation of compatibility check
    return true  # Simplified for example
end

function verify_transitivity(sync::Synchronization)
    # Implementation of transitivity check
    return true  # Simplified for example
end

function verify_reflexivity(sync::Synchronization)
    # Implementation of reflexivity check
    return true  # Simplified for example
end
```

Example

Here's an example of synchronizing neural network components:

```
# Create patterns for different network components
input_pattern = Pattern(
    ["Input", "Hidden"],
    [("Input", "Hidden", "weights1")]
)

output_pattern = Pattern(
    ["Hidden", "Output"],
    [("Hidden", "Output", "weights2")]
)

# Create synchronization between components
nn_sync = Synchronization(
    input_pattern,
    output_pattern,
    ("Hidden", "Hidden", "sync_hidden")
)

# Verify the synchronization
@assert verify_synchronization(nn_sync)
```

Applications

Neural Networks

Synchronization can coordinate neural network components:

```
# Create patterns for a complex neural network
input_pattern = Pattern(
    ["Input", "Hidden1"],
    [("Input", "Hidden1", "w1")]
)

hidden_pattern = Pattern(
    ["Hidden1", "Hidden2"],
    [("Hidden1", "Hidden2", "w2")]
)
```

```
output_pattern = Pattern(
    ["Hidden2", "Output"],
    [("Hidden2", "Output", "w3")]
)

# Create synchronizations between all components
sync1 = Synchronization(input_pattern, hidden_pattern, ("Hidden1", "Hidden1", "sync1"))
sync2 = Synchronization(hidden_pattern, output_pattern, ("Hidden2", "Hidden2", "sync2"))

# Verify all synchronizations
@assert verify_synchronization(sync1)
@assert verify_synchronization(sync2)
```

Social Networks

Synchronization can coordinate social network interactions:

```
# Create patterns for social network components
user_pattern = Pattern(
    ["User", "Post"],
    [("User", "Post", "creates")]
)

post_pattern = Pattern(
    ["Post", "Comment"],
    [("Post", "Comment", "has")]
)

# Create synchronization between components
social_sync = Synchronization(
    user_pattern,
    post_pattern,
    ("Post", "Post", "sync_post")
)

# Verify the synchronization
@assert verify_synchronization(social_sync)
```

For more detailed information about synchronization in MES, refer to the original papers in the Papers section.

Chapter 12

Categorical Accounting

12.1 Kategorische Implementierung der Monetären Makrobuchhaltung

1. Agenten und Konten

Agenten

- **Lab** (Arbeit/Haushalte)
  - Verkauft Arbeit
  - Konsumiert Güter
  - Hält Bankeinlagen
- **Res** (Ressourcen/Rohstofflieferanten)
  - Verkauft Ressourcen
  - Konsumiert Güter
  - Hält Bankeinlagen
- **Com** (Unternehmen/Produzenten)
  - Kauft Arbeit und Ressourcen
  - Produziert Güter
  - Nimmt Kredite auf
  - Zahlt Dividenden
- **Cap** (Kapitalisten)
  - Erhält Dividenden
  - Konsumiert Güter
  - Hält Bankeinlagen
- **Bank** (Banken)
  - Vergibt Kredite
  - Führt Konten

Kontentypen

1. **Nominale Konten** (Geldwerte)
   - Bankkonten (`AccXBank`)
   - Kredite (`AccComLoan`)
   - Dividenden (`AccComDiv`, `AccCapDiv`)
2. **Reale Konten** (Mengen)
   - Güterkonten (`AccXGood`)
   - Ressourcenkonten (`AccComRes`, `AccResRes`)
   - Arbeitskonten (`AccComLab`, `AccLabLab`)

2. Kategorische Struktur

Objekte (CategoryObject)

```
struct Account <: CategoryObject
    name::String    # Kontobezeichnung
    agent::String   # Besitzer
    type::String    # "asset" oder "liability"
    unit::String    # "nominal" oder "real"
    value::Float64  # Kontostand
end
```

Morphismen (CategoryMorphism)

```
struct Flow <: CategoryMorphism
    name::String           # Transaktionsbezeichnung
    source::Account        # Quellkonto
    target::Account        # Zielkonto
    value::Float64        # Transaktionswert
    description::String    # Beschreibung
end
```

Komposition

Die Komposition von Flüssen folgt dem kategorischen Diagramm:

```
A --f--> B --g--> C  =  A --∘gf--> C
```

Implementiert als:

```
compose(f::Flow, g::Flow) = Flow(
    string(f.name, "∘", g.name),
    g.source,
    f.target,
    f.value,
    string("Composition: ", g.description, " followed by ", f.description)
)
```

3. Perspektiven und Funktoren

Mikro-Perspektive

- Individuelle Konten für jeden Agenten
- Detaillierte Flüsse zwischen Konten
- Invarianzen auf Agentenebene

Makro-Perspektive

- Aggregierte Konten (`TotalBank`, `TotalReal`, `TotalNominal`)
- Netto-Flüsse zwischen Sektoren
- Systemweite Invarianzen

Funktor F: Mikro → Makro

Der Funktor bildet ab:

- Objekte: [F(Account*{\text{micro}}) \mapsto Account*{\text{macro}}]
- Morphismen: [F(Flow*{\text{micro}}) \mapsto Flow*{\text{macro}}]

4. Invarianzen und Balancen

Mikro-Invarianzen

1. Bank-Invarianz pro Agent: [InvBank_A = AccABank - AccBankABank = 0]
2. Kredit-Invarianz: [InvLoan = AccBankComLoan - AccComLoan = 0]

Makro-Invarianz

Gesamtsystem-Invarianz: [InvMacro = \sum*A InvBank*A + InvLoan = 0]

5. Haupttransaktionen

M1: Ressourcenkauf

```
ComBank ---> ResBank
ComRes <--- ResRes
```

M3: Lohnzahlung

```
ComBank ---> LabBank
ComLab <--- LabLab
```

M5: Investition

```
ComLoan ---> ComBank
```

M6: Dividendenzahlung

```
ComBank ---> CapBank
ComDiv ---> CapDiv
```

6. Emergente Eigenschaften

Geldschöpfung

- Durch Kreditvergabe (M5)
- Durch Einlagenbildung (M1, M3, M6)

Produktionskreislauf

1. Investition → Ressourcen- und Arbeitskauf
2. Produktion → Güterverkauf
3. Einkommensverteilung → Konsum

Kategorische Eigenschaften

1. Funktor-Erhaltung der Balancen
2. Kompositionserhaltung der Flüsse
3. Invarianz-Erhaltung über Perspektiven

7. Simulationsergebnisse

Die Simulation zeigt:

1. Stabile Invarianzen (nahe 0)
2. Wachsende Kontostände
3. Zyklische Produktions- und Konsummuster
4. Erhaltung der kategorischen Struktur

Die vollständigen Simulationsergebnisse finden sich in:

- `simulation_data/categorical_data_full.csv`
- `simulation_data/categorical_analysis.png`

Part IV

Examples

Chapter 13

Basic Categories

13.1 Basic Categories

This section provides practical examples of working with categories in MES.

Example Usage

```
using MES
```

Creating a Simple Category

```
# Define objects and morphisms
objects = ["A", "B", "C"]
morphisms = Dict(
    "f" => ("A", "B"),
    "g" => ("B", "C")
)

# Create the category
category = create_category(objects, morphisms)
```

Working with Patterns

```
# Create a pattern from the category
pattern_objects = ["A", "B"]
pattern_links = [("A", "B")]
pattern = create_pattern(category, pattern_objects, pattern_links)

# Calculate the colimit
colimit = calculate_colimit(pattern)
```

Next Steps

Check out the Bill of Exchange example for a more complex application.

Chapter 14

Categories

14.1 Category Examples

This chapter provides practical examples of using categories in Memory Evolutive Systems.

Basic Category Creation

Let's start with creating a simple category:

```
using MES

# Create a graph with neurons and connections
graph = create_graph([
    "Neuron1", "Neuron2", "Assembly"
], [
    ("Neuron1", "Neuron2", "synapse"),
    ("Neuron2", "Assembly", "activation")
])

# Convert to category
cat = create_category(graph)

# Add composition
add_composition!(cat, "synapse", "activation", "process")

# Verify category structure
is_valid = verify_category(cat)
```

Biological System Example

Modeling a cell and its components:

```
# Create a configuration category for a cell
cell = configuration_category([
    "Nucleus", "Mitochondria", "CellMembrane"
], [
    ("Nucleus", "Mitochondria", "energy_transfer"),
    ("Mitochondria", "CellMembrane", "transport")
])
```

```
# Model cell division
daughter_cell = configuration_category([
    "DaughterNucleus", "DaughterMitochondria", "DaughterMembrane"
], [
    ("DaughterNucleus", "DaughterMitochondria", "energy_transfer"),
    ("DaughterMitochondria", "DaughterMembrane", "transport")
])

# Create a functor representing division
division = functor(cell, daughter_cell, Dict(
    "Nucleus" => "DaughterNucleus",
    "Mitochondria" => "DaughterMitochondria",
    "CellMembrane" => "DaughterMembrane"
))
```

Neural Network Example

Modeling a simple neural network:

```
# Create a labeled category for the network
network = labeled_category([
    "Input", "Hidden", "Output"
], [
    ("Input", "Hidden", "weight1"),
    ("Hidden", "Output", "weight2")
], ["weight1", "weight2"])

# Add learning rules as compositions
add_composition!(network, "weight1", "weight2", "forward_prop")
add_composition!(network, "weight2", "weight1", "back_prop")
```

Social Network Example

Modeling social relationships:

```
# Create a category with generators
social = category_generators([
    "Person", "Group", "Organization"
], [
    ("Person", "Group", "member"),
    ("Group", "Organization", "affiliate")
])

# Add social interaction rules
add_composition!(social, "member", "affiliate", "participate")
```

Further Reading

For more information about categories in MES, refer to:

- Category Theory
- Patterns
- Memory Systems

Chapter 15

Patterns

15.1 Patterns and Colimits

This page demonstrates how to work with patterns and colimits in MES.

Basic Pattern Creation

```
using MES

# Create a simple pattern
pattern = create_pattern(
    ["A", "B", "C"],
    [("A", "B"), ("B", "C")]
)

# Calculate colimit
colimit = calculate_colimit(pattern)
```

Pattern Recognition

```
# Create a pattern recognition system
system = create_pattern_recognition_system(
    threshold=0.7,
    decay_rate=0.1
)

# Add known patterns
add_pattern!(system, ["X", "Y", "Z"], "Triangle")
add_pattern!(system, ["A", "B"], "Line")

# Recognize patterns
transactions = [
    ("X", "Y", 1.0),
    ("Y", "Z", 1.0),
    ("Z", "X", 1.0)
]

patterns = recognize_patterns(system, transactions)
```

Pattern Evolution

```
# Create a pattern evolution system
evolution = create_pattern_evolution_system()

# Define initial patterns
initial_patterns = [
    Pattern(["A", "B"], "Simple"),
    Pattern(["B", "C"], "Simple"),
    Pattern(["A", "B", "C"], "Complex")
]

# Simulate evolution
results = simulate_evolution(evolution, initial_patterns)
```

Chapter 16

Memory Systems

16.1 Memory Systems

This page demonstrates how to work with memory systems in MES.

Basic Memory Operations

```
using MES

# Create a memory component
memory = create_memory_component(
    capacity=100,
    decay_rate=0.1
)

# Store some patterns
store_pattern!(memory, ["A", "B", "C"], 0.9)
store_pattern!(memory, ["B", "C", "D"], 0.8)

# Retrieve patterns
retrieved = retrieve_pattern(memory, ["A", "B", "C"])
```

Memory Decay

```
# Create a memory with decay
memory = create_memory_component(
    capacity=5,
    decay_rate=0.2
)

# Store patterns at different times
store_pattern!(memory, ["X", "Y"], 1.0, Dates.now())
store_pattern!(memory, ["Y", "Z"], 0.9, Dates.now() - Dates.Second(10))

# Simulate decay
decay_memory!(memory)
```

Memory Consolidation

```
# Create short-term and long-term memory
stm = create_memory_component(5, 0.3)  # higher decay rate
```

```
ltm = create_memory_component(20, 0.05)  # lower decay rate

# Store in STM
store_pattern!(stm, ["A", "B"], 0.9)
store_pattern!(stm, ["B", "C"], 0.85)

# Consolidate to LTM
consolidate_memory!(stm, ltm, 0.8)  # threshold 0.8
```

Pattern Completion

```
# Create a completion system
system = create_completion_system(0.6)  # completion threshold

# Add known patterns
add_known_pattern!(system, ["1", "2", "3", "4"], 1.0)
add_known_pattern!(system, ["2", "3", "4", "5"], 1.0)

# Try to complete partial pattern
partial = create_pattern(["1", "2"], 0.9)
completed = complete_pattern(system, partial)
```

Chapter 17

Synchronization

17.1 Synchronization

This page demonstrates how to work with synchronization in MES.

Basic Synchronization

```
using MES

# Create a network of oscillators
oscillators = [
    create_oscillator(1.0, 0.1),  # frequency 1.0, coupling strength 0.1
    create_oscillator(1.1, 0.1),
    create_oscillator(0.9, 0.1)
]

# Create connections
connections = [
    (1, 2, 0.2),  # oscillator 1 to 2 with weight 0.2
    (2, 3, 0.2),
    (3, 1, 0.2)
]

# Run synchronization
history = simulate_synchronization(oscillators, connections, 100)
```

Phase Locking

```
# Create two oscillators
osc1 = create_oscillator(1.0, 0.2)
osc2 = create_oscillator(1.0, 0.2)

# Set initial phases
set_phase!(osc1, 0.0)
set_phase!(osc2, π/4)

# Run simulation
phases = []
for t in 1:100
    update_oscillators!([osc1, osc2], [(1, 2, 0.3)])
    push!(phases, (get_phase(osc1), get_phase(osc2)))
```

```
end

# Calculate phase locking value
plv = calculate_plv(phases)
```

Co-regulation

```
# Create co-regulation network
network = create_co_regulation_network(
    3,      # number of nodes
    0.1,    # coupling strength
    0.05    # noise level
)

# Set initial states
states = Dict(
    1 => 0.8,
    2 => 0.6,
    3 => 0.4
)

# Run co-regulation
history = simulate_co_regulation(network, states, 50)
```

Entrainment

```
# Create driver and responder oscillators
driver = create_oscillator(1.0, 0.3)    # strong oscillator
responder = create_oscillator(1.2, 0.1)  # weak oscillator

# Set up unidirectional coupling
coupling = [(1, 2, 0.4)]  # driver -> responder

# Run entrainment simulation
frequencies = []
for t in 1:200
    update_oscillators!([driver, responder], coupling)
    push!(frequencies, (get_frequency(driver), get_frequency(responder)))
end

# Calculate entrainment measure
entrainment = calculate_entrainment(frequencies)
```

Chapter 18

Bill of Exchange

18.1 Bill of Exchange System

This example demonstrates a complete bill of exchange (BOE) transaction in a double-entry system with multiple agents, including discounting and transaction logging.

System Components

Types and Structures

```
struct Transaction
    id::String
    timestamp::DateTime
    description::String
    amount::Float64
    from_account::String
    to_account::String
    discount_rate::Float64
    maturity_date::DateTime
    present_value::Float64
end

struct TransactionLog
    transactions::Vector{Transaction}
end
```

Discounting Functions

```
function calculate_present_value(future_value::Float64, rate::Float64, time_to_maturity::Float64)
    return future_value / (1 + rate * time_to_maturity)
end

function calculate_discount(future_value::Float64, present_value::Float64)
    return future_value - present_value
end
```

Agents and Accounts

1. Seller (S)

- Products (inventory)

- Receivables
- Bank account at bankS
- Discount account

2. Buyer (B)

- Products (inventory)
- Payables
- Bank account at bankB
- Discount account

3. Seller's Bank (bankS)

- Assets
- Liabilities
- Reserves at CB
- Discount account

4. Buyer's Bank (bankB)

- Assets
- Liabilities
- Reserves at CB
- Discount account

5. Central Bank (CB)

- Assets
- Liabilities (reserves)
- Money supply
- Discount account

Transaction Flow with Discounting

1. BOE Creation

```
# Create receivables and payables with present value
pv = calculate_present_value(initial_amount, rate, maturity_months/12)
"create_boe" => (
    ("S_receivables", "B_payables"),
    (amount, rate, time) -> calculate_present_value(amount, rate, time)
)
```

2. Bank Acceptance with Discounting

```
# Bank accepts BOE with discount calculation
pv, discount = calculate_present_value(initial_amount, rate, maturity_months/12),
               calculate_discount(initial_amount, pv)
"bankS_accept" => (
    ("bankS_liabilities", "S_receivables"),
    (amount, rate, time) -> begin
        pv = calculate_present_value(amount, rate, time)
        discount = calculate_discount(amount, pv)
        (pv, discount)
    end
)
```

3. Monthly Transfers

```
# Monthly transfers with present value calculation
transfer_amount = initial_amount / maturity_months
pv = calculate_present_value(transfer_amount, rate, (maturity_months - i)/12)
```

4. Final Settlement

```
# Settlement with present value at maturity
pv = calculate_present_value(initial_amount, rate, 0)
```

Running the Example

```
using MES

# Run example with default parameters
final_state, transaction_log = run_boe_example(
    initial_amount = 10000.0,  # BOE amount
    rate = 0.05,               # Discount rate
    maturity_months = 6        # Time to maturity
)

# Print transaction log
println("\nTransaction Log:")
for t in transaction_log.transactions
    println("$(t.timestamp): $(t.description)")
    println("  Amount: $(t.amount)")
    println("  Present Value: $(t.present_value)")
    println("  From: $(t.from_account)")
    println("  To: $(t.to_account)")
    println("  Rate: $(t.discount_rate)")
    println("  Maturity: $(t.maturity_date)")
    println()
end
```

MES Interpretation

This example demonstrates how MES can model complex financial transactions with:

1. Category Structure

- Objects represent accounts

- Morphisms represent transactions with lambda functions for calculations
- Patterns represent transaction sequences

2. Time Evolution

- Transactions are timestamped
- Present values are calculated based on time to maturity
- Multiple transfers over the 6-month period

3. Documentation and Logging

- Each transaction is logged with full details
- Present values and discounts are tracked
- Transaction sequence is preserved

4. Double-Entry System

- All transactions maintain double-entry principles
- Discount accounts track value changes
- CB operations are properly recorded

Next Steps

Check out the Theory section for mathematical details about categories and patterns in MES.

Chapter 19

MOMA System

19.1 MOMAT Examples

This section contains examples of using MES for Monetary Macro Accounting (MOMAT) modeling. These examples demonstrate how to apply the Memory Evolutive Systems framework to model monetary and financial systems.

Basic Examples

- Simple Monetary Circuit - Basic example of modeling a simple monetary circuit
- Bank Balance Sheets - Modeling bank balance sheets and their evolution
- Payment Systems - Modeling payment system interactions

Advanced Examples

- Complex Financial Networks - Modeling complex financial networks with multiple banks
- Systemic Risk Analysis - Analyzing systemic risk in financial networks
- Monetary Policy Effects - Modeling the effects of monetary policy

Theory

The MOMAT examples are based on the following theoretical foundations:

1. **Monetary Circuit Theory**
   - Bank money creation
   - Payment system dynamics
   - Balance sheet constraints
2. **Network Theory**
   - Interbank networks
   - Payment flows
   - Systemic risk

3. **Memory Systems**
    - Financial memory
    - Credit history
    - Market expectations

References

For more information on the theoretical foundations, see:

- Memory Evolutive Systems Book
- MES 2023 Paper

Chapter 20

National Accounting

20.1 National Accounting System

This document describes the categorical structure of our national accounting system.

Category Theoretical Structure

The national accounting system is modeled as a category $\mathcal{C}$ where:

- Objects are accounts and economic agents
- Morphisms are financial flows and transactions
- Composition represents transaction chains
- Identity morphisms represent account preservation

The basic categorical structure is given by:

$$\begin{aligned} \mathcal{C} &= (\mathrm{Obj}(\mathcal{C}), \mathrm{Mor}(\mathcal{C}), \circ, \mathrm{id}) \\ \mathrm{Obj}(\mathcal{C}) &= \{\textsf{Accounts}\} \cup \{\textsf{Agents}\} \\ \mathrm{Mor}(\mathcal{C}) &= \{f : A \to B \mid A, B \in \mathrm{Obj}(\mathcal{C})\} \end{aligned}$$

Basic Transaction Structure

The following diagram represents the basic transaction structure:

$$\begin{array}{ccc} \textsf{Account}_1 & \xrightarrow{[r, "f"]} & \textsf{Account}_2 \\ & \underset{[dr, "h"']}{\searrow} & \downarrow{\scriptstyle [d, "g"]} \\ & & \textsf{Account}_3 \end{array}$$

$\textsf{Account}_3$

where the diagram commutes: $g \circ f = h$

Double-Entry Structure

The double-entry principle is represented by the following pullback:

Transaction$[r, \text{"credit"}][d, \text{"debit"}']$CreditAccount$[d, \text{"balance"}]$
DebitAccount$[r, \text{"balance"}']$Money

unt [r, ”balance”’]
Money

Balance Sheet Functor

The balance sheet structure is represented by a functor $B : \mathcal{C} \to \mathbf{Ab}$ where:

$$B(\text{Account}) = \mathbb{R} \text{ (account balance)}$$
$$B(f : A \to B) = (+f_{\text{amount}}) \text{ (transaction amount)}$$

Conservation Laws

The following invariants are maintained:

$$\forall t : \sum_i \text{Balance}_i(t) = \text{Constant}$$
$$\forall a : \sum_{i \in \text{Accounts}(a)} \text{Flow}_i(t) = 0$$

Implementation

The implementation in Julia follows these categorical structures:

```
struct Transaction
    debit::Account
    credit::Account
    amount::Real
end
```

Natural Transformations

Account transformations follow the pattern:

$A[r, \text{"}f\text{"}][d, \text{"}\alpha_A\text{"}']B[d, \text{"}\alpha_B\text{"}]$
$A'[r, \text{"}f'\text{"}']B'$

B'

where α_A and α_B are natural transformations satisfying:

$$\alpha_B \circ f = f' \circ \alpha_A$$

Analysis Tools

Balance Sheet Evolution

The evolution of balance sheets follows:

$$\frac{d}{dt}Balance(A) = \sum_{f \in Inflows(A)} f - \sum_{f \in Outflows(A)} f$$

Flow Analysis

Flow patterns are analyzed through:

$$Flow_{AB}(t) = \sum_{f:A \to B} f_{amount}(t)$$

Invariance Measures

System stability is measured by:

$$\delta(t) = \left| \sum_{a \in Agents} \sum_{i \in Accounts(a)} Balance_i(t) - TotalMoney \right|$$

Overview

The system models the following key components:

- Multiple economic agents (capitalist, company, labor, resource, bank)
- Bank accounts and their balances
- Production, income, and financial circuits
- Micro and macro balance relationships

Implementation

The implementation can be found in `momascf.jl` and `momascf_categorical.jl` in the `examples/national_accounting` directory.

Agents and Their Accounts

The system includes several economic agents, each with their own accounts:

```
# Example agent definitions
agents = [:capitalist, :company, :labor, :resource, :bank]
```

Categorical Balance Structure

The system maintains balance at multiple levels:

1. Micro level: Individual agent account balances
2. Macro level: System-wide invariances
3. Compositional level: Relationships between micro and macro balances

Transactions and Flows

Key transactions in the system include:

- Wage payments
- Resource purchases
- Investment flows
- Financial transactions

Running the Example

To run the example:

```
using MES
include("examples/national_accounting/momascf.jl")

# Run simulation
sim = simulate(StateTransition, State(Parameters=Pars), 100)
```

Analysis

The simulation generates data that can be analyzed to understand:

- Balance sheet evolution
- Flow patterns
- Invariance preservation
- Categorical relationships

Categorical Implementation

A categorical implementation of this system is available in `momascf_categorical.jl`, which demonstrates:

- Category theoretical structures
- Functorial relationships
- Natural transformations
- Preservation of balances through categorical constructions

Advanced Categorical Structures

T-Algebra Structure

The transaction system forms a T-algebra where T is an endofunctor on the category of accounts:

```
T(Accounts) \arrow[r, "T(f)"] \arrow[d, "a"'] & T(Accounts') \arrow[d, "b"] \\
Accounts \arrow[r, "f"'] & Accounts'
```

Double-Entry Structure

The double-entry principle is represented by the following pullback:

```
Transaction \arrow[r, "credit"] \arrow[d, "debit"'] & CreditAccount \arrow[d, "balance"] \\
DebitAccount \arrow[r, "balance"'] & Money
```

Natural Transformations

Account transformations are represented by natural transformations:

```
A \arrow[r, "f"] \arrow[d, "\alpha_A"'] & B \arrow[d, "\alpha_B"] \\
A' \arrow[r, "f'"'] & B'
```

Complex Transaction Networks

Transaction networks form complex categorical structures:

[\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "h"'] \arrow[rd, "k"] & B \arrow[d, "g"] \arrow[rd, "m"] \ C \arrow[r, "i"'] & D \arrow[r, "j"'] & E \end{tikzcd}]

With invariance conditions: [\begin{align*} j \circ i \circ h &= j \circ g \circ f \ m \circ f &= j \circ k \end{align*}]

Conservation Laws

The conservation of money is represented by a limit:

[\begin{tikzcd} L \arrow[r] \arrow[d] & \prod*i A*i \arrow[d] \ \prod*j B*j \arrow[r] & \prod*{i,j} C*{ij} \end{tikzcd}]

Multiplicity Principle

Different transaction patterns can lead to the same net effect:

[\begin{tikzcd} P \arrow[r] \arrow[d] & colim(P) \arrow[d, "\cong"] \ Q \arrow[r] & colim(Q) \end{tikzcd}]

Figure 20.1: Double Entry Structure

Balance Preservation

Balance preservation is represented by the natural transformation:

[\begin{tikzcd} \cat{C}(t) \ar[r, "F*t"] \ar[d, "\beta*t"'] & \cat{C}(t+1) \ar[d, "\beta_{t+1}"] \\ \mathbf{Ab} \ar[r, "\id"'] & \mathbf{Ab} \end{tikzcd}]

Hierarchical Structure

The system forms a hierarchy of categories:

[\begin{tikzcd} C*0 \ar[r, "F*1"] & C*1 \ar[r, "F*2"] & C*2 \ar[r, "F*3"] & \cdots \end{tikzcd}]

Invariance Measures

The system's stability is measured through various invariants:

$$\delta_{\text{local}}(a) = \left| \sum_{i \in Accounts(a)} Balance_i \right|$$

$$\delta_{\text{global}} = \left| \sum_{a \in Agents} \delta_{\text{local}}(a) \right|$$

$$\delta_{\text{flow}}(t) = \left| \sum_{f \in Flows(t)} f_{\text{amount}} \right|$$

Categorical Diagrams

The following diagrams were created using quiver.app, a specialized tool for creating categorical diagrams. Each diagram is exported as SVG for high-quality rendering.

Double-Entry Structure

The double-entry principle is represented by the following categorical diagram:

This diagram shows the fundamental structure of double-entry bookkeeping as a pullback in the category of accounts. The Transaction object represents a single bookkeeping entry that simultaneously affects both the debit and credit accounts while preserving the balance invariant.

Creating New Diagrams

To add new categorical diagrams to this documentation:

1. Visit quiver.app
2. Create your diagram using the visual editor
3. Export as SVG
4. Save the SVG file in `docs/src/assets/images/`

Figure 20.2: Basic Transaction Structure

Figure 20.3: Natural Transformation

Figure 20.4: Complex Transaction Network

5. Include in the documentation using Markdown image syntax:

   ```
   ![Diagram Name](../../assets/images/filename.svg)
   ```

Diagram Style Guide

For consistency across diagrams:

- Use standard categorical notation (→ for morphisms, ⇒ for natural transformations)
- Label all morphisms and objects clearly
- Use consistent sizing (400x300 for simple diagrams, larger for complex ones)
- Include explanatory text below each diagram

Basic Transaction Structure

The basic transaction structure is represented by the following categorical diagram:

Where the diagram commutes: (g \circ f = h)

Natural Transformations

Account transformations are represented by the following natural transformation diagram:

Where (\alpha*A) and (\alpha*B) are natural transformations satisfying: [\alpha*B *circ f = f' *circ *alpha*A]

Complex Transaction Networks

Transaction networks form complex categorical structures as shown in this diagram:

With invariance conditions: [\begin{align*} j *circ i *circ h &= j *circ g *circ f \ m *circ f &= j *circ k *end{*align*}]

Balance Preservation

Balance preservation is represented by the following natural transformation:

This diagram shows how the balance functor commutes with time evolution.

Figure 20.5: Balance Preservation

Part V

Papers

Part VI

Related Papers and Publications

This page lists key papers and publications related to Memory Evolutive Systems (MES).

Chapter 21

Core Publications

21.1 Memory Evolutive Systems: Hierarchy, Emergence, Cognition (2007)

Authors: Andrée C. Ehresmann and Jean-Paul Vanbremeersch **Publisher**: Elsevier Science **ISBN**: 978-0444522443

This foundational book introduces the mathematical theory of Memory Evolutive Systems. It covers:

- Category theory foundations
- Hierarchical categories and complexity levels
- Memory organization and dynamics
- Cognitive systems and emergence
- Applications to biological and social systems

21.2 Memory Evolutive Neural Networks (2023)

Authors: Andrée C. Ehresmann **Journal**: Applied Sciences **DOI**: 10.3390/app13095638

This paper extends MES theory to neural networks, discussing:

- Neural category theory
- Hierarchical neural architectures
- Memory formation and recall
- Learning and adaptation mechanisms
- Applications to deep learning

Chapter 22

Implementation Papers

22.1 MES.jl: A Julia Implementation of Memory Evolutive Systems

Status: In preparation **Expected Publication**: 2024

This paper will describe:

- The Julia implementation of MES theory
- Core algorithms and data structures
- Performance optimizations
- Example applications
- Future development directions

Chapter 23

Related Work

23.1 Category Theory in Neural Systems (2019)

Authors: Various **Journal**: Frontiers in Applied Mathematics and Statistics **DOI**: 10.3389/fams.2019.00040

Discusses applications of category theory to neural systems, including:

- Neural architectures as categories
- Functorial relationships
- Memory and learning
- Connections to MES theory

23.2 Categorical Models of Memory and Learning (2021)

Authors: Various **Journal**: Journal of Applied Category Theory **DOI**: [Example DOI]

Explores categorical models in cognitive science:

- Memory representation
- Learning processes
- Hierarchical structures
- Comparisons with MES

Chapter 24

Future Directions

Current research directions in MES theory include:

1. Applications to artificial general intelligence
2. Integration with modern deep learning architectures
3. Biological neural network modeling
4. Social network analysis
5. Complex systems theory

For the latest updates and research, follow:

- MES Research Group
- Category Theory in Cognitive Science
- Neural Categories Workshop Series

Part VII

API Reference

Part VIII

API Reference

This page documents the public API of the Memory Evolutive Systems (MES) package.

Chapter 25

Categories

MES.Category – Type.

```
Category{T}
```

A type representing a category in category theory.

Fields

- `objects::Vector{T}`: The objects in the category
- `morphisms::Dict{Tuple{T,T},Vector{T}}`: Maps pairs of objects to their morphisms
- `composition::Dict{Tuple{T,T,T},Bool}`: Maps triples of objects to composed morphisms

source

MES.create_category – Function.

```
create_category(objects::Vector{T}, morphisms::Dict{Tuple{T,T},Vector{T}}) where T
```

Create a category with the given objects and morphisms.

Arguments

- `objects::Vector{T}`: A vector of objects in the category
- `morphisms::Dict{Tuple{T,T},Vector{T}}`: A dictionary mapping pairs of objects to their morphisms

Returns

A `Category` struct representing the category.

Examples

```
objects = ["A", "B", "C"]
morphisms = Dict(
    ("A", "B") => ["f"],
    ("B", "C") => ["g"]
)
category = create_category(objects, morphisms)
```

source

MES.verify_category – Function.

```
verify_category(category::Category{T}) where T
```

Verify that a category satisfies the basic axioms.

Arguments

- `category::Category{T}`: The category to verify

Returns

A boolean indicating whether the category is valid.

source

MES.verify_composition_closure – Function.

```
verify_composition_closure(category::Category{T}) where T
```

Verify that the category is closed under composition.

Arguments

- `category::Category{T}`: The category to verify

Returns

A boolean indicating whether the category is closed under composition.

source

MES.verify_identity_existence – Function.

```
verify_identity_existence(category::Category{T}) where T
```

Verify that each object has an identity morphism.

Arguments

- `category::Category{T}`: The category to verify

Returns

A boolean indicating whether each object has an identity morphism.

source

Chapter 26

Patterns

MES.Pattern – Type.

```
Pattern
```

A type representing a pattern in a category.

Fields

- `category::Category`: The category containing the pattern
- `objects::Vector{String}`: The objects in the pattern
- `links::Vector{Tuple{String,String}}`: The morphisms between objects in the pattern

source

MES.create_pattern – Function.

```
create_pattern(category::Category, objects::Vector{String}, links::Vector{Tuple{String,String}})
```

Create a pattern in a category.

Arguments

- `category::Category`: The category containing the pattern
- `objects::Vector{String}`: The objects in the pattern
- `links::Vector{Tuple{String,String}}`: The morphisms between objects in the pattern

Returns

A `Pattern` struct representing the pattern.

Examples

```
category = create_category(["A", "B", "C"], Dict())
pattern = create_pattern(category, ["A", "B"], [("A", "B")])
```

source

MES.calculate_colimit – Function.

```
calculate_colimit(pattern::Pattern)
```

Calculate the colimit of a pattern in a category.

Arguments

- `pattern::Pattern`: The pattern to calculate the colimit for

Returns

A dictionary containing:

- `:object`: The colimit object
- `:morphisms`: A dictionary mapping pattern objects to morphisms to the colimit

Examples

```
category = create_category(["A", "B", "C"], Dict())
pattern = create_pattern(category, ["A", "B"], [("A", "B")])
colimit = calculate_colimit(pattern)
```

source

Chapter 27

Memory Systems

MES.MemorySystem – Type.

```
MemorySystem
```

A type representing a memory system in a category.

Fields

- `category::Category`: The category containing the memory system
- `memory::Dict{String,Any}`: The memory storage

source

MES.store! – Function.

```
store!(system::MemorySystem, key::String, value::Any)
```

Store a value in the memory system.

Arguments

- `system::MemorySystem`: The memory system to store in
- `key::String`: The key to store the value under
- `value::Any`: The value to store

source

MES.retrieve – Function.

```
retrieve(system::MemorySystem, key::String)
```

Retrieve a value from the memory system.

Arguments

- `system::MemorySystem`: The memory system to retrieve from
- `key::String`: The key to retrieve

Returns

The value stored under the key, or nothing if not found.

source

MES.verify_memory_system – Function.

```
verify_memory_system(system::MemorySystem)
```

Verify that a memory system is valid.

Arguments

- `system::MemorySystem`: The memory system to verify

Returns

A boolean indicating whether the memory system is valid.

source

Chapter 28

Synchronization

MES.Synchronization – Type.

```
Synchronization
```

A type representing a synchronization between two patterns.

Fields

- `source_pattern::Pattern`: The source pattern
- `target_pattern::Pattern`: The target pattern

source

MES.verify_synchronization – Function.

```
verify_synchronization(sync::Synchronization)
```

Verify that a synchronization is valid.

Arguments

- `sync::Synchronization`: The synchronization to verify

Returns

A boolean indicating whether the synchronization is valid.

source