TECHNICAL UNIVERSITY OF ATHENS
DEPARTMENT OF MECHANICAL ENGINEERING
MSC IN AUTONOMOUS CONTROL SYSTEMS AND ROBOTICS

ADVANCED MANUFACTURING SYSTEMS
(CIM - INDUSTRY 4.0)

DESIGN AND IMPLEMENTATION OF A PETRI NETWORK SIMULATOR IN PYTHON

Ronaldo Tsela - 02124203
Fall 2024-2025

Contents

List of Figures

List of Tables

1. Purpose

The purpose of this project is to develop a Petri net simulator as part of the first task in the "*Advanced Manufacturing Systems - (CIM - Industry 4.0)*" lecture series for the MSc program in "*Autonomous Control Systems and Robotics*" at NTUA.

2. Overview

This document describes the design, usage, and evaluation of a simple Petri network simulator program developed in Python. The simulator provides an easy-to-use API for defining the structure of a Petri network and specifying termination conditions. In addition to supporting classical Petri nets, the simulator can also cope with inhibitory arcs. Accompanying the simulator is visualization and data exportation logic as well developed in the same Python package, that translates the results of the Petri network simulator into a graphical network representation and properly formatted files for further processing.

3. Introduction

Petri nets are a mathematical modeling tool used to describe and analyze systems characterized by concurrency, synchronization, and resource sharing [1][3]. In the context of manufacturing systems, interest in Petri nets has emerged from the need to define and model discrete production systems. Key characteristics of these systems include parallelism and non-determinism, which imply that actions within the systems can be executed in multiple ways and in parallel. During the design of such systems, it is easy to overlook significant points of interaction, potentially leading to malfunctions during operation. To manage the complexity of modern parallel systems, it is essential to provide methods that enable verification and correction before deployment. However as the size of the Petri network increases, the mathematical operations required to analyze such systems become demanding in terms of computational power. Moreover, some enhanced Petri nets variations cannot be formally analyzed at all. Therefore one approach to address these challenges is to build a model of the system and analyze it using a simulated environment through a simulation computer program [2]. This is the task that this project addresses. Before delving into the technical aspects of the development of the simulator, it is considered important to first understand the composition and functionalities of a Petri network, which will directly define the way we model the simulator software.

As a mathematical model a Petri net is characterized as a directed bipartite graph composed of two types of elements: places and transitions. Places communicate with transitions via directed arcs. A place communicates directly only with a transition. Places with places and transitions with transitions do not connect to each other directly. A place from which an arc runs to a transition is called an input place, while the places to which arcs run from a transition are referred to as the output places of that transition respectively [1][3].

Places in a Petri net may contain a discrete number of marks called tokens. Any distribution of tokens across the places represents a configuration of the network commonly known as a marking. A transition in a Petri net can fire if it is enabled which occurs only when there are sufficient tokens in all its input places, as defined by the weights assigned to the arcs connecting the input places to the transition. When a transition fires, it consumes the required input tokens and creates tokens in its output places according to the weights of the arcs connecting the transition to those output places. Each firing of a transition is atomic, meaning it is a single, non-interruptible step[1][3].

Although Petri nets are a powerful mathematical tool, the need to model complex systems and behaviors lead to a set of extensions to the classical Petri networks. There are multiple such extensions [2][3]. In this project we only cope with the extension that permits inhibitory arcs in the network. An inhibitor arc imposes the precondition that a transition may only fire when the connected place is empty. Unlike regular arcs, which enable transitions when tokens are present, an inhibitory arc blocks the firing of the transition when tokens exist in the linked place [1] (similar to a *not* operator in the Bollean algebra logic). This inhibitory feature allows for more precise control over transitions, creating conditions where certain transitions can only fire if specific places are empty. This is particularly useful for representing situations where an action should only occur when a particular resource or condition is absent.

Using the definitions provided above for Petri networks and their operations, we developed a software simulator that replicates these processes. The following section details the development of the Petri network simulator in Python as well as a user manual covering everything from installation of the package to running a simulation script. The final section presents and analyzes the results obtained from simulations performed with this tool on given Petri nets.

4. Development Methodology

4.1. Requirements

The first step in every development project is to define a clear set of system requirements and functionality goals. This ensures that the system will perform as intended and guides the development stages appropriately. **Table 1** below provides the complete list of requirements set for this project.

Table 1 - The list of requirements for the Petri net simulator program.

Number	Definition
Rq.1	The simulator must support up to 20 places.
Rq.2	The simulator must support up to 20 transitions.
Rq.3	The simulator must support inhibitor arcs.
Rq.4	The simulator must randomly select one of the active transitions at each step.
Rq.5	The simulator must output a list of all markings up to the last marking.
Rq.6	The simulator must output a list of activated transitions for each marking.
Rq.7	The simulator must halt operation when a deadlock is detected.
Rq.8	The simulator must provide real-time status updates to the user.

4.2. Programming Language

To develop this simulator, Python was chosen as the programming language due to its versatility, ease of use, and for its object-oriented nature. Python is well-suited for modeling the operations of a Petri network, as described in the *Introduction* section, while meeting the specified requirements. Also its scripting nature and platform portability are ideal for rapid development and testing, while its capabilities for data visualization simplify the representation of the results. Furthermore, Python is highly hardware and operating system agnostic which means that it has high compatibility across different hosting computer systems. All of the above make the selection of Python an excellent choice for this project, ensuring the flexibility and timely completion of the task with good results while in the given timeframe.

4.3. Data Structures and User Interface Methods

To manage each basic component that builds a Petri network, Python classes are implemented which contain methods that facilitate easy handling of places, transitions, tokens, and arcs within a network. A Petri network afterall comprises multiple instances of these elements. The classes implemented are as follows:

1. The *Place* class which manages place nodes and their tokens.
2. The *Transition* class which manages transition nodes along with their input, output, or inhibitory arcs.
3. The *PetriNet* class utilizes the former *Place* and *Transition* instances to construct and manage the entire network.

The PetriNet class serves also as the primary interface for users, abstracting the low-level implementations of places, transitions, and arcs and presenting a higher-level user interface and API .

This class includes intuitive methods to add elements, construct the network, export data in commonly used formats, and simulate network behavior iteratively until specified user conditions are met. The following section will further elaborate on the simulation logic. **Table 2** below provides a list of key methods in the PetriNet class that structure the high-level user interface and API.

Table 2 - The PetriNet class methods that are used to implement the user interface and API

Method	Parameters	Description
add_place()	name tokens	This method adds a place node to the Petri net with a specified name and initial token count.
add_transition()	name	This method adds a transition node to the Petri net with a specified name.
add_input_arc()	transition place weight	This method adds an input arc from a place to a transition with a specified weight.
add_output_arc()	transition place weight	This method adds an output arc from a transition to a place with a specified weight.
add_inhibitor_arc()	transition place	This method adds an inhibitor arc from a transition to a place.
export_structure()	structure file path	This method exports a JSON-formatted file that describes the structure of the Petri net.
simulate()	target transition activation count log file path	This method runs the Petri net's simulation loop until the target transition fires for the specified number of activations. At each step, the Petri net's configuration (marking) is logged to the specified file in JSON format.

In addition to the aforementioned core methods contained in **Table 2**, the following functions are implemented to support data visualization and format conversion:

1. *visualize_petri_net*(): Accepts the Petri net structure file and simulation log file produced by the simulator core and generates a visual representation of the Petri net.
2. *json_to_txt*(): Converts JSON simulation results to a comma-separated TXT file format for easier readability and further processing.
3. *print_to_console()*: Prints the content of the TXT converted simulation log file to the terminal emulator of the host computer for local display of results.

4.4. Simulation Loop

The method *simulate()* implements the core of the system: the simulation loop. The simulation continues until either the target transition activation counter reaches the specified number of activations or the system encounters a deadlock. At each simulation step, a list of all enabled transitions is generated. If no transitions are enabled at a given step, a deadlock is detected, and the simulation halts. Otherwise, if there are enabled transitions, one is selected at random to fire. Upon firing, tokens are consumed from the input places via input arcs, and tokens are created in the output places via output arcs. The number of tokens consumed or created is determined by the arc's weight. After each step, the state of the network is recorded in the output log file, and the total step counter is

incremented by one. If the chosen transition is the target transition, the target transition activation counter is also incremented by one.

4.5. Visualization and Data

The structure of the Petri network is exported as a JSON file containing all places (through the method *export_structure*()), transitions, and arcs that define the network. For places, only the name and initial token count are needed, while for transitions, only the name is required. Arcs include information on their source and destination (since they are directional), type (input, output, or inhibitor), and assigned weight. The snippet below shows the exported format for a simple Petri net, as depicted in **Fig. 1**. Currently, the API does not support configuring the Petri net directly from this output JSON file (by reading it as an input). In this version instead the JSON file is used solely for visualization purposes by the *visualize_petri_net()* utility function.

The simulator output also includes a JSON file that records each configuration of the network at each simulation step (total marking), along with the transition activated at each event or a deadlock indication if one occurs. This JSON file is used again by the visualization function to generate a graphical representation of the simulation and is also processed by *json_to_txt()* utility function as well to create a more user-friendly TXT file in comma-separated format which then can be either processed or displayed immediately to the host computer terminal emulator through the *print_to_console()* function. Here visualizations are based on the *networkx* Python package which is used to construct graph widgets, and the visualization of the graph is rendered with *matplotlib* via the *pyplot* module.

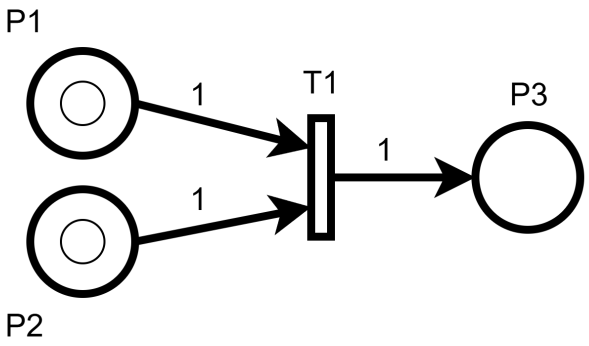

Fig. 1 - The example Petri net which is described by the snippet shown at the left.

```
{
    “places”: [
        {
            “name”: “P1”,
            “tokens”: 1
        },
        {
            “name”: “P2”,
            “tokens”: 1
        },
        {
            “name”: “P3”,
            “tokens”: 0
        },
    ],
    “transitions”:[
        “T1”
    ],
    “arcs”:[
        {
            "from": "P1",
            "to": "T1",
            "type": "input",
            "weight": 1
        },
        {
            "from": "P2",
            "to": "T1",
            "type": "input",
            "weight": 1
        },
        {
            "from": "T1",
            "to": "P3",
            "type": "output",
            "weight": 1
        }
    ]
}
```

5. User Manual

This package is written in Python, making it largely hardware- and operating system-agnostic, so it should operate similarly across different systems. However, as development and testing were conducted in a Linux-based environment, it is recommended using a Linux environment to run the simulator. Most commands should also work on Windows via Windows Subsystem for Linux (WSL), or by using a virtual machine on a Windows host running one of the many Linux distros available. The only requirement is Python version 3.5 or higher. Other than that, there are no additional dependencies to run the simulator. The required packages are installed by the package handler itself, if they are not already installed in the host system.

5.1. Package Installation

To keep this package separate from other Python packages on your system, it is best practice to use a virtual environment. To create and activate a virtual environment, on a terminal emulator run within the project folder the following:

```
$ python3 -m venv petri_net_env
$ source ./petri_net_env/bin/activate
```

Navigate to the *./petri_net_sim_package* directory and run the following command to install the package locally on your computer (within the virtual environment you just created).

```
$ pip3 install -e .
```

Upon successful installation of the package and the required internal packages needed for the implementation, the simulator is ready for use.

5.2. Simulator Set-Up

Within the project folder there is a template Petri net simulator Python script called *run.py*. Open the Python script. You must insert code in the section beginning with ##### *Define here your Petri network* #### and ending with ##### *End of network definitions* ####. Avoid modifying any other parts of the code. To model the Petri network for simulation follow these steps:

- ***Step 1 Add places :*** Use *net.add_place(place_name, initial_tokens)* to add a place node to the graph. Place names are typically assigned names as *Pi*, where *i = 1, 2, 3, ... n* for *n* places, but any other name is welcome. If the initial token count is not specified for a place, by default is assigned to 0.
- ***Step 2 Add transitions :*** Use *net.add_transition(transition_name)* to add a transition node to the graph. Transitions are named usually with *Ti*, where *i = 1,2, 3, ... m*, for *m* transitions, but any other name is permissible.
- ***Step 2 Setup the arcs :*** The simulator supports three types of arcs that connect places and transitions: input, output, and inhibitor.
    - Use the *net.add_input_arc(transition_name, place_name, weight)* to add an input arc from the specified place to the specified transition with the given weight. If the weight is not defined then the arc will be assigned unit weight.
    - Use *net.add_output_arc(transition_name, place_name, weight)* to add an output arc from the specified transition to the specified place with the specified weight. If the weight is not defined then the arc will be assigned unit weight.
    - Use *net.add_inhibitory_arc(transition_name, place_name)* to add an inhibitory arc from the specified place node to the specified transition node.
- ***Step 4 Define termination conditions :*** Assign to the *target_transition_name* variable the desired target transition node and *total_transition_activations* to the total number of

activations for that transition. These values will serve as the simulation's termination conditions. Specifically the simulation will run for as many steps as required for either the target transition node is fired for the total transition activation parameter defined or a deadlock occurs. In both situations the simulation will terminate.

5.3. Simulation Execution

To run the newly configured simulation, simply execute the program as follows:

$ python3 run.py

In folder *./simulations/* you will find three ready-to-execute scripts that model three Petri networks defined in the requirements of this task as described in the section *Results*.

5.4. Outputs

The simulator upon execution outputs two JSON files: one describes the network's structure, while the other contains the simulation results. These files are then parsed as input to the visualization script, which produces a graphical representation of the changes that occurred in the Petri network throughout all the simulation steps. Each step is graphically visualized with a timestep in between steps set to 1 second. Also the content of the simulation results JSON file is extracted into a more intuitive and easy to work with TXT file. You can find such files in *./results/* folder produced from the three simulation files respectively.

6. Experimentation and Results

6.1. Analyzing Network 1 : Deadlock

The network simulated in **Fig. 2** is prone to deadlock conditions. Specifically if the simulation is executed for several times, on average approximately one-third of runs result in a deadlock. **Fig. 3** illustrates the network's state at simulation step 36, where a deadlock is detected. The target, transition node t16, is activated twice of 5 total times that the termination condition specifies.

Another sequence of activations leading to deadlock is as follows: {t1, t6, t2, t3, t4, t5, t1, t6, t11, t2, t3, t12, t13, t4, t5, t14, t15, t1, t16, t2, t11, t6, t12, t13, t3, t14, t4, t5, t16, t1, t2, t3, t4, t6} with marking {0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 5, 0} for places {p1, p2, p3, p4, p5, IN, ^IN, WIP, ^WIP, p11, p12, p13, p14, p15, FP, ^FP, R}

On the other hand, when sequences such as: {t1, t2, t3, t4, t5, t6, t11, t12, t13, t1, t2, t3, t14, t6, t15, t16, t4, t5, t1, t11, t2, t12, t13, t14, t6, t3, t15, t16, t4, t5, t11, t12, t1, t13, t14, t2, t3, t6, t15, t4, t16, t5, t11, t12, t13, t1, t14, t2, t3, t15, t16, t6, t4, t5, t1, t2, t3, t6, t11, t12, t13, t14, t15, t4, t5, t11, t12, t1, t16} with marking {0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 5, 1} for places {p1, p2, p3, p4, p5, IN, ^IN, WIP, ^WIP, p11, p12, p13, p14, p15, FP, ^FP, R}, allows the network to operate without ant deadlock after five activations of t16.

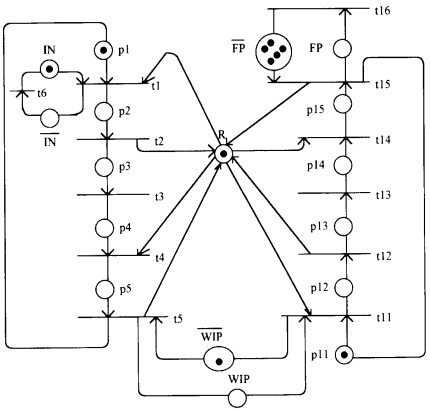

Fig. 2 - The Petri network with deadlock.

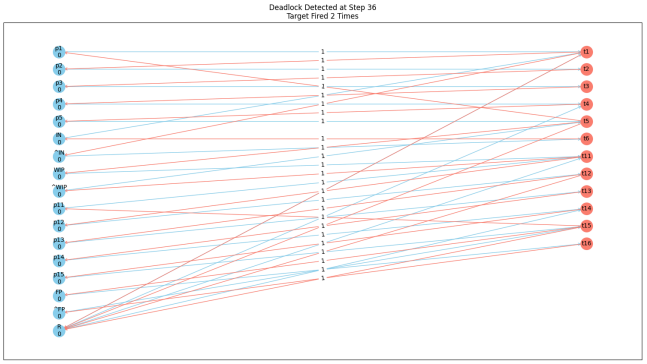

Fig. 3 - The Petri network simulation last marking with deadlocks detected.

6.2. Analyzing Network 2 : Deadlock-Free

By adding an additional pair place-transition nodes (virtual) in between the t4 and the WIP as depicted in **Fig. 4**, then we get rid of the deadlock present in the network of **Fig. 2**. This is verified by running the simulator on this particular configuration. **Fig. 5** depicts the network's state after a successful simulation run, ending at step 68, with the final marking: {0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 5, 1} for places {p1, p2, p3, p4, p5, pd, IN, ^IN, WIP, ^WIP, p11, p12, p13, p14, p15, FP, ^FP, R} respectively.

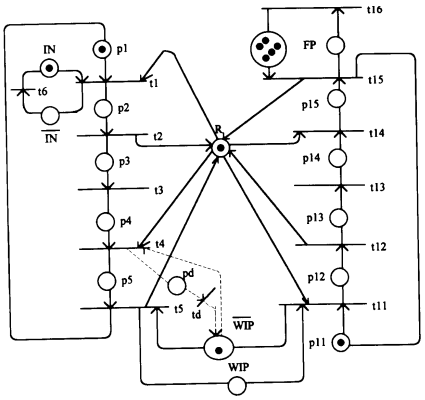

Fig. 4 - The Petri network without deadlock.

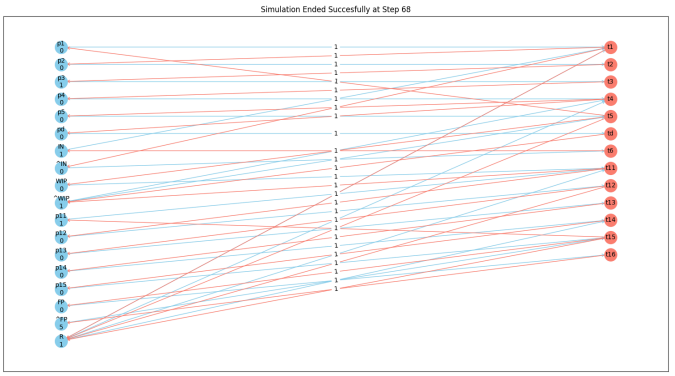

Fig. 5 - The Petri network simulation last marking without deadlocks.

6.3. Analyzing Network 3 : Inhibitory Arc

In this network configuration, depicted in **Fig**. **6**, an inhibitory arc allows transition t11 only when place p1 is empty (and of course the rest of inputs provide the right tokens), setting a priority constraint, which prioritizes the place p1 to receive the token from place R. Deadlock still occurs in some sequences on this network configuration. **Fig. 7** shows an example where a deadlock arises at step 21. Another deadlock appears on step 35, when marking is {0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 5, 0} for places {p1, p2, p3, p4, p5, IN, ^IN, WIP, ^WIP, p11, p12, p13, p14, p15, FP, ^FP, R}. In all situations however no matter the conclusion (deadlock or not) the t1 transitions always fires before t11.

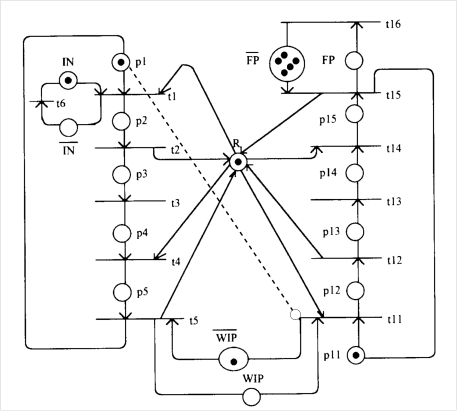

Fig. 6 - The Petri network with inhibitor arcs.

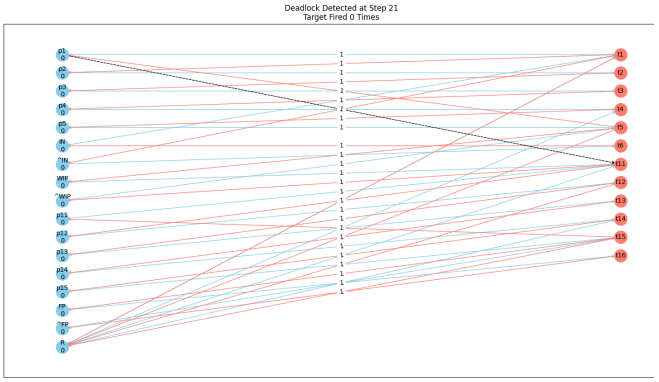

Fig. 7 - The Petri network simulation last marking with inhibitor arcs.

7. References

[1] Petri net.
Wikipedia.
https://en.wikipedia.org/wiki/Petri_net.

[2] Πετρόπουλος Κωνσταντίνος.
Διπλωματική Εργασία, ΕΜΠ.
Σχεδιασμός και Υλοποίηση Ελεγκτών σε Ευέλικτο Σύστημα Κατεργασιών.
2018.

[3] Helios.
Προηγμένα Συστήματα Κατεργασιών (CIM-INDUSTRY 4.0)
Παρουσιάσεις, Σημειώσεις και Παραδείγματα.
2024.

[4] NetworkX - Network Analysis in Python
https://networkx.org/