Statewise: A Petri Net-Based Visual Editor for Specifying Robotic Systems

Zejun Zhou[1], **Yuchen Jin**[2], **Pragathi Praveena**[3]

[1]Department of Computer Science, Brown University
[2]Department of Computer Sciences, University of Wisconsin–Madison
[3]The Robotics Institute, Carnegie Mellon University
zejun_zhou@brown.edu, jin252@wisc.edu, pragathi@cmu.edu

Abstract

We present *Statewise*, a visual editor designed to enable developers to model and simulate complex systems using colored Petri nets in an intuitive, graphical way. Utilizing *Statewise*, we explore two use cases to demonstrate its capabilities. We also discuss potential enhancements to further extend its applicability in more complex scenarios.

Introduction

In our previous work (Praveena et al. 2023), we advocated for the use of Petri nets as a modeling language for the iterative development process of interactive robotic systems. We proposed that Petri nets, particularly when adapted into a ***human-centered, domain-specific variant*** tailored for system development, could serve as a unifying representation across various stages of the development process. The development of complex systems relies on the expertise of a diverse team of *application developers*, including domain experts, designers, programmers, and researchers. The representations and tools that application developers utilize for their respective stages of system development — such as low-fidelity prototypes by designers and high-fidelity simulators by algorithm developers — may not be effectively interoperable across different stages. While representations and tools tailored for individual phases are indispensable, an accompanying, unified representation can enhance this design process by encoding information that can be used to coordinate between phases. Petri nets can provide a precise, concise, and human-friendly way to represent system behavior, yielding models that are both mathematically sound and accessible to systems developers through their visual and graphical nature (Jensen and Kristensen 2009).

In Praveena et al. (2023), we also identified the need for development tools that selectively expose aspects of the underlying representation to developers in an intuitive, graphical way. In pursuit of this goal, we are developing *Statewise*, a visual editor designed to simplify the creation, editing, and simulation of Petri net models. Our hope is that, in the future, using this tool for development will allow us to demonstrate the potential of Petri nets as a modeling language for the iterative development of interactive systems. This paper presents our ongoing efforts in developing the tool.

Related Work

This overview on Petri nets is excerpted from our prior work (Praveena et al. 2023). Petri nets are a class of state machines that specialize in modeling the flow and dependencies between events and resources in distributed and concurrent systems (Peterson 1977). Petri nets are defined as a set of *place* nodes that hold *tokens*, *transition* nodes that indicate actions that move tokens from place to place, and directed *arcs* between places and transitions that define the logic of where and which tokens are consumed and produced by transitions. *Firing* a transition, *i.e.,* consuming tokens to execute a transition, depends on the availability of sufficient tokens on incoming arcs from places to the transition. *High-level* Petri nets have additional features that allow for a more compact representation of dependencies while producing a more semantically meaningful model (Jensen 1983). A *Colored* Petri net is a High-level Petri net that includes data values of different types (*colors*) in individual tokens, such as integers, strings, or user-defined types, and *guard* functions that evaluate whether the tokens present in incoming arcs are sufficient for the transition to fire.

Several tools have been developed for Petri net modeling in the research community. Here, we present two notable tools. *CPN Tools* (Jensen, Kristensen, and Wells 2007) is a popular tool for generating colored Petri nets. Despite its popularity, it presents several challenges for new users. For example, the "add nodes" function is difficult to locate and the floating toolboxes used to create nodes obscure the view of the Petri net canvas. Additionally, the tool offers a limited color palette, has restricted customization options, and lacks visualization for tokens in each place. *PIPE* (Bonet et al. 2007) offers similar functionality to *CPN Tools*, but its node creation process is more complex, requiring multiple parameters to be set before creating a single place or transition node. However, *PIPE* does provide visualization of tokens in each place and supports run-time changes to the Petri net model, which is beneficial for dynamic visualization.

Historically, the development of Petri net tools has been driven by academic communities focused on formal systems modeling, with little overlap with the HCI community. As a result, these tools often have a steep learning curve and require significant expertise to use effectively. The lack of tools designed with a human-centered approach has also limited the ability to gather evidence demonstrating the util-

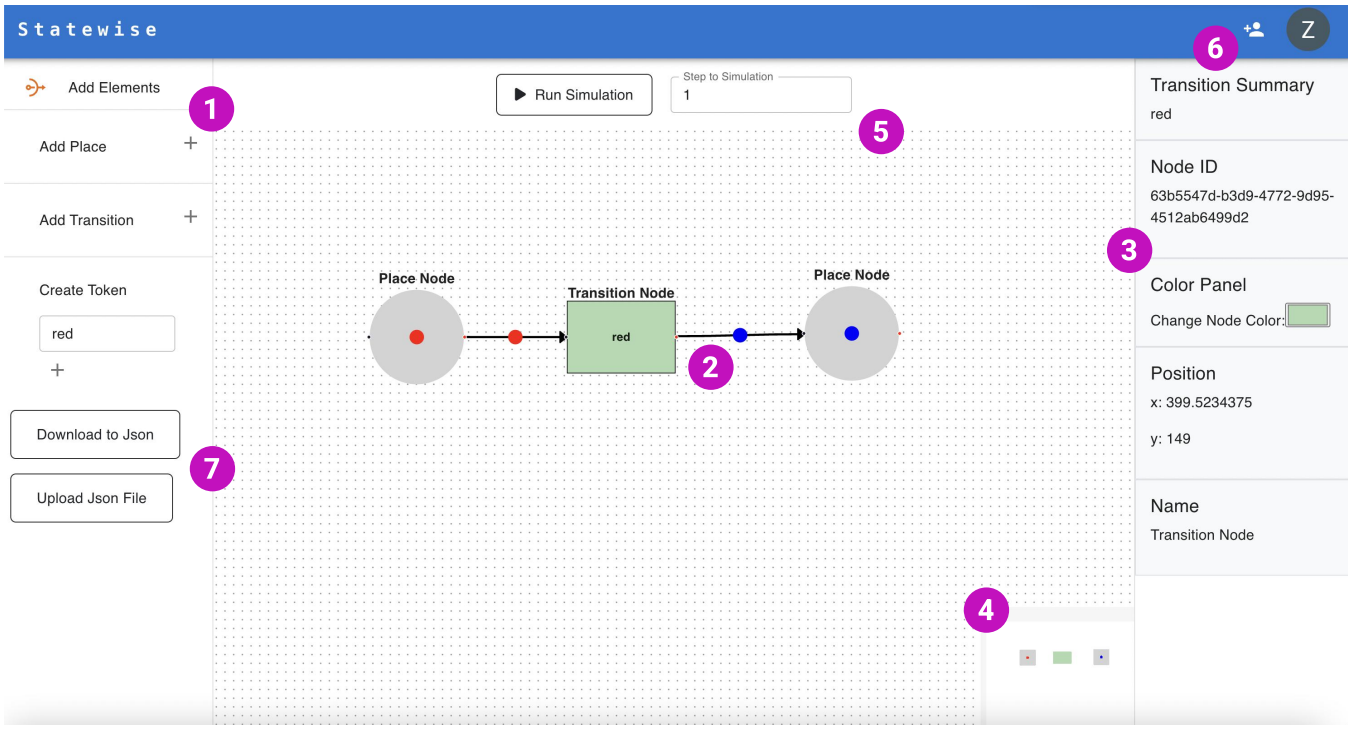

Figure 1: Key features of the *Statewise* interface are numbered. ① Drag-and-drop functionality allows users to add places, transitions, and tokens to the canvas. ② Arcs can be added by dragging from the red source dot of one node to the black target dot of another. ③ When an element is selected, its details are displayed in the right panel. ④ A minimap provides an overview and aids in navigation of the entire Petri net. ⑤ Stepped simulation enables runtime analysis of the Petri net. ⑥ Users can invite collaborators to join the current project. ⑦ Projects can be saved offline as JSON files and uploaded from local drives.

ity of Petri nets as a representation in application development. Our work aims to address this gap through human-centered design and a focus on usability and accessibility.

Statewise Editor

Statewise is a visual editor for colored Petri nets, designed to simplify the creation, editing, and simulation of Petri net models. Key features of *Statewise* are illustrated in Figure 1.

Visual Editor

Drag-and-Drop Functionality *Statewise* offers drag-and-drop functionality (Figure 1: ①) to simplify the process of constructing Petri net models. Users can drag place nodes, transition nodes, and tokens from the left panel onto the canvas. When creating tokens, users can select a specific color for the token. Dragging place nodes, transition nodes, and tokens outside the canvas will cancel the action to prevent unintentional additions to the Petri net model. Additionally, tokens can only be added to place nodes. If a user drops a token onto a transition node, no action will be taken since transitions cannot hold tokens.

Arcs are used to connect place nodes to transition nodes, and vice versa, but cannot connect two nodes of the same type. If the user attempts to do so, an error message will appear, and no arc will be created. To add an arc, users can drag from the red source dot on one node to the black target dot on another node (Figure 1: ②). The direction of the arc will always be towards the target dot, visually representing the flow of tokens between place nodes and transition nodes.

Element Manipulation *Statewise* provides users with the ability to manipulate Petri net elements on the canvas. When an element (place, transition, token, or arc) is selected, its information is displayed in the right panel (Figure 1: ③).

Place nodes and transition nodes can be moved to any location on the canvas, but tokens cannot. Tokens can only be moved within the place node where they were created. This restriction ensures that the relationship between the token and its place node remains intact. When a node is moved, any connected arcs will be automatically repositioned with respect to the new location of the node. Users can also delete any element on the canvas. Deleting a place or transition connected to arcs will automatically remove the associated arcs to ensure consistency in the model.

Users can relabel place nodes, transition nodes, tokens, and arcs by selecting the element, entering the desired name in the input field above it, and clicking submit. Arc labels indicate which tokens are consumed from source place nodes or produced at destination place nodes when a transition fires. For example, in Figure 1, the arcs are labeled with a red and a blue token. This indicates that when the transition fires, a red token is consumed from the source place, and a blue token is produced at the destination place. Further details on arc labels are discussed in §Simulation.

Minimap Located in the bottom right corner of the canvas, the MiniMap provides an overview of the entire Petri net model and is updated in real-time to reflect any changes to the model (Figure 1: ④). It allows users to quickly locate and focus on specific sections of the model, making it useful for navigating large models efficiently.

Simulation

Guard Expressions Guard expressions use Boolean logic to define the conditions that must be met for a transitions to fire. In *Statewise*, labels from multiple arcs connected to a transition are automatically combined into a guard expression on the transition. When this guard condition is met during simulation, the transition will fire. When nodes and arcs are deleted or updated, the guard expression is updated to reflect these changes.

Users can label arcs with a simple condition, such as *color:red*, or a more complex condition using Boolean operators, such as *color:red || color:blue* (see Figure 2). The labels from multiple arcs connected to a transition are combined into a guard expression for that transition. For instance, if a transition has two incoming arcs labeled with *color:red || color:blue* and another with *color:black*, the resulting guard expression would be *(red || blue) && black*. This implies that the transition can only fire if the place connected to the first incoming arc contains either a red or blue token, and the place connected to the second incoming arc contains a black token.

Defining guard expressions directly can be a challenging and error-prone task, especially when dealing with complex conditions involving multiple tokens and Boolean operators. Instead, users specify the consumption and production of tokens for each arc, and *Statewise* automatically combines these into a guard expression.

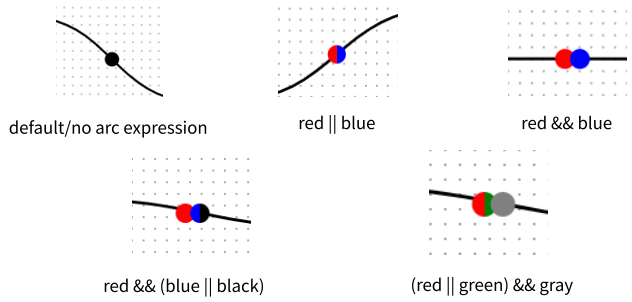

Figure 2: Examples of arc labels

Stepped Simulation Users can progress through a simulation and observe state changes at each step (Figure 1: ⑤). This iterative process allows users to understand the dynamics of their Petri net models, identify potential issues, and refine their designs. At each simulation step, all guard expressions in the model are evaluated. If a guard condition is satisfied, the corresponding transition fires and is highlighted in green; otherwise, it remains gray. When a transition fires, it consumes tokens from the source place nodes and generates tokens at the destination place nodes, following the rules defined by the arc labels.

For more in-depth analysis, users can run multi-step simulations to monitor the system's behavior over time. Users can set the number of simulation steps and pause at any point to review the model's current state.

Other Features

User Authentication When users first visit *Statewise*, they are prompted to log in using their Google account.

Project Management Once logged in, users are presented with a dashboard that provides an overview of all their projects, each displayed as a card showing the project name, creation date, and owner's email. Users can create new projects by clicking the "Create New Project" button and providing a project name, which gives them immediate access to a blank canvas for Petri net modeling.

Collaborative Editing *Statewise* supports collaborative modeling, allowing users to develop and refine Petri net models together. Through the navigation bar, users can invite collaborators by clicking the "Invite" button and entering their email. Once the collaborator logs in, the shared project appears on their dashboard along with the creator's name. When multiple users collaborate on the same project, they can see each other's cursors on the canvas. Each collaborator's cursor is uniquely colored for easy identification. Any actions, such as moving elements, adding tokens, or modifying labels, are instantly updated on all users' canvases.

Save and Load Users can save the current state of their Petri net canvas as a JSON file, which captures all elements, labels, and color configurations. This allows them to preserve their work offline and share it with collaborators. Additionally, users can load previously saved JSON files to restore their Petri net models.

Implementation

Statewise is built using a combination of modern web technologies. The frontend is built with React.js, using the React Flow library to create a node-based editor on the canvas. React Flow provides an effective mechanism to capture the position of elements on the canvas, which facilitates the implementation of the drag-and-drop feature. Through this library, we encapsulate custom logic for selecting, deleting, and modifying places, transitions, tokens, and arcs. For example, users can select an element, view its details in the right panel, and then delete it by pressing the delete key (on Mac) or backspace key (on Windows). The frontend also handles inviting collaborators, saving and loading projects, managing the dashboard, executing simulation logic, and customizing colors using React.js. User authentication is securely managed using the Google OAuth 2.0 API.

The backend is implemented using Python Flask, which provides a RESTful API for interaction between the frontend and the database. The backend handles all Create, Read, Update, and Delete (CRUD) operations for places, transition, tokens, and arcs. Additionally, the backend manages

project and user data, and implements the logic for inviting users to existing projects.

For real-time collaborative editing, *Statewise* uses Yjs technology combined with the WebRTC protocol. Yjs is a powerful implementation of CRDT (Conflict-free Replicated Data Type) that enables real-time shared editing by ensuring consistency across all collaborators' views. WebRTC provides peer-to-peer communication capabilities and efficient direct data transfer between users. This implementation allows efficient real-time data sharing to ensure that changes made by one user are immediately reflected on the canvases of all collaborators.

Cassandra was chosen as the database for *Statewise* due to its fast write capabilities and strong data consistency, which are essential for handling frequent model updates. Each piece of information, such as place, transition, token, or arc data, is stored as a row to maintain data integrity during save operations. Cassandra's partitioning feature allows efficient data retrieval and writing based on project and node IDs. Cassandra's architecture supports scalability, making it well-suited for both collaborative editing and managing large, complex Petri net models.

Use Cases

In this section, we present two examples demonstrating the use of *Statewise* to create Petri net models for application development scenarios.

Barber Shop Service Process

This simplified model models an interaction between customers and a robot barber in a barber shop.

Model Setup

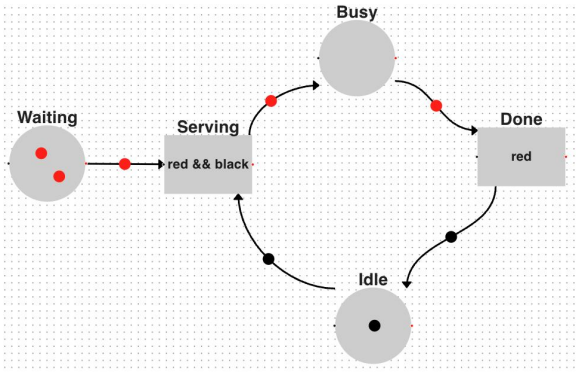

Figure 3: Initial state of barber shop service model

Place Nodes

Waiting: Represents the area where customers wait for service. Initially, it contains two red tokens, each representing a customer waiting to be served.

Busy: Indicates that a customer is currently being served. Tokens in this place show customers in the process of receiving a haircut.

Idle: Indicates that the robot barber is available for service. Initially, it contains one black token, indicating the robot barber is ready to serve a customer.

Transition Nodes

Serving: This transition represents the action of the robot barber starting to serve a customer. It is fired when there is at least one red token in *Waiting* and one black token in *Idle*.

Done: This transition signifies the completion of a customer's service. It is fired when there is a red token in *Busy*, indicating the customer has finished their haircut.

Tokens

Red Tokens: Represent customers. The movement of red tokens through the nodes illustrates the flow of customers in the barber shop.

Black Tokens: Represent the robot barber. The movement of black tokens indicates the robot barber's availability and engagement in serving customers.

Simulation

Refer to Figure 4.

Step 1: First Customer Serviced The simulation begins with the *Serving* transition firing because its guard condition, *red && black*, is met. One red token is consumed from *Waiting*, and one black token is consumed from *Idle*. A red token is produced in *Busy*, indicating the customer who is now being served by the robot barber.

Step 2: First Customer Completes Service The *Done* transition fires as a red token is present in *Busy* from the previous step, signaling the service is complete The red token is consumed from *Busy*, and a black token is produced back in *Idle*, indicating the robot barber is now available.

Step 3: Second Customer Serviced With a black token back in *Idle*, the *Serving* transition fires again as the second red token in *Waiting* meets the condition. The second red token moves from *Waiting* to *Busy*, and the black token is consumed from *Idle*, repeating the service process for the second customer.

Step 4: Second Customer Completes Service The *Done* transition fires again with the red token in *Busy*. The final red token is consumed, and the black token returns to *Idle*. The process concludes with no more customers waiting, leaving the robot barber idle.

Golang's GMP Model Process

The Go language's GMP (Goroutine, Machine, Processor) model is a core component of its concurrency management system. It efficiently schedules goroutines — the lightweight threads used in Go — across available system resources. In this model, Goroutines (G) are the tasks to be executed, Machines (M) represent OS threads that run those tasks, and Processors (P) are the queues of goroutines waiting to be assigned to M. The number of Processors is determined by the GOMAXPROCS runtime configuration. This model allows Go to handle massive concurrency efficiently, utilizing available hardware resources for execution while abstracting the complexity of thread management from the developer.

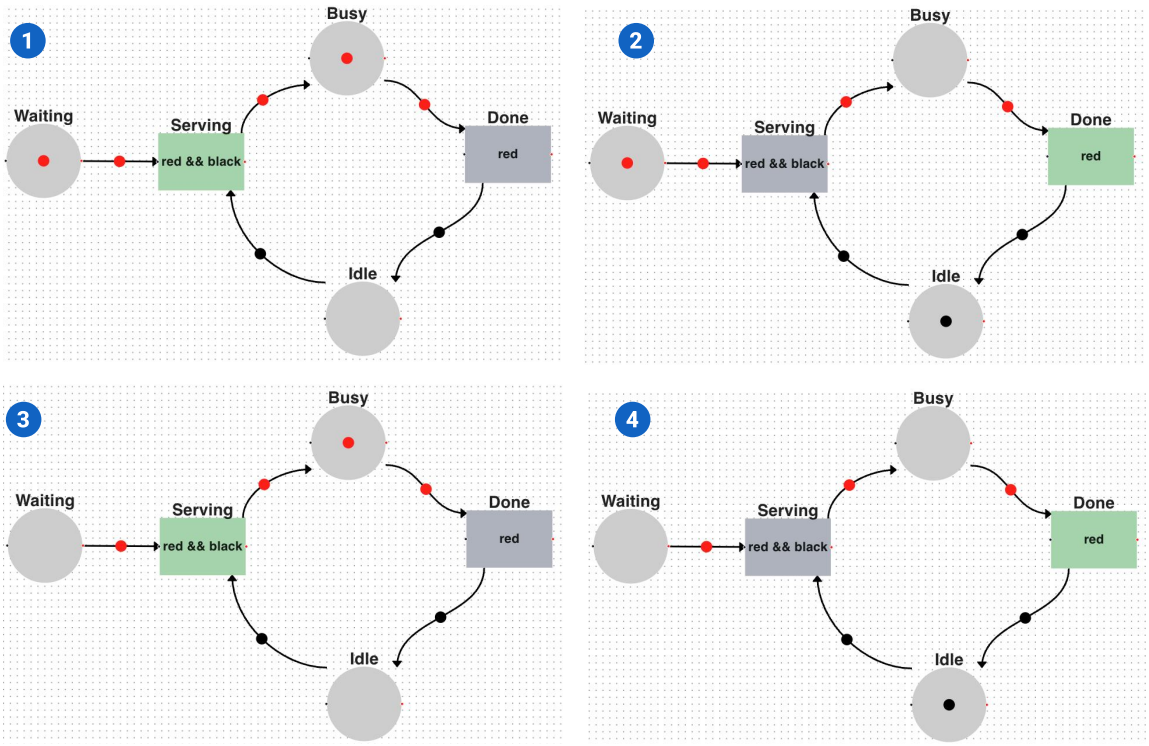

Figure 4: Simulation of barber shop service process

Model Setup

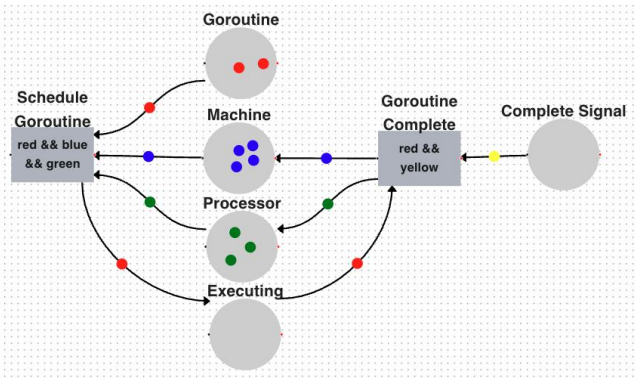

Figure 5: Initial state of GMP model

Place Nodes

Goroutine: Represents the pool of goroutines waiting to be scheduled. Initially, it contains two red tokens, each symbolizing a goroutine ready to be executed.

Machine: Represents the available operating system threads. Initially, it contains four blue tokens, each corresponding to an available OS thread.

Processor: Represents the queue of processors, represented by green tokens, ready to manage the scheduling of goroutines. The number of processors is determined by the GOMAXPROCS setting in the Go runtime, which is set to three in this example.

Transition Nodes

Schedule Goroutine: Represents the scheduling of a goroutine. It fires when a red token is present in *Goroutine*, a blue token in *Machine*, and a green token in *Processor*.
Goroutine Complete: Represents the end of a goroutine's life cycle. It fires when a red token is in *Executing*, and a yellow token is added to *Complete Signal*.

Tokens

Red Tokens: Represent goroutines, illustrating the scheduling and execution process within the GMP model.
Blue Tokens: Represent machines (OS threads), indicating available execution resources.
Green Tokens: Represent processors, showing the system's scheduling capacity.
Yellow Tokens: Act as completion signals within the model to manage the goroutine lifecycle.

Simulation

Refer to Figure 6.

Step 1: First Goroutine Scheduled The simulation begins with the *Schedule Goroutine* transition firing because its guard condition is met. One red token is consumed from *Goroutine*, one blue token is consumed from *Machine*, and one green token is consumed from *Processor*. A red token is produced in *Executing*, representing the goroutine that has been scheduled and is now being executed by an available machine and processor.

Step 2: Second Goroutine Scheduled The *Schedule Goroutine* transition fires again. This is possible because the required resources are still available: one red token remains

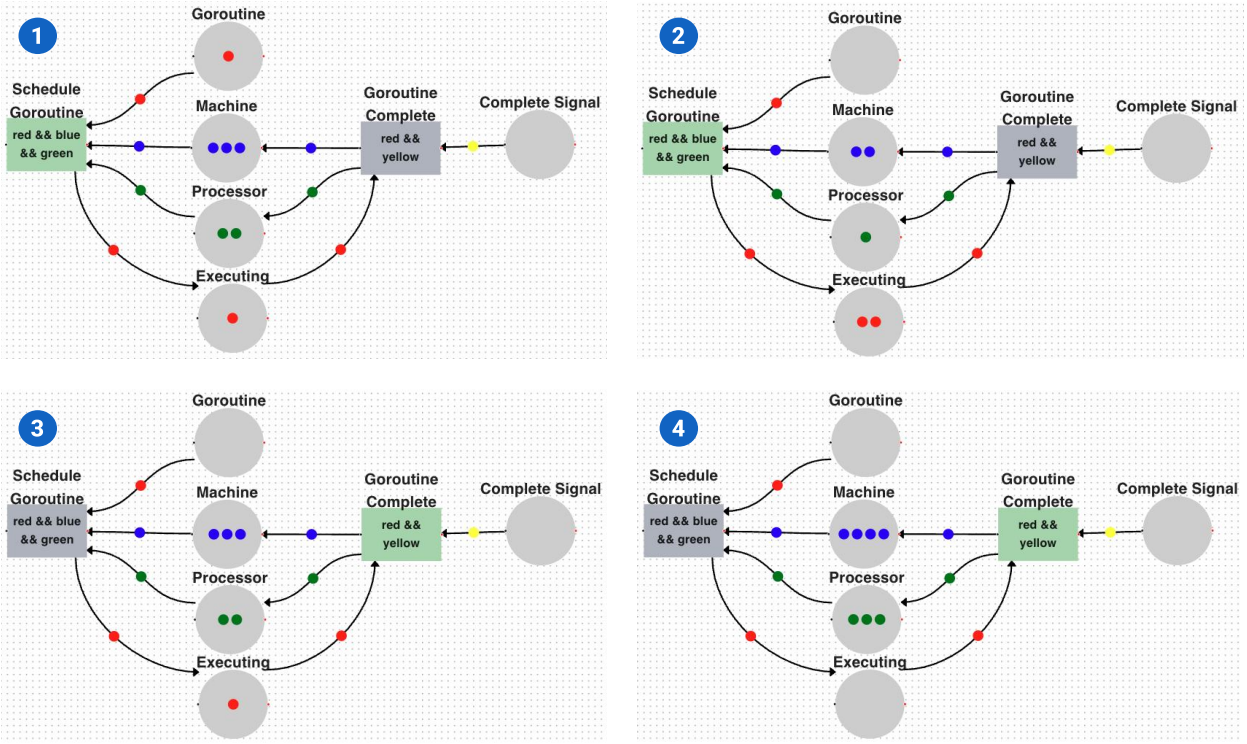

Figure 6: Simulation of Golang's GMP model process

in *Goroutine*, indicating another goroutine is ready to be scheduled; three blue tokens remain in *Machine*, indicating another OS thread is available for scheduling; and two green tokens remain in *Processor*, indicating at least one processor is ready to manage the scheduling. After the same consumption rule as the first step, a second red token is produced in *Executing*. This reflects that two goroutines are now in execution, utilizing available machines and processors.

Step 3: First Goroutine Finished The *Goroutine Complete* transition fires because the required conditions are met: two red tokens are present in *Executing*, indicating two goroutines are currently being executed, and a yellow token has been manually added to *Complete Signal* to signify that the execution of the first goroutine has completed. As a result, one red token is consumed from *Executing*, and the yellow token is consumed from *Complete Signal*. The completion of the goroutine frees up the resources it was using, so one blue token is produced in *Machine*, and one green token is produced in *Processor*, representing the availability of the OS thread and processor for future scheduling.

Step 4: Second Goroutine Finished *Goroutine Complete* fires again because the necessary conditions are met. Following this, the red token from *Executing* and the yellow token from *Complete Signal* are both consumed. Similar to the previous step, this frees up the resources that were used by the goroutine. Consequently, one blue token is produced in *Machine*, and one green token is produced in *Processor*, making them available for future tasks. Now, all goroutines have been scheduled and executed, and the system has returned to a state where all resources are available and no more goroutines are waiting to be scheduled or executed.

Discussion

Statewise offers an initial platform for modeling and simulating colored Petri nets, with opportunities for further extensions to enhance its capabilities. One key area for development is the incorporation of a timing mechanism, where tokens must remain in a place node for a specified duration before a transition can fire, better reflecting real-world scenarios. For example, in the Barber Shop model, a customer would need to stay in *Busy* for the estimated time required for a haircut before the *Done* transition can fire.

Another enhancement involves enabling *Statewise* to take input from external sources, such as APIs, to automate token generation. This would allow tokens, like the yellow token in the *Complete Signal* place, to be generated based on real-world signals. Additionally, having the ability to record the execution process of the Petri net allows for tracking metrics such as token durations in place nodes and the sequence of fired transitions. This would provide insights into model behavior and aid in the refinement of system behavior.

Our next steps will involve conducting user studies with application developers to gather insights into the usability and accessibility of *Statewise* as a tool for specifying interactive system behavior. Our hope is that, in the future, using this tool for development will allow us to demonstrate the potential of Petri nets as a modeling language for the iterative development of interactive systems.

Acknowledgements

This work was supported by National Science Foundation award 1925043. We would like to thank Andrew Schoen, Haoming Meng, and Yuna Hwang for their contributions to the development of *Statewise*.

References

Bonet, P.; Lladó, C. M.; Puijaner, R.; Knottenbelt, W. J.; et al. 2007. PIPE v2. 5: A Petri net tool for performance modelling. In *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*.

Jensen, K. 1983. High-Level Petri Nets. In *Applications and Theory of Petri Nets: Selected Papers from the 3rd European Workshop on Applications and Theory of Petri Nets Varenna, Italy, September 27–30, 1982 (under auspices of AFCET, AICA, GI, and EATCS)*, 166–180. Springer.

Jensen, K.; and Kristensen, L. M. 2009. *Coloured Petri Nets*. Berlin, Heidelberg: Springer.

Jensen, K.; Kristensen, L. M.; and Wells, L. 2007. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9: 213–254.

Peterson, J. L. 1977. Petri Nets. *ACM Computing Surveys (CSUR)*, 9(3): 223–252.

Praveena, P.; Schoen, A.; Gleicher, M.; Porfirio, D.; and Mutlu, B. 2023. Petri Nets for the Iterative Development of Interactive Robotic Systems. In *Proceedings of the AAAI Symposium Series*, volume 2, 526–531.