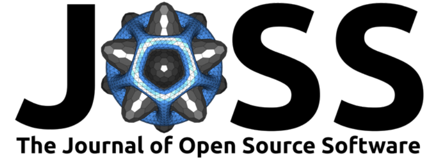


AlgebraicAgents.jl: Hierarchical Composition of Multi-Formalism Dynamical Systems

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: 10.21105/joss.09989

Software

- Review
- Repository
- Archive

Editor: Mehmet Hakan Satman

Reviewers:

- @zhenwu0728
- @mkitti

Submitted: 18 January 2026
Published: 25 June 2026

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

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 annotating 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.

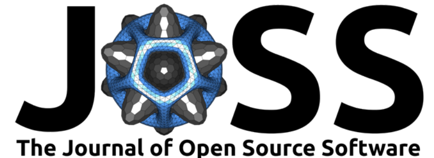
This challenge decomposes into three sub-problems:

- **Multi-formalism coupling.** Subsystems should be expressible in different formalisms, such as ordinary differential equations (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.

AlgebraicAgents.jl is intended for Julia modelers composing heterogeneous dynamical systems across modeling formalisms — users of ecosystems and frameworks such as SciML, Agents.jl, and AlgebraicDynamics.jl, coupling existing models, and practitioners wrapping models built outside these frameworks. Primary applications lie in pharmaceutical value-chain modeling, systems biology, and multi-physics engineering.

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, target symbolic-numeric modeling and equation-based composition. Catalyst.jl (Loman et al., 2023) builds on ModelingToolkit.jl with hierarchical subsystem composition for chemical reaction networks. All require components expressible as equations or as a symbolic intermediate representation, which limits their ability to accommodate discrete or agent-based dynamics. AlgebraicDynamics.jl (Baez et al., 2023; Brown et al., 2022) supplies categorical semantics for dynamical systems but imposes strict interface typing that constrains exploratory work. Agents.jl (Datseris et al., 2024), despite its similar name, solves a complementary problem: It provides a performant runtime for individual agent-based models, whereas AlgebraicAgents.jl coordinates such models alongside other formalisms within a hierarchy.

AlgebraicAgents.jl relaxes these requirements in favor of compositional flexibility. Components are black boxes: No equation, symbolic representation, or typed interface contract is required; an agent need only expose an internal clock and an incremental stepping rule. Any agent can access any other agent's state, and synchronization is temporal rather than type-enforced. The design prioritizes iteration speed and introspection over 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.

We illustrate this mechanism with three agents A, B, and C.

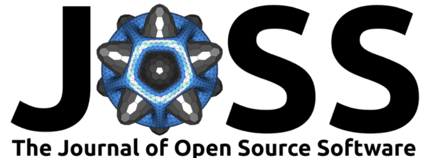


Table 1: Stepping mechanism for three agents with step sizes Δt = 1, 1.5, 3. Columns 2–4 show each agent's projected time after each simulator step. Bold entries mark the agents that advanced in that step. Only agents at the minimum projected time — the frontier — step, and the rest stay at their projected times and are read when queried.

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

Pseudocode for a single simulator step is sketched below.

```
function step!(a::Agent, t=projected_to(a))
    # Recurse depth-first.
    t_min = minimum(
        (step!(c, t) for c in children(a));
        init = projected_to(a),
    )

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

    return min(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 a stepping update, any agent in the system can be accessed and modified.

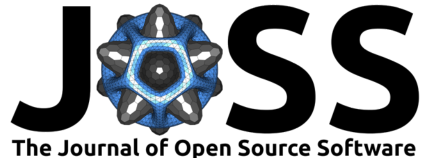
Beyond evolutionary stepping, the framework supports three callback types:

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 containment hierarchy, *wires* explicitly declare directed information flows between agents. While agents can programmatically access any other agent's state, wires promote these dependencies to first-class annotations, enabling visualization and structured queries:




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

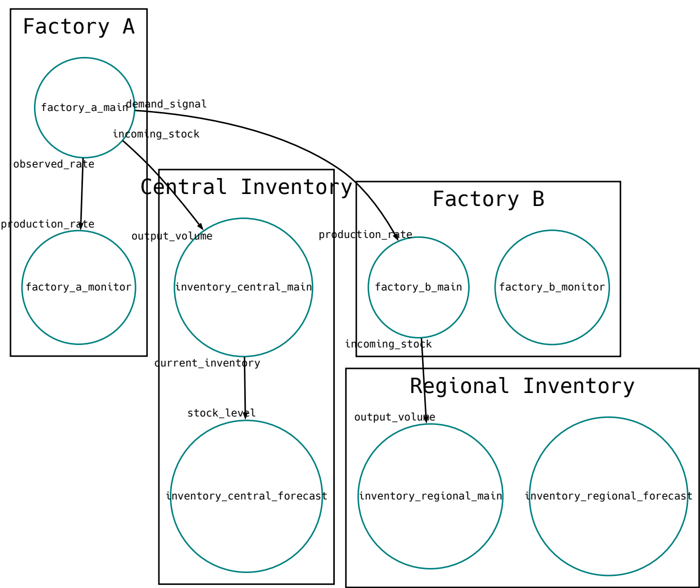
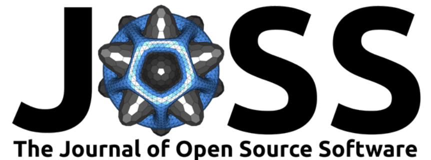


Figure 1: Agent hierarchy and information flows. Teal-outlined circles denote agents, each labeled with its hierarchy path. Black-bordered rectangles group children under their parent. Solid black arrows show declared information flows, each annotated with the source variable on the producing agent and the destination variable on the consuming agent.

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)
```



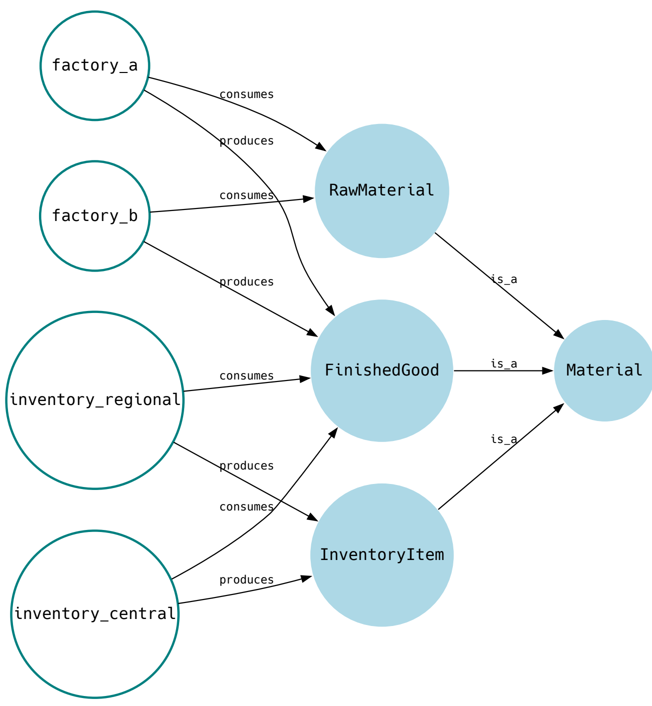


Figure 2: Diagram of concepts and relations. Blue filled circles denote concepts and teal-outlined circles denote agents. Directed arrows indicate typed relations between any two relatable elements; arrow labels give the relation kind (e.g. `produces`, `consumes`, `is_a`).

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

Community Influences

The composability patterns of the framework build on ideas from the AlgebraicJulia ecosystem (Baez et al., 2023; Brown et al., 2022), which applies concepts from applied category theory to scalable dynamical systems modeling through packages such as AlgebraicPetri.jl and AlgebraicDynamics.jl. AlgebraicAgents.jl complements this foundation by extending the compositional perspective to settings where subsystems span multiple modeling formalisms and where interface contracts may not be fully specified during early-stage modeling.

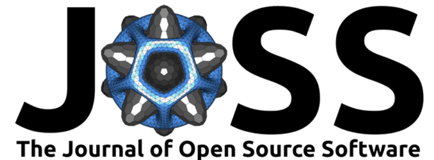
Research Impact Statement

Applications and Value Modeling Ecosystem

The framework was presented at JuliaCon 2023 (Bíma, Ritter, et al., 2023) and has since been applied to develop proprietary pharmaceutical value-chain models. Two companion packages by the authors extend this foundation: ReactiveDynamics.jl (Bíma et al., 2022), providing chemical reaction network–inspired syntax for business process modeling natively compatible with AlgebraicAgents.jl, and CEEDesigns.jl (Bíma, Chen, et al., 2023), implementing Bayesian cost-efficient experimental design for drug discovery.

Third-Party Package Integrations

AlgebraicAgents.jl provides native wrappers for Julia's scientific modeling ecosystem. `DiffEqAgent` wraps DifferentialEquations.jl problems (any `SciMLBase.AbstractDEProblem`),



enabling ODEs, stochastic differential equations (SDEs), delay differential equations (DDEs), and differential-algebraic equations (DAEs) to participate in hierarchical simulations. Integration with Agents.jl (Datseris et al., 2024) 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.

The integration contract is minimal. In general, any third-party dynamical system framework that provides an integrator with a clock and an incremental step can be wrapped as an agent within AlgebraicAgents.jl. These correspond to the two required interface methods, `_step!` and `_projected_to`. Contributors can expose simulation state through these methods to integrate new solver backends; see Contributing.

Availability and Documentation

AlgebraicAgents.jl is MIT-licensed and registered in Julia's General registry; it is installed with Julia's built-in package manager via `Pkg.add("AlgebraicAgents")` (see the Pkg.jl documentation). Documentation includes installation, tutorials, a comprehensive example modeling a synthetic pharmaceutical company, and an API reference. Contributions are welcome via GitHub.

AI Usage Disclosure

The authors used GitHub Copilot for inline code suggestions and Claude Opus 4.5 and 4.7 (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

Bíma, J., Chen, T., Dadashova, K., Jones, G., Ritter, O., & Wu, S. L. (2023). *CEEDesigns.jl: Cost-efficient experimental design in Julia*. https://github.com/Merck/CEEDesigns.jl.

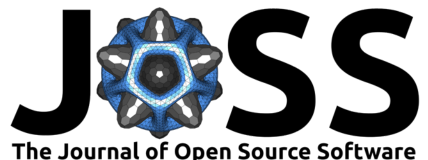
Bíma, J., Ritter, O., & Wu, S. L. (2022). *ReactiveDynamics.jl: A modeling package for reaction network-type dynamical systems*. https://github.com/Merck/ReactiveDynamics.jl.

Bíma, J., Ritter, O., & Wu, S. L. (2023). *DyVE, a framework for value dynamics*. Contributed talk. JuliaCon 2023, Massachusetts Institute of Technology, Cambridge, MA. https://pretalx.com/juliacon2023/talk/DN387J/

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. (2024). Agents.jl: A performant and feature-full agent-based modeling software of minimal code complexity. *SIMULATION*, *100*(10), 1019–1031. 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.

Loman, T. E., Ma, Y., Ilin, V., Gowda, S., Korsbo, N., Yewale, N., Rackauckas, C., & Isaacson, S. A. (2023). Catalyst: Fast and flexible modeling of reaction networks. *PLOS Computational Biology*, *19*(10), 1–19. https://doi.org/10.1371/journal.pcbi.1011530

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