Matt Stewart

Petri Dish

CSC 464 Final Project

Abstract

Petri Dish is a Python program which builds, simulates, and animates a Petri Net from an input text file. It can simulate a given net using sequential operations in random order, concurrent operations using threads, or sequential operations which are then rendered and animated via Python's matplotlib module[1]. This report comprises a justification for **Petri Dish**, an overview of prior work in similar directions, an explanation of its implementation, evaluations of its different modes of operations, and future work to be done.

Problem

Concurrent programming presents a collection of issues and complexities which sequential programming generally does not. Although the rewards in time and efficiency can be great, an incorrect implementation of a concurrent algorithm runs the risk of deadlock, starvation, and race conditions, all of which can impede or halt the execution of a program or introduce an unacceptable degree of uncertainty & non-determinism into the results.

Part of the difficulty of concurrent programming - and concurrent thinking - stems from the difficulty of fully realizing the implications of a piece of code when run in parallel. Often, too many threads will be executing too many tasks, and too many different outcomes and operational orders are possible, for a programmer to form a complete picture of the issues that might arise in a distributed system over time. Even when a system is working and producing output, it is often difficult to discern exactly what control sequences are occurring behind the scenes without parsing many thousands of print statements and making inferences about the conditions that produced their ordering.

Petri Nets

One mitigating factor to this problem lies in the development of visualizations and models for concurrent behaviours. In particular, this solution deals with Petri Nets: directed bipartite graphs for modeling the behaviour and relationships in a distributed system.

A Petri Net consists of two types of node: states, each holding a variable number of tokens representing both the current control location in the system and the resources needed to perform operations, and transitions, connecting one or more states to one or more different states by inputs and outputs. Each transition has M input states and N output states. When all input states contain at least one token, the transition is able to "fire" by consuming one token from each of its input states and producing one token to each of its output states. This system is capable of modeling the control flow through a system and the usage of key resources in a corresponding piece of code or concurrency problem.

Petri Dish

Petri Dish presents a method by which Petri Nets can be constructed from input text files containing lists of states & their initial token configurations and transitions & their directed edges to and from connected states. The system is capable of displaying and animating a given Petri Net in order to provide the user with a concrete visual representation of the control flow within a system. Deadlock and starvation, as well as race conditions and other unexpected behaviours, can be seen clearly and reproduced given enough time or the proper initial conditions. As a result, the pitfalls of concurrent programming can be better anticipated and circumvented, and long-standing concurrency problems can be better understood through simulation.

Previous Work

Petri Net editors and simulators are fairly common, though most require an installation or more complicated framework to run.

Adrian Jagusch of the Carl von Ossietzky University of Oldenburg created APO, a browser-based Petri Net editor based on the university's APT (Analysis of Petri Nets and Transitions Systems) software[2]. Unfortunately, the web app appears to no longer be hosted or running correctly.

Several students from the Department of Computer Science and Engineering at Shinshi University developed a tool named HiPS, a Hierarchical Petri net Simulator capable of hierarchical or timed-net simulation[3]. HiPS is a powerful tool with a wide range of functionality including generation of reachability graphs and deadlock detection.

Nicholas J. Dingle, William Knottenbelt, and Tamas Suto of Imperial College London developed PIPE2: a Java-based tool for the performance evaluation of Generalized Stochastic Petri Net models [4]. Its interface is similar to HiPS and includes tools for parallel and distributed performance analysis.

Implementation Details

Overview

Petri Dish is a Python program, and requires the matplotlib module to construct its animations. It consists of 5 modules: a main program (PetriDish.py), and 4 modules (SequentialState.py, SequentialTransition.py, ConcurrentState.py, ConcurrentTransition.py) containing sequential and concurrent implementations of State and Transition classes & class methods respectively. Some implementation details are shared between the sequential, concurrent, and animated operational modes, and the basic control flow remains the same across all modes. Details on each of the modes are covered in the next 3 subsections.

Petri Dish is run from the command line using "Python PetriDish.py filename mode", where "filename" is the .txt file containing the Petri Net to be simulated and "mode" is a character flag representing the desired mode ('s' for sequential, 'c' for concurrent, and 'a' for animated).

The Petri Net text file must be formatted in a specific manner to be parsed by **Petri Dish**. States and transitions are separated with a header including the number of each present in the file. The STATE header is followed by a list of parameters separated by newlines: State ID, input transitions, output transitions, initial tokens, and position to render in the animation. The TRANSITION header is similarly followed by transitions with parameters: Transition ID, input states, output states, and render position.

```
STATES 2

S1
T2
T1
1
5,4

S2
T1
T2
0
5,6

TRANSITIONS 2

T1
S1
S2
6,5

T2
S2
S1
4,5
```

Fig A. *Sample input text file for* ***Petri Dish****.*

Petri Dish reads in the provided file and constructs a tuple for each state and transition in the graph: either State(ID, inputs, outputs, tokens, position) or Transition(ID,inputs,outputs,position) as required. It then calls InitializeSequential [or InitializeConcurrent, depending on mode] to construct a list of sState[cState] objects and sTransition[cTransition] objects.

At this point, the entirety of the Petri Net is represented by State objects and Transition objects in two lists. Each object contains references to the input and output objects it is connected to. Depending on the mode of operation, the system then simulates the Petri Net and collects and/or displays the results.

During the operation of any mode, the sTransition[cTransition] objects repeatedly perform checks to assess their eligibility to fire. A given transition is eligible to fire iff each of its input states contains at least one token, and firing causes one token to be removed from each input state and one token to be added to each output state.

Random Sequential

The Random Sequential mode simulates the Petri Net via a series of discrete timesteps. To avoid the same results for every simulation loop of the same Petri Net, the order in which transitions check their eligibility is randomly shuffled every step.

Random Sequential mode relies on one sState and one sTransition object for every state and transition in the Petri Net. sState, an alias for the State class in the SequentialState module, keeps track of its own id, pointers to input sTransition objects, pointers to output sTransition objects, current tokens, and 2D position. It contains the following methods:

- **Add_input(self, input):** Appends the given Transition to its input list.
- **Add_output(self, output):** Appends the given Transition to its output list.
- **Add_tokens(self, num):** Adds the given number of tokens to its tokens.
- **Set_position(self, pos):** Updates its 2D position to pos, a tuple (x,y).
- **Ready(self):** Returns true if self contains any tokens, otherwise returns false.
- **Input(self):** Adds one token to self; currently, the system is not designed to simulate or animate executions involving more than one token per state per timestep.
- **Output(self):** Removes one token from self; throws an error and halts execution if it contains no tokens before the operation.

sTransition, an alias for the Transition class in the SequentialTransition module, contains a similar structure, but it does not need to track tokens, and contains the following methods:

- **Add_input(self, input):** Appends the given State to its input list.
- **Add_output(self, output):** Appends the given State to its output list.
- **Set_position(self, pos):** Identical to Transition.set_position(self, pos).
- **Eligible(self):** Iterates through each input State. If all input States contain a token, returns true, otherwise returns false.
- **Fire(self):** For each input State, calls State.output(); for each output State, calls State.input(). Removes one token from every input and produces one to every output.

After the system has constructed a list of sStates and sTransitions from the input file, it sets a “fires” counter and a “checks” counter to keep track of how many successful transitions were conducted and how many transitions were checked for eligibility overall. It then logs the current time and conducts a sequence of simulation iterations (1000 by default). In each iteration, the list of transitions is shuffled to produce a non-deterministic eligibility check order, and each transition in the list checks its eligibility by assessing the token status of all of its input states. If a given transition is eligible to fire, it will do so, and increment the fires counter. Regardless, every transition that checks eligibility increments the checks counter.

After the specified number of iterations, the system returns results: how much time elapsed, how many checks were performed, and how many transitions fired.

Random Animated Sequential

Random Animated Sequential mode follows the same execution method as above, and relies on the same sState and sTransition classes, but its specific implementation is altered to permit rendering and animation through matplotlib.

Prior to reading from the input file, a figure window is initialized and a 10x10 grid is created for it. The 2D positions of each state and transition are used to render shapes on this grid according to the placements specified in the input file.

Once the lists of sStates and sTransitions are produced as above, a state, token, and transition primitive is created for each state and transition. Since the system only currently supports one token per state, each state is given its own token primitive, and these tokens are later rendered only selectively. The rendering environment is then initialized and populated with states, transitions, labels for each object based on its ID, tokens, and arrows pointing from each state to each transition and from each transition to each state according to their specified inputs and outputs.

Rather than being in its own iteration loop, the simulation execution is instead wrapped inside matplotlib's animate(i) function, where each frame each transaction checks its eligibility to fire and then does so if it can, incrementing checks and fires as above. Once these checks have all been performed for the frame, each state is checked once more; if it contains a token, its token is set to visible in the simulation, otherwise a token is not rendered.

By default, the animation is set to run for 360 frames, each one at a 100 ms interval in order to make the execution steps visible and comprehensible. As above, the system logs the start time and the time the simulation is stopped by the user, at which points it returns the elapsed time, the number of checks, and the number of fires.

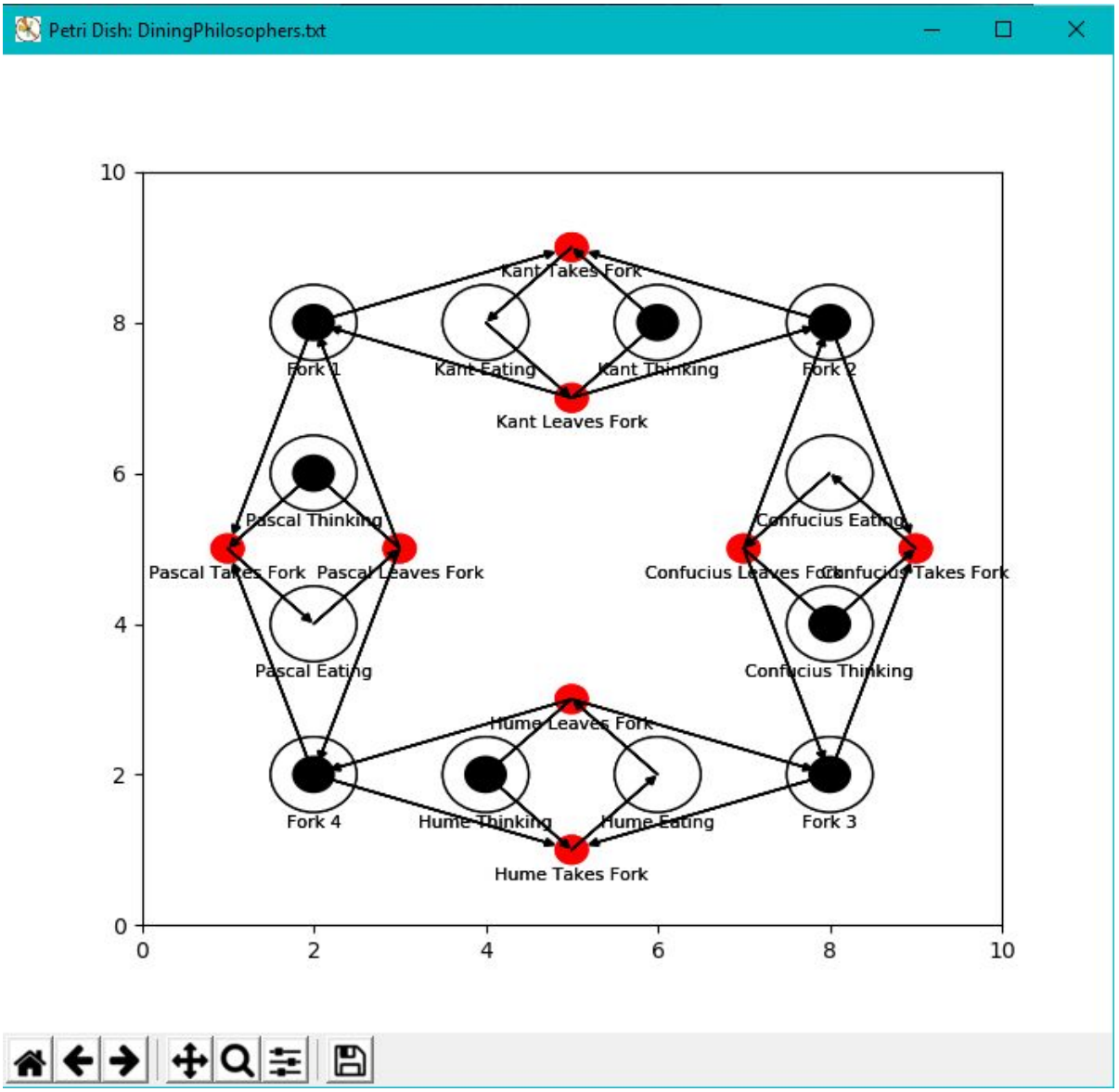

Fig 2. *The Dining Philosophers problem simulated in Random Animated Sequential mode.*

Concurrent

Concurrent mode uses the ConcurrentState and ConcurrentTransition modules, which describe modifications of the State and Transition classes to support atomic operations and threading. Specifically, each cState object is initialized with its own Lock object, in order for cTransition objects to acquire and release the lock as they check eligibility and prevent deadlock.

Each cTransition object is an extension of the Python Thread class, and contains a list of cState locks it has currently acquired, as well as the following methods in addition to those defined in the sState class:

- **Release_locks(self):** Releases each lock in its lock list, and removes each lock from its list as it does so.
- **Run(self):** Initializes a counter to 0 (similar to “iterations” in Random Sequential mode, but for each thread). For a number of counts (1000 by default), the transition checks for eligibility by first attempting to lock down access to every input and output state it needs to modify in the case of a successful fire. If any lock is already held by another thread, it releases all locks it has acquired so far and abandons the attempt to try again later, thereby ensuring the atomicity of the fire transaction and also avoiding deadlock by blocking while waiting for other states to become available. If it succeeds in acquiring all the necessary locks, it then checks for eligibility and fires if it can, before releasing all locks and incrementing its counter. Once it has run 1000 times, it joins with the main thread and returns the number of fires it executed.

Evaluation

In each evaluation case, the mode being tested was used to run a simulation of the Dining Philosophers problem in Petri Net form with 4 philosophers around the table.

Random Sequential

Random Sequential mode was tested with an iteration value of 100, 1000, and 10000. Each value was run 10 times and the results averaged to produce the following averages:

Iterations	100	1000	10000
Time Elapsed	0.223 seconds	1.865 seconds	18.088 seconds
Checks	800	8000	80000
Fires	335	3345	33288

Table 1. *Results of Random Sequential mode evaluation.*

In every case, the percentage of checks that provoked a successful fire was within 0.4 of 42%. Higher iteration values naturally required a higher time cost, but since the algorithm requires few nested loops, it appears to inflate linearly.

Random Animated Sequential

Since Random Animated Sequential relies on the user to close the animation window before results are displayed, it is trickier to provide rigid a testing framework. Instead, several simulations were run for variable amounts of time and the results recorded to provide a rough benchmark for ordinary operation. The results are displayed below:

Experiment	1	2	3
Time Elapsed	6.262 seconds	13.811 seconds	44.680 seconds
Checks	320	752	2456
Fires	143	313	1050

Table 2. *Results of Animated Random Sequential mode evaluation.*

As with Random Sequential mode, the percentage of checks that resulted in a successful fire operation was in the environment of 40-45% (44.6%, 41.6%, and 42.8% respectively), and the time taken for the simulation increased linearly with the number of checks performed. As expected, the animation operations significantly increased the processing overhead, requiring over 30x more time for less than half the checks in the first experiment.

Concurrent

Concurrent execution presented a further dilemma in the implementation of the project: while it was deemed important to provide a concurrent implementation for the project in order to demonstrate that it could be done, the necessity of ensuring atomicity in the eligiblity and fire transactions meant that while more checks could potentially be made in the same time period, much fewer would result in successful fires, as they would require no other thread to be checking any of the required states to proceed. A solution loosening the requirements for atomicity and allowing threads to block while waiting for states to become available was considered, but in practice this led to deadlock within milliseconds, as predicted.

Concurrent mode was tested in a similar manner as Random Sequential, using iterations of 100, 1000, and 10000 for each thread, and introducing a small sleep duration of 0.001 seconds between loops for each thread. The results are displayed below:

Iterations	100	1000	10000
Time Elapsed	0.200 seconds	1.995 seconds	19.940 seconds
Checks	800	8000	80000
Fires	65	695	6793

Table 3. *Results of Concurrent mode evaluation.*

As expected, Concurrent mode allowed for far fewer successful fires, since so many comparisons had to be scrapped immediately when it was discovered another transition held a lock. Time taken was within 10% of Random Sequential's result at every iteration level.

It should be noted, however, that these results should not be directly compared with Random Sequential, since an artificial sleep time was added to space checks apart further. Before the sleep was added into the sequence, Concurrent mode was able to perform 800 checks across 8 threads within 0.002 seconds, but only executed 8 fires during that period. As a result, though Concurrent can perform operations at its peak drastically faster that Random Sequential, it produces far fewer successes since threads very rarely have time to secure all the locks they need before another thread tries to snap them up as well.

Future Work

The original intent for **Petri Dish** was to create an application interface that the user could directly manipulate to create Petri Nets, populate them with tokens, and set them running to see the results in real time. That proved monumentally too widely-scoped for the allotted time, and thus **Petri Dish** was narrowed down to one token per state and input via text file. Future work could be undertaken in the direction of a full user interface: snapping controls, parametrically-generated spline connections between states & transitions, and a click-and-drag creation method for nodes.

Additionally, Concurrent mode shows promise despite its less impressive results. Modeling a graphical representation of a concurrent system using concurrent systems would seem from a distance to be the logical choice; however, the matplotlib module did not lend itself to supporting multi-threaded operations particularly well. The frame updates must be made within the animation() function[5], to which there can be only one call per frame. Future work in this direction would entail a queue-like data structure to mitigate this issue, into which threads could place their data for frame updates, and the animation system could pull from this queue as needed.

Finally, and perhaps most trivially, support for more than one token per state would open the simulation up to handle more complex Petri Nets, including those involving finite buffers, queues, limited data access, semaphores, and so forth. The project was limited to one token to make rendering a simpler process; support for more than one would entail a system to populate states with tokens spread out according to a spacing algorithm, and resize states dynamically to accommodate larger or smaller token payloads.

References

[1] Hunter, J. D. et al. "MatPlotLib: A 2D graphics environment." https://matplotlib.org/

[2] Jagusch, Adrian. "APO - A Free Online Petri Net Editor." https://adrian-jagusch.de/2017/02/ apo-a-free-online-petri-net-editor/

[3] kwasaki, y-mitsui, youjirouharie. "HiPS: Hierarchical Petri Net Simulator." https://sourceforge.net/projects/hips-tools/

[4] N.J. Dingle, W.J. Knottenbelt and T. Suto. "PIPE 2: Platform-Independent Petri net Editor 2." http://pipe2.sourceforge.net/

[5] VanderPlas, Jake. "Matplotlib Animation Tutorial." https://jakevdp.github.io/blog/2012/08/18/matplotlib-animation-tutorial/