Browser-Based Graphical Interface for JuliaBUGS

Organization: The Julia Language (Turing.jl)
Contributor: Shravan Goswami
Mentors: Xianda Sun, Hong Ge, Markus Hauru

Personal Information

Name: Shravan Goswami
Website: shravangoswami.com
GitHub: @shravanngoswamii
Email: shravanngoswamii@gmail.com
Country: India
Time Zone: GMT +5:30 Hours
University: Uka Tarsadia University, India
Julia Slack: @Shravan Goswami

Personal Background

I am currently a remote Research Assistant at the Machine Learning Group, University of Cambridge, and a pre-final year undergraduate student majoring in Computer Science and Engineering at Uka Tarsadia University, India. My technical expertise spans Julia, Python, JavaScript, and basic R. My journey with open source began two years ago through contributions to Documenter.jl, after which I transitioned to actively contributing within the Turing.jl ecosystem.

My contributions include improving Turing's documentation, website infrastructure, and continuous integration workflows (GitHub Actions). Additionally, I have developed the official website and academic publications for the Machine Learning Group at the University of Cambridge. I've also built multiple web-based projects, developed a cybersecurity-focused Windows desktop application, and continue to actively engage with Julia's open-source community.

A complete list of my projects can be found at shravangoswami.com/projects.

Logistics

I will be fully available throughout the entire duration of the Google Summer of Code timeline. My end-semester examinations are scheduled to conclude in April, allowing me complete availability from the Community Bonding period starting May 8, through to the project completion in September (and beyond, if necessary).

- I will dedicate approximately 25–30 hours per week to ensure timely completion and high-quality outcomes.
- Regular meetings with mentors will be scheduled weekly to discuss progress, receive feedback, and plan subsequent tasks.
- I will actively participate in community forums such as Julia Slack and GitHub discussions to address user feedback and queries promptly.

Abstract

JuliaBUGS is a modern Julia implementation of the classic Bayesian Inference using Gibbs Sampling — BUGS language for Bayesian modeling, combining BUGS's intuitive model specification with Julia's high-performance sampling methods like Hamiltonian Monte Carlo. This project proposes a browser-based graphical model editor for JuliaBUGS, enabling users to visually construct probabilistic graphical models by adding nodes and drawing arrows, instead of writing code. Built using React with TypeScript and React Flow, the web interface will support adding, connecting, grouping, and editing random variables and deterministic nodes via an interactive canvas. Users can specify probability distributions and hyperparameters through intuitive forms. The editor will export model structures to JSON and generate valid JuliaBUGS (or BUGS) code. The GSoC project will focus entirely on delivering a polished, user-friendly frontend with clear usage examples and tutorials. Inspired by tools like DoodleBUGS (for WinBUGS or MultiBUGS), shinystan, and the R Package causact, this editor aims to make model specification more accessible and streamline the Bayesian modeling workflow. Backend integration for inference via Julia will be outlined as future work, with all GSoC deliverables limited to the client side.

Background and Motivation

Probabilistic programming is, but often difficult to learn due to specialized syntax. BUGS (Bayesian inference Using Gibbs Sampling) was one of the first languages to simplify Bayesian modeling for statisticians. Variants like WinBUGS, OpenBUGS, JAGS, Nimble, and MultiBUGS have improved on different aspects, but all still require writing BUGS code. Graphical model notation—drawing nodes and arrows—is a natural way to represent Bayesian networks and inspired tools like DoodleBUGS, which allowed users to visually build models and convert them into BUGS code. Though promising, DoodleBUGS was limited in functionality and tied to early 2000s desktop software.

Modern probabilistic tools like Stan and PyMC3 offer advanced sampling but remain code-heavy. JuliaBUGS is a fresh take, combining the simplicity of BUGS with Julia's strengths—like automatic differentiation and advanced MCMC algorithms. It supports powerful inference (e.g., Hamiltonian Monte Carlo) while keeping a familiar model specification style. This opens the door to making Bayesian modeling more accessible through a graphical frontend.

A drag-and-drop interface for JuliaBUGS would help a wide range of users—from students and educators to domain experts—focus on modeling without dealing with syntax. Even experts often sketch graphical models to structure their thinking; a tool that lets them build and run such models visually would streamline workflows. In educational and industrial contexts, visual modeling could foster better learning and collaboration.

Recent tools like the R package causact and Stan's ShinyStan highlight a growing demand for graphical interfaces in Bayesian workflows. However, a modern, browser-based graphical model builder tightly integrated with a probabilistic language is still missing. This project aims to fill that gap for JuliaBUGS, enabling intuitive creation and sharing of probabilistic graphical models.

Project Scope

This GSoC project focuses solely on frontend development for the JuliaBUGS graphical interface. The aim is to build a fully functional and polished in-browser editor for constructing probabilistic models. Backend or inference-related functionality is out of scope and will only be briefly outlined in future work. Key features and deliverables include:

Interactive Graph Editor (React with TypeScript + React Flow)

A web-based canvas where users can add nodes (random variables or deterministic quantities) and draw arrows to define dependencies. Users will be able to drag, connect, delete, and rearrange nodes. React Flow will handle core interactions like drag-and-drop, zooming, and edge drawing, allowing the focus to remain on model-specific behavior.

Stochastic vs Deterministic Nodes

Nodes will be classified as either stochastic (random variables with distributions) or deterministic (defined by formulas). Each node will have editable properties—stochastic nodes will include a selectable distribution and its parameters, while deterministic nodes will allow custom expressions. A sidebar or form will provide a user-friendly way to configure these details.

Plate Notation (Grouping)

To support repeated or hierarchical structures, users will be able to group nodes into "plates"—rectangular containers representing repeated subgraphs. Plates will include a repetition count (e.g., N), and connections crossing plate boundaries will imply indexed dependencies. This feature is essential for modeling i.i.d. data and hierarchical effects, enabling scalable model design.

Distribution Library UI

A built-in library of common distributions (e.g., Normal, Binomial, Poisson) will be available for stochastic nodes. Users will select a distribution from a dropdown or autocomplete field and provide parameters through form inputs, with validation for required constraints (e.g., positivity). The UI will align with JuliaBUGS's syntax and offer sensible defaults to simplify input.

JSON Export of Model Structure

The editor will export models in a structured JSON format capturing nodes (with type and properties), edges (dependencies), and plates (grouping info). This format will support saving/loading models and lay the groundwork for future backend integration. The JSON schema will be designed to be comprehensive and extensible for long-term use.

By focusing on these core areas, the project will deliver a clean, usable graphical interface for building JuliaBUGS models, setting the stage for broader adoption and future extensions.

Code Generation

A key deliverable is a front-end function that converts the visual model into executable JuliaBUGS code. When users click “Generate Code” or “Export to JuliaBUGS,” the editor will output a BUGS-style model script based on the diagram. For example, a node θ ~ Normal(0,1) and a plate of y[i] ~ Bernoulli(θ) with N observations would generate:

```
model {
  theta ~ dnorm(0, 1)
  for (i in 1:N) {
    y[i] ~ dbern(theta)
  }
}
```

The code generator will traverse the internal graph: declare stochastic nodes, apply deterministic expressions, and handle plates using loops. It will ensure proper ordering (parents before children) and valid naming. Implemented in TypeScript, this module will produce syntactically correct code aligned with JuliaBUGS’s expectations and be tested on simple models for correctness.

User Interface & UX Polish

Creating a clean, intuitive UI is a priority. The editor will feature a toolbar for core actions (e.g., add node, add plate, undo/redo), distinct styling for stochastic and deterministic nodes, and real-time validation. It will guide users in building correct models (e.g., prevent graph cycles, alert on undefined nodes). The UI will support saving/loading models (via JSON), canvas reset, and usability enhancements like keyboard shortcuts, panning, zoom controls, and responsive design. The goal is to deliver a polished, professional interface by the end of GSoC.

Documentation & Tutorial

Comprehensive documentation will accompany the tool, likely hosted via GitHub Pages using Markdown or a static site generator. It will include screenshots, a walkthrough tutorial (e.g., a coin-flip or “eight schools” model), and an explanation of how visual elements map to JuliaBUGS code. At least one full example (with JSON and generated code) will be provided in a public repo to help users and future contributors understand the system and its outputs.

Out of Scope

Server-side functionality, inference algorithms, or live integration with Julia (e.g., running MCMC) are out of scope for GSoC. The project will not implement full client-server architecture or advanced modeling features. Support will be limited to common distributions and core modeling concepts. Backend hooks or API designs may be outlined but not implemented. By keeping the scope focused, the goal is to deliver a robust and usable frontend within the GSoC timeline.

Technical Approach and Stack

Frontend Framework

The web app will be built using React with TypeScript. React's component-based architecture and hook-based state management make it well-suited for handling the dynamic interactions of a graphical model builder. Key state elements—nodes, edges, plates, and selected elements—will be managed via React state, potentially using Context API or a state management library like Redux or Zustand if needed. TypeScript will ensure type safety and clear interfaces for model elements such as nodes, edges, and plates.

React Flow (XyFlow)

React Flow (now XyFlow) is central to the app's architecture. It provides a robust foundation for building interactive node-link diagrams, handling SVG rendering, drag-and-drop, pan/zoom, and connection drawing. It supports custom node components, enabling us to visually differentiate stochastic and deterministic nodes with distinct styling and icons. Plates may be implemented using special nodes or background containers. React Flow's API and event handlers will drive updates to our internal state in response to user interactions.

Testing Frameworks

Testing will cover unit, integration, and end-to-end levels using tools like Jest (for logic), React Testing Library (for component interaction). These tests will ensure model editing, validation, and code generation behave as expected, catching regressions early and providing a safety net for future development.

Interfacing with JuliaBUGS

Though direct integration with JuliaBUGS is out of scope, the generated code will be validated by manually running it in Julia. Simple test models will be loaded into JuliaBUGS to confirm they parse correctly and behave as expected. A frontend-side mapping of distribution names and parameters will be maintained to align with JuliaBUGS syntax.

Performance and Browser Considerations

The app will be a single-page browser-based application. React and React Flow are performant for typical usage (dozens to hundreds of nodes), and optimization techniques like memoization and controlled re-renders will be applied. Cross-browser testing will ensure compatibility across Chrome, Firefox, Safari, and Edge. The layout will be responsive, though primarily optimized for desktop screens due to the nature of diagram editing.

By choosing a modern stack (React + React Flow + TypeScript + UI/testing libraries), the project leverages established tools for reliability, scalability, and community adoption. The selected technologies also align with the contributor's experience, enabling rapid progress from the outset of GSoC.

Timeline

It outlines a detailed week-by-week plan with milestones for midterm and final evaluations. The plan is ambitious yet achievable within the 350-hour limit (~30 hours/week), and includes time for iteration, feedback, and polish.

Community Bonding (May 8 – June 1)

Focus: Preparation and planning.

- Study JuliaBUGS's syntax, DSL quirks, and supported distributions.
- Explore React Flow through small prototypes and documentation.
- Finalize core requirements with mentors (distributions, plates, etc.).
- Design the JSON schema and draft code generation pseudocode.
- Create UI wireframes and set up the project repo with dependencies.

Deliverables:
Wireframes, distribution list, project plan, and a working basic React app.

Week 1–2 (June 2 – June 15): Core Editor Setup

- Integrate React Flow and set up basic canvas layout.
- Enable node creation (via button/double-click) and dragging.
- Implement basic edge drawing and maintain internal state.
- Start simple node editing (name and type toggle).
- Begin scaffolding the sidebar and distribution constants list.

Deliverables:
A basic web app with addable, draggable nodes and connectable edges, with rudimentary editing functionality.

Week 3–4 (June 16 – June 29): Node Editing and Distribution Support

- Build detailed side panel for node properties.
- Support distribution selection (with fields shown conditionally).
- Add input validation (e.g., required fields, type constraints).
- Implement node/edge deletion and context menu.
- Begin plate UI structure: basic visual container with a label.

Deliverables:
Editor supports assigning distributions and formulas. Node/edge deletion works. Initial plate visuals in place.

Week 5–6 (June 30 – July 13): Plate Notation and JSON Export

- Finalize plate implementation (with support for nested plates if feasible).
- Show plate membership visually and maintain grouping state.
- Add validation: prevent graph cycles, enforce plate scoping rules.
- Implement JSON export of model structure (nodes, edges, plates).
- Begin code generation module (handle simple, plate-free models first).

Deliverables:
Plate support and validation complete. JSON export working. Initial code generation functional for simple models.

Midterm Evaluation (July 14 – July 18)

- Write report with progress summary, screenshots, and example outputs.
- Demonstrate building a sample model in the UI and exporting code.
- Gather mentor feedback and refine roadmap.

Midterm Goal:
A working prototype that lets users build and export simple models.

Week 7–8 (July 19 – August 2): Full Code Generation and Refinement

- Implement full code generation for models with plates and deterministic nodes.
- Finalize loop generation, handle parent-child scoping across plates.
- Validate generated code using JuliaBUGS locally.
- Improve JSON import/export (load saved models).
- Apply midterm feedback (UI polish, bug fixes, minor features).

Deliverables:
Complete code generator. App can build full models (e.g., hierarchical) and export valid JuliaBUGS code.

Week 9–10 (August 3 – August 16): Testing and UI Polish

- Write unit tests (e.g., test code generation logic).
- Write UI tests using React Testing Library and Cypress.
- Conduct cross-browser testing and fix compatibility issues.
- Refine UI/UX based on feedback (tooltips, messages, visuals).
- Start documentation and tutorial writing.

Deliverables:
Beta version with extensive testing and a polished UI. Draft documentation/tutorial ready.

Week 11–12 (August 17 – August 30): Finalization and Delivery

- Complete documentation site (tutorial, examples, component reference).
- Final UI and performance refinements.
- Create sample models and demos (JSON + code + screenshots).
- Final round of user testing and feedback integration.
- Prepare codebase for handoff (README, license, install guide, Dockerfile).

Deliverables:
Final release ready. Fully documented, tested, and demoed. App is stable and usable by others.

Final Submission (August 31 – September 1)

- Submit final GSoC materials: code, summary, documentation, and demos.
- Ensure everything is merged and published.
- Reflect on progress and outline possible next steps.

Post-GSoC Plan

Remain engaged to help integrate with JuliaBUGS backend, guide users, and contribute to future improvements.

Testing and Documentation Plan

Ensuring the correctness and stability of the JuliaBUGS graphical editor is essential. Our testing strategy will be based entirely on Jest, covering all key functionality from logic to UI behavior. This simplifies the toolchain while maintaining thorough test coverage.

Testing Strategy

1. Unit Tests (Jest)
   We will use Jest to test all core logic and utility functions in isolation:
   - Code generation tests: Given an internal model (e.g., one node with Normal(0,1)), Jest will assert the generated BUGS code matches expected output.
   - Edge cases: Handle unusual model structures such as empty graphs, disconnected nodes, and nested plates.
   - Utility functions: Functions like graph validation (e.g., acyclicity checks) and topological sorting will have their own dedicated tests.
2. Component-Level Tests (Jest + React Testing Library)
   Jest will also be used in combination with React Testing Library to test UI components:
   - Form behavior: Verify that selecting a distribution shows the correct parameter fields.
   - State updates: Simulate user input and check that state reflects the correct node or plate properties.
   - Graph rendering: Confirm nodes and edges appear as expected based on mock state.
3. Simulated End-to-End Behavior (via Jest)
   While not full browser automation, we will simulate E2E-like flows by:
   - Rendering the full App component and simulating user actions (e.g., adding a node, assigning a distribution, exporting code).
   - Verifying the final output JSON or BUGS code matches expected structures.
   - These tests will ensure the system works cohesively without requiring a separate E2E framework.
4. Manual Testing
   In addition to automated tests, we'll manually verify the UI in real browsers:
   - Cross-browser checks (Chrome, Firefox, Safari).
   - Usability and UX feedback from mentors.
   - Edge cases like deleting nodes inside plates or reconnecting edges will be tested interactively.

5. CI Integration (GitHub Actions)
   We'll use GitHub Actions to:
   - Run all Jest tests on each push or pull request.
   - Perform type-checking and linting.
   - Ensure regressions are caught early and maintain code reliability throughout development.

Documentation Plan

1. Developer Documentation
   - Clear comments in code, especially for the model schema and code generation logic.
   - A concise guide explaining state structure, how to add new distributions, and how the app is structured.
   - Maintained in a /docs folder or GitHub Wiki.
2. User Guide
   - Covers usage from creating nodes to exporting code.
   - Includes local setup instructions and possibly deployment options (static build, hosted link).
   - Rich in visuals: screenshots and step-by-step instructions (Markdown or static site generator).
3. Tutorial Example
   - A complete walkthrough of building a model (e.g., coin toss or Eight Schools).
   - Each step accompanied by visuals and final generated code.
   - Helps teach both the tool and Bayesian modeling concepts.
4. JSON & Code Schema Reference
   - Formal description of the model JSON format.
   - Guide to interpreting and possibly editing model files manually.
   - Notes on how the code generation maps graphical models to BUGS syntax.
5. Examples and Templates
   - A library of example models (examples/ directory), including their JSON files and the generated code.
   - Useful for demos, learning, and starting points for new users.

Future Work and Expansion

Due to the time limitations of GSoC, some advanced goals are out of scope for this summer but remain highly relevant for the long-term success of the JuliaBUGS graphical interface. The foundation built during GSoC is designed to support these future enhancements, ensuring continued growth and utility of the project.

Backend Integration (Inference Engine)

A major next step is connecting the frontend to a Julia backend to allow running inference directly from the UI. The goal is for users to provide data and trigger MCMC sampling, receiving results (posterior samples, diagnostics) in return. One approach is using Genie.jl to host a web API that takes model JSON/code and data, runs JuliaBUGS, and returns results.

The code generation and model JSON designed during GSoC serve as the integration point. A future /run_model endpoint could make the tool interactive and complete the modeling pipeline. This may later be deployed via Genie apps or Jupyter environments, expanding usability for Julia users.

Interactive Diagnostics and Visualization

Once inference is supported, the next logical step is adding an interactive results and diagnostics dashboard. Features could include:

- Trace plots, posterior densities, $\hat{R}$, ESS metrics
- Visual overlays on the model graph showing uncertainty or sample quality
- Dynamic result panels with plotting libraries like Plotly, D3, etc.

This would make the tool not just a model builder but a complete Bayesian workflow environment, integrating modeling, inference, and validation in one interface.

Collaboration and Sharing

Long-term usability benefits from collaborative and import/export capabilities:

- Cloud integration for saving/loading models
- Exporting self-contained model + data files
- Support for importing existing JuliaBUGS/BUGS code and visualizing it (i.e., reverse-parsing code to a graph). Although complex, this may be enabled by JuliaBUGS's internal parser and would greatly help those with legacy models.

Advanced Modeling Features

Once the base system is stable, more sophisticated features can be explored:

- Hierarchical plates: Multi-level nesting for models like students → schools → districts
- Mixture models: Support for nodes with multiple parent distributions and mixture weights
- Temporal models: Time-indexed structures like Dynamic Bayesian Networks
- Function blocks: Visual elements for common transformations (e.g., log, exp)
- Hyperpriors: Templates or UI guidance for hierarchical priors on parameters

Performance and Scalability

As usage grows, performance may need to scale:

- Lazy rendering or WebGL for large models
- Memory optimizations for model state
- Possible Electron-based desktop version, bundling Julia for local offline use (a heavier but useful extension)

Educational Use and Community Growth

The editor is also a powerful educational tool:

- Creation of interactive exercises for Bayesian modeling courses
- Presentations at JuliaCon or community calls to attract interest. Over time, a contributor base and user community can form. Future GSoC students or contributors could take on these enhancements, ensuring long-term momentum.

Summary

This GSoC project lays the groundwork for a full-featured, browser-based Bayesian modeling platform. Post-GSoC, the top priorities are backend integration and result visualization. Beyond that, collaboration tools, advanced model support, and educational outreach will help the tool grow into a core part of the Julia probabilistic programming ecosystem. With a solid architectural foundation, this project is positioned to evolve well beyond the summer and become a widely adopted, community-driven tool.

Why Me?

I have a strong foundation in both probabilistic programming and web development, making me well-prepared for this project. As a contributor to Turing.jl, I've worked closely with Julia's Bayesian tools and understand the needs of the community. I'm also experienced in building polished, interactive web apps using React and TypeScript. This rare combination of skills allows me to bridge statistical modeling with frontend development effectively. See my projects for reference: shravangoswami.com

Acknowledgement

I am extremely grateful to my prospective mentors, Xianda Sun, Hong Ge and Markus Hauru for their guidance, feedback, and support throughout the application process. I also express my sincere appreciation to the Julia community, especially the TuringLang ecosystem, for providing invaluable resources, discussions, and motivation, which have significantly contributed to shaping this proposal.

Conclusion

This project offers a unique opportunity to create an accessible, modern interface for JuliaBUGS, helping users visually build Bayesian models. With the planned design and guidance from mentors, I'm confident we can deliver a valuable tool for learners, researchers, and practitioners alike. I'm excited to contribute meaningfully to the Julia ecosystem through GSoC 2025.

References

1. **WinBUGS Bayesian Modeling Framework (Lunn et al., 2000)**
   Introduced WinBUGS software and its DoodleBUGS graphical model interface, making Bayesian modeling visually accessible.

2. **MultiBUGS Parallel Implementation (MultiBUGS Development Team, 2020)**
   Extends WinBUGS/OpenBUGS with parallel MCMC computation to speed up inference for large Bayesian models.

3. **MultiBUGS: A Parallel Implementation of the BUGS (2018 Paper)**
   MultiBUGS is a new version of the general-purpose Bayesian modelling software BUGS that implements a generic algorithm for parallelising Markov chain Monte Carlo (MCMC) algorithms to speed up posterior inference of Bayesian models.

4. **ShinyStan Interactive Diagnostics (Gabry, 2018)**
   Web GUI providing interactive exploration of MCMC diagnostics for Bayesian models, highlighting benefits of interactive interfaces.

5. **causact R Package (Fleischhacker & Nguyen, 2022)**
   Allows creation of probabilistic graphical models through generative DAGs and automatic Greta code generation, inspiring visual-to-code design approaches.

6. **JuliaBUGS.jl (Xianda Sun & JuliaBUGS Contributors, 2025)**
   Julia-based modern implementation of the BUGS language, combining intuitive model specification with Julia's computational capabilities.

7. **React Flow Documentation (2025)**
   Library used for building node-based interactive UIs with React, featuring draggable nodes, zooming, and easy diagram management.

8. **Plate Notation (Wikipedia)**
   Explains plate notation, a concise graphical method for representing repeated variables or sub-models within Bayesian graphical models.

9. **Differences from Other BUGS Implementations (JuliaBUGS Docs, 2025)**
   Lists and contrasts various BUGS implementations (WinBUGS, OpenBUGS, JAGS, Nimble, MultiBUGS), emphasizing JuliaBUGS' modernization efforts.

10. **Interactive Bayesian Modeling Representations (Taka et al., 2020)**
    Explores graphical tools enhancing Bayesian model interpretability, emphasizing the value of interactive graphical interfaces.

11. **DoodleBUGS Interface (OpenBUGS User Manual, 2007)**
    Describes graphical model specification using nodes and arrows in WinBUGS/OpenBUGS, highlighting GUI strengths and limitations.

11. **Bayesian Graphical Models Lecture Notes (Breheny)**
    Concise lecture notes introducing Bayesian graphical models, detailing basic concepts, DAG structures, and illustrating model representation clearly.

12. **The BUGS Project Overview (Lunn et al., 2009)**
    Provides a detailed retrospective on the BUGS project evolution, architecture, critiques, and directions for extending Bayesian modeling tools.

Thank You So Much!