The Journal of Open Source Software

AlgebraicAgents.jl: Hierarchical Composition of Multi-Formalism Dynamical Systems

Jan Bíma[1], **Sean L. Wu**[2], **and Otto Ritter**[1]

1 DSSI Decision Science, MSD Czech Republic **2** DSSI Decision Science, Merck & Co., Inc., Rahway, NJ, USA

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Mehmet Hakan Satman

Reviewers:

- @SergeStinckwich
- @zhenwu0728

Submitted: 18 January 2026
Published: unpublished

Summary

AlgebraicAgents.jl is a Julia framework for hierarchical dynamical systems modeling that treats formalism as a per-node choice rather than a global constraint. Differential equations, discrete-event systems, and agent-based models coexist within a single hierarchy, coupled through a minimal stepping interface. The framework supports annotation of information flows between components, visualizing model architecture, and querying relationship structure. Native wrappers for Julia's scientific computing ecosystem enable practitioners to compose domain-specific models into representations of complex dynamics.

Statement of Need

Modeling dynamical systems at scale—enterprises with interacting business units, multi-physics engineering systems, coupled socioeconomic networks—requires composing components naturally expressed in different formalisms and operating at different temporal granularities. Continuous dynamics govern reactors, discrete events drive logistics, and agent-based rules capture decisions. These subsystems must integrate while remaining independently developable, often beginning as mockups and refined iteratively as data arrives.

This challenge decomposes into three sub-problems:

- **Multi-formalism coupling.** Subsystems should be expressible in different formalisms, such as ODEs, discrete-time systems, or agent-based models, at varying granularity, with the framework facilitating their coupling.
- **Hierarchical modularity.** Subsystems should support independent development, validation, and reuse as building blocks within larger models.
- **Semantic transparency.** Visualizing and querying information flows across models should support both validation and explainability.

State of the Field

Prior work addresses aspects of this challenge. The Functional Mock-up Interface (Blochwitz et al., 2011) standardizes co-simulation of black-box models but imposes protocol overhead suited to industrial interoperability rather than rapid prototyping. Meta-modeling frameworks like the Generic Modeling Environment (Ledeczi et al., 2001) operate at a higher abstraction, enabling construction of domain-specific formalisms. The Ptolemy project (Eker et al., 2003) and Lingua Franca (Menard et al., 2023) provide principled foundations for heterogeneous component interaction across concurrent, real-time, and distributed settings.

Within the Julia ecosystem (Bezanson et al., 2017), ModelingToolkit.jl (Ma et al., 2021; Rackauckas & Nie, 2017), and its commercial extension, JuliaHub's Dyad, excel at symbolicnumeric modeling and equation-based composition but do not naturally accommodate discrete or agent-based dynamics. AlgebraicDynamics.jl (Baez et al., 2023; Brown et al., 2022) brings categorical semantics to dynamical systems yet enforces strict interface typing that constrains exploratory work.

AlgebraicAgents.jl relaxes formalism and shifts focus towards compositional flexibility. Any agent can access any other agent's state, and synchronization is temporal rather than type-enforced. This design prioritizes iteration speed and introspection over type-enforced interface contracts, a trade-off suited to exploratory modeling, where specifications evolve alongside understanding.

Software Design

The central abstraction of the framework is the *agent*, which serves a dual role. First, an agent implements a dynamical system with a custom evolution rule, exposing its internal clock and state variables as observable quantities to other agents. Second, an agent acts as a node in a rooted tree hierarchy, serving as a container for nested child agents, each of which is itself an agent with this dual character.

Agents are implemented as Julia structures that subtype `AbstractAlgebraicAgent`. The `@aagent` macro provides a lightweight inheritance mechanism, automatically including common interface fields while permitting user-defined fields:

```
@aagent struct InventoryAgent
    stock_level::Int
    reorder_time::Int
end
```

Synchronized Evolution

Each agent implements `_step!`, which advances its state and returns the time on its internal clock, that is, the furthest point for which its trajectory has been computed. The simulation loop coordinates agents by identifying the minimum projected time across the hierarchy and stepping only those agents at that frontier.

This mechanism is exemplified below for the case of three agents A, B, and C.

Global Step	Agent A (Δt=1)	Agent B (Δt=1.5)	Agent C (Δt=3)	Frontier (min)	Stepped
0 (init)	0	0	0	0	—
1	**1**	**1.5**	**3**	1	A, B, C
2	**2**	1.5	3	1.5	A
3	2	**3**	3	2	B
4	**3**	3	3	3	A
5	**4**	**4.5**	**6**	4	A, B, C

Formally, the single step of the simulator is defined as follows.

```
function step!(a::Agent, t=projected_to(a))
    # Recurse depth-first.
    t_min = minimum(step!(c, t) for c in children(a); init=nothing)

    # Step only agents at the time frontier.
    projected_to(a) == t && _step!(a)
```

```julia
    return something(t_min, projected_to(a))
end
```

For models where an agent with a shorter step queries another agent already projected further ahead, *observables* mitigate temporal inconsistency. Agents expose state variables through `gettimeobservable`, which can implement interpolation or extrapolation logic.

We note that during the evolutionary step, any agent in the system can be accessed and modified.

Beyond evolutionary stepping, the framework supports three callback types:

Callback Type	Execution Timing	Typical Use Case
Futures	At predetermined time	Scheduled events, delayed triggers
Controls	Every solver step	Invariant enforcement, monitoring
Instantaneous	Within current step	Intra-step coordination, priority-ordered effects

Topology, Wires, and Relations

Each agent occupies a node in a tree topology. Path-based references enable navigation: `getagent(a, "../sibling/child")`. Container agents with trivial evolution organize subsystems into logical compartments, enabling modular development and hierarchical visualization.

Beyond the execution hierarchy, *wires* explicitly declare directed information flows between agents. While agents can programmatically access any other agent's state, declared wires serve as documentation and enable dependency analysis:

```julia
add_wire!(full_system;
              from=factory_a_main, to=inventory_central_main,
              from_var_name="output_volume", to_var_name="incoming_stock")
```

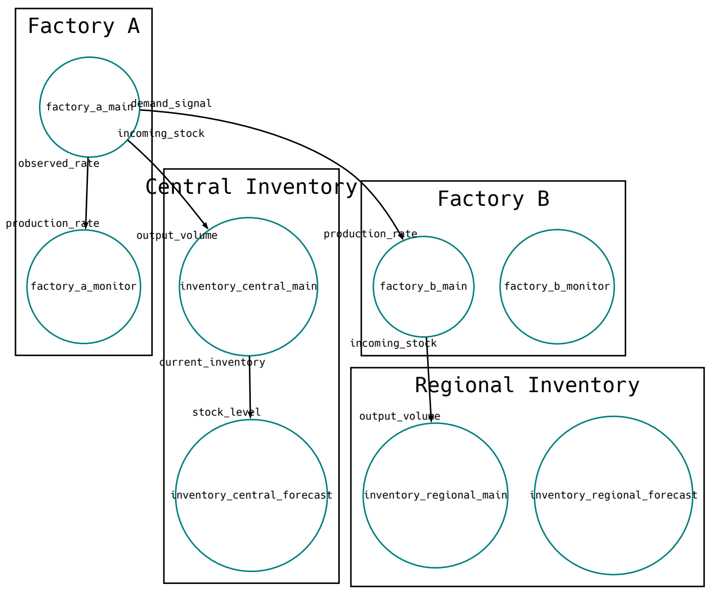

Figure 1: Visualization of agent hierarchy and information flows. Grey dashed arrows represent parent-to-child edges and solid black arrows represent information flows. Black rectangles group related agents.

Concepts represent atemporal notions—resources, constraints, abstractions—that participate in relations alongside agents. This enables modeling of "what" (materials, approvals, markets) separate from "how" (processes), supporting dependency queries and ontological visualization. Both agents and concepts belong to the union type `RelatableType`, enabling *relations* to connect any pair with typed labels:

```
c_finished_good = Concept("FinishedGood", Dict(:type => "product"))
add_relation!(factory_a, c_finished_good, :produces)
```

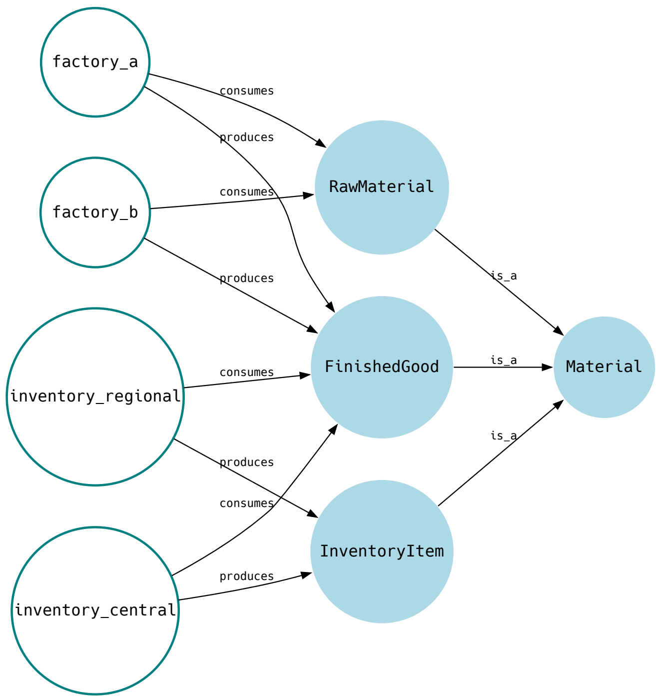

Figure 2: Diagram of concepts and relations. Blue filled circles denote Concepts and green outlined circles denote Agents. Directed arrows indicate relations between two relatable types.

These semantic annotations support Graphviz visualization and structured queries over model architecture.

Research Impact Statement

Integrations

AlgebraicAgents.jl provides native wrappers for Julia's scientific modeling ecosystem. `DiffEqAgent` wraps `DEProblem` instances from DifferentialEquations.jl, enabling ODEs, SDEs, DDEs, and DAEs to participate in hierarchical simulations. Integration with Agents.jl (Datseris et al., 2022) allows agent-based models to compose with continuous or discrete dynamical systems. `GraphicalAgent` wraps `AbstractResourceSharer` or `AbstractMachine` from AlgebraicDynamics.jl, providing compatibility with categorical composition patterns. Visualization functions generate Graphviz DOT and Mermaid diagram output for documentation.

Availability and Documentation

AlgebraicAgents.jl is registered in Julia's General registry. API documentation and a comprehensive example—a synthetic pharmaceutical company model—are available online. Contributions welcome via GitHub. The framework is licensed as MIT license.

Applications and Value Modeling Ecosystem

The framework has been applied to develop proprietary pharmaceutical value chain models. Two companion packages by the authors extend this foundation: ReactiveDynamics.jl, providing chemical reaction network–inspired syntax for business process modeling natively compatible with AlgebraicAgents.jl, and CEEDesigns.jl, implementing Bayesian cost-efficient experimental design for drug discovery.

AI Usage Disclosure

The authors used GitHub Copilot for inline code suggestions and Claude Opus 4.5 (Anthropic) for editorial refinement of the manuscript. The authors reviewed and validated all AI-suggested edits and bear full responsibility for the final work.

References

Baez, J., Li, X., Libkind, S., Osgood, N. D., & Patterson, E. (2023). Compositional modeling with stock and flow diagrams. *Electronic Proceedings in Theoretical Computer Science*, *380*, 77–96. https://doi.org/10.4204/eptcs.380.5

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauss, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., Neumerkel, D., Olsson, H., Peetz, J.-V., & Wolf, S. (2011, June). The functional mockup interface for tool independent exchange of simulation models. *Proceedings from the 8th International Modelica Conference, Technical Univeristy, Dresden, Germany*. https://doi.org/10.3384/ecp11063105

Brown, K., Hanks, T., & Fairbanks, J. (2022). *Compositional exploration of combinatorial scientific models*. https://doi.org/10.48550/arXiv.2206.08755

Datseris, G., Vahdati, A. R., & DuBois, T. C. (2022). Agents.jl: A performant and feature-full agent-based modeling software of minimal code complexity. *SIMULATION*, *0*(0), 003754972110688. https://doi.org/10.1177/00375497211068820

Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., & Xiong, Y. (2003). Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, *91*(1), 127–144. https://doi.org/10.1109/jproc.2002.805829

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., & Volgyesi, P. (2001). The generic modeling environment. *Workshop on Intelligent Signal Processing, Budapest, Hungary*, *17*, 2001.

Ma, Y., Gowda, S., Anantharaman, R., Laughman, C., Shah, V., & Rackauckas, C. (2021). *ModelingToolkit: A composable graph transformation system for equation-based modeling*. https://doi.org/10.48550/arXiv.2103.05244

Menard, C., Lohstroh, M., Bateni, S., Chorlian, M., Deng, A., Donovan, P., Fournier, C., Lin, S., Suchert, F., Tanneberger, T., & others. (2023). High-performance deterministic concurrency using lingua franca. *ACM Transactions on Architecture and Code Optimization*, *20*(4), 1–29. https://doi.org/10.1145/3617687

Rackauckas, C., & Nie, Q. (2017). DifferentialEquations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, *5*(1). https://doi.org/10.5334/jors.151