PNET MODULE : EMPOWERING PETRI NET MODELING AND SIMULATION

A Thesis submitted
in Partial Fulfilment of the Requirements for the Degree of

MASTER OF SCIENCE
In
Applied Mathematics

by
Charu Singh
(2K22/MSCMAT/05)

Shefali
(2K22/MSCMAT/36)

Under the supervision of

Dr. Payal

Department of Applied Mathematics

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daultapur, Main Bawana Road, Delhi-42

May, 2024

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

CANDIDATE'S DECLARATION

We, (Charu Singh) 2K22/MSCMAT/05 and (Shefali) 2K22/MSCMAT/36 hereby certify that the work which is being presented in the thesis entitled "PNet Module: Empowering Petri Net Modeling and Simulation " in partial fulfilment of the requirement for the award of the Degree of Master of Science, submitted in the Department of Applied Mathematics, Delhi Technological University is an authentic record of our own work carried out during the period from August 2023 to April 2024 under the supervision of Dr. Payal

The matter presented in the thesis has not been submitted by me for the award of any other degree of this or any other Institute.

Candidate's Signature **Candidate's Signature**

This is to certify that the student has incorporated all the corrections suggested by the examiners in the thesis and the statement made by the candidate is correct to the best of our knowledge.

Signature of Supervisor **Signature of External Examiner**

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

CERTIFICATE BY THE SUPERVISOR

Certified that Charu Singh (2K22/MSCMAT/05) and Shefali (2K22/MSCMAT/36) have carried out their search work presented in this thesis entitled "PNet Module: Empowering Petri Net Modeling and Simulation" for the award of Master of Science from Department of Applied Mathematics, Delhi Technological University, Delhi, under my supervision. The thesis embodies results of original work, and studies are carried out by student themselves and content of the thesis do not form the basis for the award of any other degree to the candidates or to anybody else from this or any other University/Institution.

Place: Delhi
Date: June, 2024

Dr. Payal
SUPERVISOR
DEPARTMENT OF APPLIED MATHEMATICS
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-
110042

PNET MODULE: EMPOWERING PETRI NET MODELING AND SIMULATION

(Charu Singh and Shefali)

ABSTRACT

Petri Nets provide a way of representing changes over time which is structured and is widely used for representing models of various systems. However, expertise in programming is required for working with Petri Nets and can often be complex. This is where PNet, a Python library that is intended to make working with Petri Nets much easier, is introduced. Without needing to be a coding expert using PNet, you can define your Petri Net using a simple text based language. You can use regular Python functions even if you need more complex rules for how things change.
To demonstrate the simplicity of PNet, we provide three examples: one that shows you how to bake a cake, another that models the spread of diseases, and the last one that describes the evolution of molecule count in a system. With PNet, you can easily bring your ideas to life and simplify the process of working with Petri Nets.

ACKNOWLEDGEMENTS

At the outset of this report, we extend our heartfelt appreciation to all individuals who have supported us in completing this dissertation. Without their proactive direction, assistance, collaboration, and support, we could not have advanced toward achieving the desired outcomes. Dr. Payal provided diligent assistance and support that enabled us to complete our dissertation, for which we are eternally grateful. We express our sincere appreciation to each other for working together to complete this project while preserving our individuality. We are thankful that Delhi Technological University provided this opportunity to us. We additionally express our sincere gratitude and respect to our parents as well as other family members, who have always provided us with both material and moral support. Finally, but just as importantly, we would like to express our heartfelt gratitude to all of our friends who supported us in any way during this effort.
This quick acknowledgement does not imply a lack of gratitude for anything.

Thanking You

CHARU SINGH
SHEFALI

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction to Petri Net

Petri nets, devised by Carl Adam Petri in the 1960s, are graphical and mathematical tools utilized for representing systems that exhibit concurrency, synchronization, and mutual exclusions. For understanding and analyzing systems Petri Nets are an invaluable tool. Petri net theory allows mathematical modeling of a system as a Petri Net. Analyzing the Petri net reveals significant details regarding the portrayed system's dynamic behavior and structure. This in turn, we can use to get access to the modeled system in order to suggest improvements or alterations. Therefore, the formation of a theory of Petri Net depends on the use of Petri nets for the purpose of designing and modeling systems.

1.2 Modeling

Petri Nets can be used to model a variety of phenomena. Sometimes, we create a model of a system rather than studying it directly. A model is like a mathematically expressed, cut-down version, which gives us the important aspects of the thing we're studying. By working with this model, our main goal is to learn new things about the real phenomenon without having to deal with the challenges or expenses of directly manipulating it. The majority of modeling activities require the utilization of mathematical principles. Many physical processes commonly employ numerical representations to describe them, while inequalities or equations illustrate the relationship between different features. Mathematics can be used to define elementary concepts like location, mass, forces, and momentum in domains such as engineering and the natural sciences. In order to utilize modeling efficiently, it is important to have a comprehensive understanding of both the subject being modeled and the underlying principles of the modeling technique. A significant amount of importance has been acquired by mathematics due to its ability to facilitate the modeling of phenomena in several scientific disciplines. To analyze and represent phenomena that undergo continuous change, calculus was devised, such as the concepts of velocity, position and acceleration in the field of physics.

The efficiency and prevalence of modeling have significantly increased due to the introduction of high-speed computers. The conversion of a system into a mathematical model and subsequently providing these instructions to a computer can lead to the conduction of simulations. As a result, we can now simulate larger and more intricate systems than ever before. Accordingly, extensive studies have been conducted on computer hardware and computer modeling methodologies. In modeling, computers have a dual purpose: they function as tools for modeling, and they are also subjects of modeling themselves.

1.3 Components of a Petri Net

A basic Petri net is made of the following elements:

Places: Which are represented by circles and denote conditions or states of the system.

Transitions: They are represented by rectangles or bars; transitions signify events or activities that can alter the system's state.

Tokens: They are placed inside the places, tokens indicate the presence or absence of certain conditions. Mathematically, we can think of them as markers of a place.

Arcs: Places and transitions are related, and vice versa by Arcs. The flow of the system and the determination of the input and output conditions for transitions are defined by arcs.

1.4 Operational Semantics

The way a Petri net behaves is governed by the firing of transitions, which are as follows:

Enabled Transitions: If all the input places (places connected to the transition by arcs that are incoming) have the required number of tokens, then it is said for a transition to be enabled.

Firing: A transition fires when it consumes tokens from its input places and produces them in its output places.

1.5 Execution Rules

Initial Marking: The arrangement of tokens in the locations determines the starting condition of the system. We refer to this distribution as the initial marking.

Enabling of Transitions:

- For a transition to be considered enabled, every input place (a place with an arrow pointing towards the transition) should contain a minimum of one token.
- In other words, we can define it as, the required conditions (represented by tokens in the places) that must be met for the event (transition) to potentially occur.

Firing of Transitions:

- Triggering of an enabled transition leads to the removal when the transition takes one token from each input site and adds one token to each related output spot.
- The happening of the event is represented by firing, which causes a change in the system's current condition.
- The moving of tokens from the input places through the transition to the output places is described by this process.

1.6 Properties of Petri Nets

Several fundamental properties that can be derived from Petri net models are as follows:

Boundedness: A Petri net is considered bounded if there exists a maximum limit on the amount of tokens that any spot can hold. **Liveness:** A Petri net is considered live if it is possible to fire any transition from any reachable state at any point. **Deadlock:** It refers to a state in which the system halts due to the absence of enabled transitions.

Conservation: The net is conservative if the total number of tokens remains constant throughout all possible firings.

1.7 Variants of Petri Nets

Colored Petri Nets (CPN): Extension of the basic model by assigning colors to tokens, which allows the model to be more expressive and detailed. **Timed Petri Nets:** Incorporates information related to timing and enables the modeling of systems with time- dependent behaviors. **Stochastic Petri Nets (SPN):** The introduction of probabilistic behavior is done in order to make them suitable for performance dependability and assessment studies.

1.8 Analysis Methods

Petri Net Theory gives us various methods to analyze the properties of systems and their behavior. Here are the main analysis methods described:

1. **Reachability Analysis:**

   What It Is: This method lets us know that by firing a sequence of transitions, a particular state (or marking) can be reached from the initial state.

   Why It Matters: It helps determine whether the system can enter undesirable states, such as unsafe conditions or deadlock.

   How It Works: The software produces a reachability graph that illustrates all the reachable markers from the initial state by executing transitions. This allows for a comprehensive view of all feasible system states.

2. **Invariant Analysis:**

   What It Is: Using this method, we can identify invariants, which are the conditions that remain true and do not depend on how transitions fire.

   Types:

- **Place Invariants:** The constant linear combinations of places. They aid in the verification of conservation properties, guaranteeing the absence of creation or destruction of certain resources.
- **Transition Invariants:** These are defined as the sequences of transitions that leave the marking unchanged, indicating repeated patterns or cyclical behavior.
  Why It Matters: Invariants, help us verify the consistency and correctness of the system, ensuring its behavior as intended over time.

3. **Liveness Analysis:**

What It Is: This method verifies whether the Petri net is live, implying that each transition has the potential to initiate from a reachable marker.

Why It Matters: Liveness ensures that the system has no deadlocks and that all parts of the system can eventually be activated and remain operational.

How It Works: By examining the reachability graph or using mathematical techniques like the rank theorem, we can get to know if all transitions have the potential to fire.

4. **Boundedness Analysis:**

   What It Is: This method helps us determine if there is a restriction on the maximum amount of tokens that can accumulate in each location.

   Why It Matters: Ensuring boundedness helps manage system resources effectively by preventing resource overflow.

   How It Works: By calculating place invariants or examining the reachability graph, we can establish upper limits on the number of tokens for each place.

5. **Coverability Analysis:**

   What It Is: This method determines whether a marking can be reached where the number of tokens in one or more places exceeds a certain threshold.

   Why It Matters: It helps identify situations where resources might be exhausted or overused.

   How It Works: By generating a coverability tree, which is a modified version of the reachability graph, we can see if and when certain token counts are exceeded.

6. **Performance Analysis:**

   What It Is: This method helps to evaluate the efficiency and performance of the system modeled by the Petri net.

   Why It Matters: It helps to optimize resource usage, identify bottlenecks and improve overall performance of the system. **How It Works:** Techniques like timed Petri nets (where transitions have firing delays) and simulation are used to measure and analyze performance metrics like latency, utilization and throughput.

CHAPTER 2

INTRODUCTION TO PNET

2.1 Introduction

Petri Nets have diverse uses in domains such as system modeling, biochemistry, and software engineering, signal transduction networks, gene control networks and namely in biochemical reactions. For instance, Liu and Heiner used Petri Nets to investigate biological reaction networks. A unified Petri Net framework was designed to model and analyze the networks created by them. Petri Nets are valuable for analyzing several process features, including reachability, termination, boundedness, safety, reversibility, liveness, coverability, home state, fairness, and persistence. These features can be analyzed using techniques such as reachability graphs, coverability trees, state equations and incidence matrices. Multiple libraries have been created to simulate Petri Nets, simplifying their utilization in various programming contexts. SimForge GUI is incorporated into OpenModelica, but MATLAB provides the Petri Net Simulink Block (PNSB) as an option. Python offers the SNAKES library, developed by Pommereau, which enables the implementation of Petri Nets using a sophisticated object oriented methodology. This approach involves representing transition rules and tokens as Python objects. This strategy gives adaptability but may necessitate a more challenging learning process, particularly for individuals who are not acquainted with

Python. Translating Convert a text-based definition of a Petri Net into a model using SNAKES can be a difficult task. In addition, SNAKES may not be capable of handling intricate transition rules that require implementation as functions. Nevertheless, SNAKES provides benefits such as the integration of plugins, , the ability to transform implemented Petri Nets into the C language and tools for Petri Net analysis. A package designed for Petri Net modeling is being presented in this paper. The main objective is to reduce the additional costs related to object-oriented programming by providing Python functions as an alternate form of transition rule. This approach helps to make it easier for beginners to get started with Petri Nets, serving as a stepping stone before moving on to more complex libraries like SNAKES. PNet is now part of COPADS, a Python library of data structures, and algorithms which doesn't rely on any third-party dependencies.

2.2 Description of PNet

In this section, we'll explain how to utilize PNet by delineating the necessary procedures to construct a simulation. There are five primary stages involved:

- **Setting up the Petri Net:** This mainly involves creation of the structure of the Petri Net.
- **Adding places or states:** These are the conditions or locations within the system.
- **Adding transition rules:** These gives the definition of the actions that can occur between places/states.
- **Running the simulation:** This involves execution of the Petri Net model to observe how it behaves over time.
- **Generating the results file:** After simulation is done, the outcomes are recorded and saved for analysis.

To construct a Petri Net using PNet, we begin by importing the PNet module and thereafter instantiate the PNet class from the said module. Next, you add states or places by using the `add_places` method to the Petri Net simulation.

This method requires two parameters:

- A dictionary representing the initial tokens.
- The name of the place

The dictionary consists of token names as keys and the number of tokens of each type as values. This enables locations to accommodate multiple varieties of tokens. For example, if you possess a container holding 1000 yellow seeds and 1000 green seeds, you can depict it as net.add_places('vessel', {'yellow_seeds':1000,

'green_seeds':1000})

The value of the 'green_seeds' attribute is set to 1000.

Occasionally, we come across situations where an inexhaustible quantity or requirement is necessary, such as when contemplating an infinite number of births or the Earth as an infinite reservoir of charges in electronics. To handle this, we introduce a special place called `ouroboros`, named following the infinity symbol in math. This place is defined using a certain set of criteria or parameters having an unlimited number of "U" tokens to represent infinity.

In order to complete the third phase, you need to incorporate transition rule(s) by utilizing the add_rules method. We now assign a name to each transition rule. Transitions serve as a pathway for tokens to transition between places, while the rules take care of the mechanics of this transition. A transition rule typically contains a destination place and a source place to define the movement of tokens. It also specifies source and destination token types to ensure clarity. The valuation of the transition processes is then conducted by employing logical operators, taking into consideration the present token values. These operators are resolute by criteria, which represent the desired outcome after the transitions occur. The five types of rules includes ratio, step, function, incubation, and delay rules. The implementation of transition rules is contingent upon the time interval. While it is preferable for each rule to specify only one transition, in practice, a single rule can trigger multiple transitions. This is because PNet enables the specification of many transitions within a rule, which serves as a simple and concise method.

2.2.1 Step Rule

A step rule operates in a step-by-step manner, triggering at each time-step. It requires specifying the origin place and the token at that place, as well as the destination place and the affected token there. This defines a single transition. For instance, consider a vessel containing yellow and green seeds. The given step rule outlines the process of exchanging a single seed at each time step. The rule is added to the network using the net.add_rules() function, using the parameters 'swap_seed', 'step', and the following seed exchange: 'B1.yellow_seed -> B2.yellow_seed; 1'.

The value of B2.green_seed is assigned to B1.green_seed with a weight of 1. net.add_rules('swap_seed', 'step', ['B1.yellow_seed -> B2.yellow_seed; 1', 'B2.green_seed -> B1.green_seed; 1'])

2.2.2 Ratio Rule

The ratio rule also operates in a step-by-step manner, similar to the step rule. Both rules have similar parameters, but the key difference lies in how they determine when to trigger the execution. Instead of specifying a fixed number of tokens, the ratio rule uses a proportion of tokens to decide when to execute. This proportion is compared against a specified limit using a logical operator. Based on a certain ratio the number of tokens moved can increase or decrease is useful for defining transitions. For instance, imagine we have two vesseles, one filled with yellow seeds and the other empty. We want to move 12% of the remaining yellow seeds from the first vessel to the second. This can be represented as,

```
net.add_rules( 'swap_ratio', 'ratio',['B1.yellow_seeds
-> B2.yellow_seeds; 0.12; \ B1.yellow_seeds < 1; 0' ])
```

2.2.3 Delay Rule

The delay rule is essentially a step rule that has a specific time period between each movement of a token. This means it can create a regular, intermittent pattern of token movement, akin to spiking. For instance, let us transfer 10 seeds from vessel B1 to B2 occurs once every 6th time step, and we can characterize this as thus.

```
net.add_rules( 'interval_transfer’,
'delay',['B1.seeds
-> B2.seeds; 10; 6'])
```

2.2.4 Incubation Rule

The incubation rule represents a period of anticipation preceding a particular action takes place. It involves specifying a timer and a value, which checks provided that
the necessary requirements are fulfilled, one may travel to the intended destination.

As an illustration,

if we want to soak a of seeds for 60 time steps (equivalent to 60 minutes) in a vessel after adding water, and then relocate the soaked seeds into a pot, we can define this using an incubation rule.

```
net.add_rules('soak', 'incubate',

['60; vessel.seeds -> pot.seeds; \ vessel.water

> 0'])
```

2.2.5 Function Rule

The function rule is a customizable condition defined by the user. It's typically used when the predefined step, ratio, delay, or incubation rules don't meet the user's specific needs. However, all types of transition rules can be represented as function rules, making them a more flexible option. Another regular use of function rules is to alter tokens from one type to another. The main disparity among transition rules lie in the situations that activate them. To compute a function rule, you need to specify the source and end places, as well as the initial and final tokens involved. For instance, the earlier described ratio rule.

```
net.add_rules( 'swap_ratio', 'ratio',['B1.yellow_seeds

-> B2.yellow_seeds; 0.20; \ B1.yellow_seeds < 2; 1' ])
```

can be defined by the following function rule:

```
def seed_swap(places): place = places['B1']  n =

place.attributes['yellow_seeds'] if n > 0.0: return 0.0 else:

return 0.20 * n

net.add_rules( 'swap_ratio', 'function',

['B1.yellow_seeds -> B2.yellow_seeds' , seed_swap, 'B1.yellow_seeds >

0' ])
```

Only one parameter is accepted in the function rule(s) that you're working with, which is a dictionary called `places`. This dictionary contains information about different states or places in a network (let's call it PNet). You can access each state or place by using its name as the key to the places dictionary.
To access each state or place, simply use its name as the key in the places dictionary.
Each place or state in the "places" collection is related with tokens, which are implemented as an attributes dictionary. The tokens can be accessed by their respective names.
The simulate method stores simulation results in memory. Therefore, increasing the reporting intervals will generate more reports and result in faster memory usage. On the other hand, there is a generator function "the simulate_yield method" in which there is no pre-storing of all the simulation results are stored in memory. The three parameters necessitated by the simulate method are: the period of time to simulate, the rise of time, and the rate of reporting. However, the simulate_yield method requires only two parameters: the duration of time steps and the time to simulate.

Finally, PNet provides a method to convert the simulation outcomes into a format that is compatible with CSV file output. The current simulation's step count and the status of each token kept in memory will be provided by the reports. Furthermore, you have the option to construct a list that accurately represents the current state of the tokens either for a single step or for the entire simulation. Simulation and reporting are frequently interconnected. An example of how simulation and report creation are coupled is demonstrated in the code given below, which utilizes either the simulate or simulate_yield method. The length of the simulation is set to

100 units. Each timestep represents a unit of time. The simulation will return results every timestep. Use the function net.simulate to run a simulation for a specific length of time, with a specified timestep and report frequency.

To obtain the report tokens, use the net.report_tokens() method. This method is specifically designed for simulating yield. The variable "status" is assigned a list comprehension that iterates over the results of the "simulate_yield" method of the "net" object. The method is called with the arguments "length_of_simulation" and "timestep". The code assigns the value of a list comprehension to the variable "status". The list comprehension iterates over each element "d" in the list "status" and creates a tuple with the first element of "d" and the result of calling the "report_tokens" method of the "net" object with the second element of "d".

```
length_of_simulation = 90

timestep = 1  report_frequency

= 1


# for simulate method
net.simulate(length_of_simulation,timestep,
report_frequency)  status = net.report_tokens() #
for simulate_yield method status = [d for d in
net.simulate_yield(length_of_simulation, timestep)]
status = [(d[0], net.report_tokens(d[1])) for d in
status]
Framework for Petri Nets Typed Applications
class Place(object):
```

Class to represent a place or container in Petri nets. The tokens are represented as a dictionary where each token is represented as a key-value pair. The key represents the type of token and the value represents the number of such tokens. This enables more than one type of tokens to be represented.

```
def _init_(self, name):
```

Contructor method.

```
          self.name = str(name)
self.attributes = {}
class PNet(object):
```

Class to represent a Petri Net or Petri Net typed object.

The places and transition rules are represented as dictionary objects. Places dictionary will have the name of place as key and the Place (pnet.Place object) as value. Transition rules dictionary will have the name of rule appended with a number (in the format of <rule name>_<number> in order to ensure uniqueness) as key and the value is a dictionary to represent the transition rule (the structure of the transition rule dictionary is dependent on the type of rules).

Again recognizing the following types of transition rules are allowed:

- step rule
- delay rule
- incubate rule
- ratio rule
- function rule

Step rule is to be executed at each time step. For example, if 20g of flour is to be transferred from flour vessel to mixer vessel at each time step, this 'add_flour' rule can be defined as:

```
net.add_rules('add_flour','step',['flour.flour
mixer.flour; 20'])
```

A single step rule can trigger more than one token movement. For example, the following step rule simulates the mixing of ingredients into a flour dough:

```
net.add_rules('blend', 'step', ['mixer.flour
-> mixer.dough; 15', 'mixer.water ->
mixer.dough; 10', 'mixer.sugar ->
mixer.dough; 0.9', 'mixer.yeast ->
mixer.dough; 1'])
```

Delay rule acts as a time delay between each token movement. For example, the following rule simulates the transfer of 0.5g of yeast into the mixer vessel:

```
net.add_rules('add_yeast', 'delay', ['yeast.yeast ->
mixer.yeast; 0.5; 10'])
```

Incubate rule is a variation of delay rule. While delay rule is not condition dependent, incubate rule starts a time delay when one or more conditions are met. For example,

```
  net.add_rules('rise', 'incubate', ['10;
mixer.dough -> pan.dough; mixer.flour == 0;
mixer.water == 0; mixer.sugar == 0;
mixer.yeast == 0'])
```

sets a 10 time step delay when all flour, water, sugar, and yeast in the mixer vessel are used up, which simulates the complete mixing into a bread dough. The 10 time step then simulates the time needed for the dough to rise. After 10 time steps, dough in the mixer is transfered into the pan.

Ratio rule is a variant of step rule. Instead of absolute number of tokens to move, the movement is a percentage of the number of tokens. For example,

```
net.add_rules('bake', 'ratio', ['pan.dough ->
pan.bread; 0.3; pan.dough < 1; 0'])
```

will move 30% of the token value from dough in pan to bread in pan. If the token value of dough in pan is less than 1, then the token value of dough in pan will be set to 0.

Function rule is a generic and free-form rule, that takes the form of a Python function. This is usually used when the transition cannot be represented by any other rules. Given a user-defined function, FUNC,

```
net.add_rules('cool', 'function', ['table.temperature >
air.temperature', FUNC, 'table.bread > 0;  table.temperature
> 30'])
```

FUNC will be executed when table.bread > 0 and table.temperature > 30. The returned result of FUNC will be the token transfer from table.temperature to air.temperature.

FUNC takes all the places as a single parameter, such as FUNC(places), where 'places' is a dictionary with the name of each place as key. Token values in any place can be assessed. For example,

```
 >>> place_names = places.keys()

  >>> a_place = places[places_names[0]]

>>> token_set = a_place.attributes.keys()

>>> a_token_value =
a_place.attributes[token_set[0]]

    '''     def _init_(self,
zerolowerbound=True):
```

Contructor method.

Zero lower bound boolean: flag to determine whether number of tokens is bounded at zero. Default = True (the lowest number for tokens is zero)

```
self.places = {}
self.add_places(place_name:'ouroboros',to
kens:{'U':flo at('inf')}) self.rules = {}
self.report = {}  self.losses = {}
self.zerolowerbound = zerolowerbound
self.rulenumber = 1
      def
add_places(self, place_name,
tokens):
```

Method to add a place/container into the Petri Net.

For example, the following adds a "flour" place containing

1000 tokens of flour, which can be seen as a vessel of 1000g of flour:

```
net.add_places('flour', {'flour':
1000})
```

place_name string: name of the place/container tokens dictionary: token(s) for the place/container where key is the type of token and value is the quantity of tokens for the specific type

```
self.places[place_name] =
Place(place_name)
self.places[place_name].attributes =
tokens       def add_rules(self,
rule_name, rule_type, actions):
```

Method to add a transition rule into the Petri Net.

rule_name string: name of the transition rule. This name need not be unique within the model as this

method will append a running rule number to the name
to ensure internal uniqueness.
rule_type string: type of rule. Allowable types are 'step' for
step rule, 'delay' for delay rule, and 'incubate' for incubate
rule. Please see module documentation for the description
of rules.
actions list: describe the action(s) of the transition rule

```
if rule_type not in ['function']: For t
in actions:
       t = [x.strip() for x in
t.split(';')] d = {'type': rule_type,
'movement': None}    if rule_type ==
'step':           movement = [x.strip()
for x in t[0].split('->')]
d['movement']=[(loc.split('.')[0],loc.s
plit('.')[1])
for loc in movement]
d['value'] = float(t[1]) if rule_type
== 'delay':            movement =
[x.strip() for x in t[0].split('->')]
d['movement']=[(loc.split('.')[0],loc.s
plit('.')[1])
for loc in movement]
         d['value'] = float(t[1])
d['delay'] = int(t[2]) if rule_type ==
'incubate': d['value'] = float(t[0])
movement = [x.strip() for x in
t[1].split('->')]
d['movement']=[(loc.split('.')[0],loc.s
plit('.')[1])
for loc in movement]
d['conditions'] = [cond for cond in
t[2:]]          d['timer'] = 0 if
rule_type == 'ratio':
movement = [x.strip() for x in
t[0].split('->')]
d['movement']=[(loc.split('.')[0],loc.s
plit('.')[1])
for loc in movement]
          d['ratio'] = float(t[1])
d['limit_check'] = t[2]
d['limit_set'] = float(t[3])
self.rules[rule_name + '_' +
str(self.rulenumber)] =d
self.rulenumber = self.rulenumber + 1
if rule_type in ['function']: d =
{'type': rule_type, 'movement': None}
if rule_type == 'function':
movement = [x.strip() for x in
actions[0].split('->')]

d['movement']=[(loc.split('.')[0],loc.s
plit('.')[1])
for loc in movement] d['function'] =
actions[1]
     d['conditions'] = [cond.strip() for
cond in actions[2].split(';')]
self.rules[rule_name + '_' +
str(self.rulenumber)] = d
self.rulenumber = self.rulenumber
+ 1
             def _step_rule(self,
movement, value, interval):
```

Private method which simulates a step rule action.
movement string: defines the movement of a token type.

Each movement is defined in the following format:

<source place>.<source token> ->

<destination

place>.<destination

token> value float:

the number of tokens

to move interval

integer: simulation

time interval

```
'''                 source_place        =
self.places[movement[0][0]]
source_value       =      movement[0][1]
destination_place                     =
self.places[movement[1][0]]
destination_value  =  movement[1][1]  if
source_place.attributes[source_value]<(
value*interval)                     and
self.zerolowerbound == True:
   value =
source_place.attributes[source_value]
source_place.attributes[source_value] =
\
source_place.attributes[source_value]-
(value*interval)
destination_place.attributes[destinatio
n_value] = \
destination_place.attributes[destinatio
n_value] + \ (value*interval)  def
```

```
_test_condition(self, place, token,
operator, value):
        '''!
```

Private method used by rule processors for logical check of condition. For example, the condition

```
'mixer.flour == 0' will be written as
```

```
>>> _test_condition('mixer', 'flour',
'==', 0)
```

place string: name of place/container token string: name of token operator string: binary operator. Allowable values are '==' (equals to), '>' (more than), '>=' (more than or equals to), '<' (less than), '<=' (less than or equals to), and '!=' (not equals to).
value: value to be checked

@return 'passed' if test result is true, or 0 if test result is false

```
        ''' value = float(value)
if operator == '==' and \
self.places[place].attributes[to
ken] == value:    return
'passed' elif operator == '>' and
\
self.places[place].attributes[tok
en] > value: return 'passed' elif
operator == '>=' and \
self.places[place].attributes[tok
en] >= value:
     return 'passed' elif operator
== '<' and \
self.places[place].attributes[tok
en] < value:
     return 'passed' elif operator
== '<=' and \
self.places[place].attributes[toke
n] <= value:
     return 'passed' elif operator
== '!=' and \
self.places[place].attributes[toke
n] != value:
return 'passed' else:
return 'failed' def
_conditions_processor(self
, conditions):
```

conditions list: one or more logical conditions in the format of '<place>.<token> <binary operator> <criterion>', such as 'oven.heat > 300', for evaluation

```
test = [0] * len(conditions)  for i in
range(len(conditions)): cond = conditions[i]      if
len(cond.split('==')) == 2: operator = '=='  cond =
[c.strip() for c in cond.split('==')]      elif
len(cond.split('>')) == 2: operator = '>' cond =
[c.strip() for c in cond.split('>')]      elif
len(cond.split('>=')) == 2:          operator = '>='
cond = [c.strip() for c in cond.split('>=')]       elif
len(cond.split('<=')) == 2:            operator
= '<='             cond = [c.strip() for c in
cond.split('<=')]       elif len(cond.split('!=')) ==
2:             operator = '!='             cond =
[c.strip() for c in cond.split('!=')]
source_place = cond[0].split('.')[0]
source_value = cond[0].split('.')[1]       criterion =
cond[1]        test[i] =
self._test_condition(source_place,   source_value,
operator, criterion) return test          def
_incubate_rule(self, rule, interval):
```

rule: a dictionary representing the incubate
rule interval integer: simulation time interval
@return modified rule dictionary

```
        value = rule['value']         timer =
rule['timer']         conditions = rule['conditions']
movement = rule['movement']
        test = self._conditions_processor(conditions)
if len(['failed' for t in test if t == 'failed']) ==
0:             if (timer + interval) < value:
                rule['timer'] = timer + interval
else:
                source_place                           =
self.places[movement[0][0]]
                source_value = movement[0][1]
                                                       =
                destination_place
self.places[movement[1][0]]
                destination_value = movement[1][1]
destination_place.attributes[destination_value] = \


destination_place.attributes[destination_value] + \


source_place.attributes[source_value]

source_place.attributes[source_value]

=
0                  rule['timer'] = 0          return
rule
         def   _ratio_rule(self,   movement,
ratio,                          limit_check,
limit_set, interval):
```
movement string: defines the movement of a token type.

Each movement is defined in the following format:

```
<source place>.<source  token> ->
<destination place>.<destination token>
```
ratio float: the ratio of tokens to move
limit_check string: logical check for
remainder value limit_set flaot: value to
set token if token in   limit_check is
true interval integer: simulation time
interval
```
source_place =
self.places[movement[0][0]]
source_value = movement[0][1]
destination_place =
self.places[movement[1][0]]
destination_value = movement[1][1]
```
Step 1: Perform ratio rule operation

```
       token_value                                    =
source_place.attributes[source_value] *
\                        ratio * interval
source_place.attributes[source_value] =
\

source_place.attributes[source_value]                 -
token_value
```

```
destination_place.attributes[destinatio
n_value]

= \
destination_place.attributes[destinatio
n_value]     + token_value
```

# Step 2: Perform remaining checks and corrections

```
if len(limit_check.split('>')) == 2:
operator = '>'              limit_check
= [c.strip() for c in
limit_check.split('>')] if
len(limit_check.split('<')) == 2:
            operator = '<'
            limit_check = [c.strip() for
c in limit_check.split('<')]
check_place =
limit_check[0].split('.')[0]
check_token =
limit_check[0].split('.')[1]
check_value = float(limit_check[1])
if
 self._test_condition(check_place,
check_token,              operator,
check_value) == 'passed':            if
source_value in self.losses:
self.losses[source_value] =
self.losses[source_value] +

\  source_place.attributes[source_value]
- limit_set else:
self.losses[source_value] =
\
source_place.attributes[source_value] -
limit_set
source_place.attributes[source_value] =
limit_set          def
_function_rule(self, movement,
function, conditions):         '''!
Private method which simulates a
function rule action.
```

movement string: defines the movement of a token type.
Each movement is defined in the following format:

```
<source place>.<source token> ->
<destination place>.<destination token>
```

function: a Python function to be executed when conditions are met. This function describes the transition of token.
conditions list: one or more logical conditions in the format of

```
'<place>.<token> <binary operator>
<criterion>',          such as
'oven.heat > 300',
```

for evaluation

```
        '''
source_place =
self.places[movement[0][0]]
source_value = movement[0][1]
        destination_place    =
self.places[movement[1][0]]
destination_value = movement[1][1]
test                               =
self._conditions_processor(conditions)
if len(['failed' for t in test if t ==
'failed'])   ==   0:
token_value = function(self.places)
source_place.attributes[source_value] =
\
source_place.attributes[source_value]  -
token_value
destination_place.attributes[destinatio
n_value] = \
destination_place.attributes[destinatio
n_value]     + token_value
def _execute_rules(self, clock,
interval):
'''!
```

Method used by `PNet.simulate()` and

`PNet.simulate_yield()` to execute all the rules.

clock cloat: wall time of the current simulation interval integer: simulation time interval

```
        affected_places
= []  for rName in
self.rules.keys():
```

# Step rule

```
if self.rules[rName]['type'] == 'step': movement  =
self.rules[rName]['movement']                  value
=
self.rules[rName]['value']
self._step_rule(movement,  value,
interval)
```

# Delay rule

```
            if
```

```
self.rules[rName]['type'] == 'delay' and
\
                (clock %
self.rules[rName]['delay']) == 0:
                movement                                      =

self.rules[rName]['movement']

value =

self.rules[rName]['value']
self._step_rule(movement,  value,
interval)
```

# **Incubate rule**

```
            if self.rules[rName]['type']
== 'incubate':

                value =
self.rules[rName]['value']
                rule                                          =

self._incubate_rule(self.rules[rName],
interval) self.rules[rName] = rule
```

# **Ratio rule**

```
            if self.rules[rName]['type']
== 'ratio':

                movement                                      =

self.rules[rName]['movement']
ratio = self.rules[rName]['ratio']

                limit_check                                   =

self.rules[rName]['limit_check']

                limit_set                                     =
self.rules[rName]['limit_set']


self._ratio_rule(movement,  ratio,
limit_check,


limit_set, interval)
```

# Function rule

```
                                                    if
             self.rules[rName]['type'] ==
    'function':                      movement
                                                   =
self.rules[rName]['movemen
t'] conditions =
self.rules[rName]['conditi
ons']
                   function                             =

self.rules[rName]['function']


self._function_rule(movement,  function,
conditions)          def simulate(self,
end_time,                    interval=1.0,
report_frequency=1.0):
'''!
```

Method to simulate the Petri Net. This method stores the generated report in memory; hence, not suitable for extended simulations as it can run out of memory. It is possible to conserve memory by reducing the reporting frequency. Use `simulate_yield` method for extended simulations.

`end_time` integer: number of time steps to simulate. If `end_time = 1000`, it can be 1000 seconds or 1000 days, depending on the significance of each step interval float: number of intervals between each time step. Default = 1.0, simulate by time step interval `report_frequency` float: number of time steps between each reporting. Default = 1.0, each time step is reported

```
        '''
clock = 1
end_time =
int(end_time)
while clock < (end_time
+ 1):
self._execute_rules(clock,
interval)             if (clock %
report_frequency) == 0:
self._generate_report(clock)
clock = clock + interval
def simulate_yield(self,
end_time, interval=1.0):
        '''!
```

Method to simulate the Petri Net. This method runs as a generator, making it suitable for extended simulation.

`end_time` integer: number of time steps to simulate. If `end_time = 1000`, it can be 1000 seconds or 1000 days, depending on the significance of each step interval float: number of intervals between each time step. Default = 1.0, simulate by time step interval

```
        '''
clock = 1
end_time = int(end_time
)         while
clock
< end_time:
self._execute_rules(clock,
interval)
self._generate_report(clock)
rept = {}              for k in
self.report[str(clock)].keys():
rept[k] =
self.report[str(clock)][k]
del self.report[str(clock)][k]
yield (clock, rept)            clock
= clock + interval
            def
_generate_report(se
lf, clock):
```

Method to generate and store report in memory of each token status (the value of each token) in every place/container. clock float: step count of the current simulation

```
        '''   rept =
{}         for pName
in self.places.keys():
           for  aName  in
self.places[pName].attributes.keys():
                value                                    =
self.places[pName].attributes
[aName]                  name =
'.'.join([pName, aName])
rept[name] = value
self.report[str(clock)] = rept
def report_tokens(self,
reportdict=None):
        '''!
```

Method to report the status of each token(s) from each place as a list. This can be used in 2 different ways: to generate a list representation of a status from , one time step (such as from `simulate_yield` method), or to generate a list representation of a status from entire simulation (such as from simulate method). from simulate method

```
net.simulate(65, 1, 1)
status = net.report_tokens() from
simulate_yield method status = [d
for d in net.simulate_yield(65,
1)]         status = [(d[0],
net.report_tokens(d[1])) for d in
status]
```

reportdict dictionary: status from one time step.

Default = None. If None, it will assume that simulate method had been executed and all status are stored in memory, and this method will generate a report from status stored in memory @return tuple of `([<place.token name>], [([<place.token value>]])` if reportdict is given, or tuple of `(time step, [<place.token name>],`

```
[([<place.token value>]]) if reportdict is
None.           if reportdict:

            placetokens =
reportdict.keys()
tokenvalues =
[reportdict[k] for k in placetokens]
return (placetokens, tokenvalues)
else:                timelist =
list(self.report.keys())
datalist = [0] * len(timelist)
for i in range(len(timelist)):
                placetokens   =
list(self.report[timelist[i]].keys())
                                        =
                tokenvalues
[self.report[timelist[i]][k]
for k in placetokens]

                datalist[i]    =        (timelist[i],
placetokens,
tokenvalues)
return datalist
```

CHAPTER 3

MODELING USING PNET

3.1 Baking a Cake

In this example, we modeled the recipe of baking a cake (look at Appendix A for execution) and simulation was done for 120 time steps. We note that the recipe except the use of infinite tokens from Ouroboros, it utilizes all features of PNet. In the recipe used we call for 1000 g of flour, 6 eggs, 500 g of water and 200 g of butter in the following steps:

1. First, we issue the command to activate the mixer. Then, we carefully incorporate 500 g of flour, 250 g of sugar, 3 eggs, and 100 g of butter during each time step.
2. In each time step, the mixer will now turn the mixer batter to 800g of cake pan batter. 3. After thoroughly combining the ingredients, allow the dough to rise in the mixer for 60 time steps.
4. Next, we'll transfer the dough into the pan and allow it to rise for an additional 60 time steps.
5. Transfer the cake to the table for cooling.
6. Enjoy your cake.

The two Steps, we need to display the process of adding and mixing the ingredients into a cake batter. The rate at which dough forms is slower compared to the rate at which ingredients are added. For instance, during each time step, the mixer receives 500 g of flour, 100 g of butter, 3 eggs, and 250 g of sugar and are converted to a batter. After, completion of above steps, the cake is then transferred to a table and cooled down(Step 5).

We described a baking process using a Petri net structure and firing rules. Now, let's explore its working and application in detail:

Application using Petri Net Structure:

1. **Places**: These represent different states or locations in the baking process:

- flour, sugar, eggs, butter: Places represents the ingredients needed for baking.
- mixer: Represents the mixing process where ingredients are combined to form a cake batter.
- cake_pan: Represents the container where the batter is poured.
- oven: Represents the oven where the cake is baked.

2. **Transitions**: Transitions represent the actions or steps used in the baking process:

- mix: Represents the mixing of ingredients in the mixer.
- pour_into_pan: Represents pouring the batter from the mixer into the cake pan.
- bake: Represents the baking process in the oven.

Working using Firing Rules:

1. **Enabling Transitions**: Before a transition can occur, it is important to be enabled. A transition gets enabled if all of its input places have enough tokens (ingredients) to assure their input arcs. In this simulation, each transition checks if there are enough ingredients available in the consequent places.

2. **Firing Transitions**: When a transition is triggered, it takes tokens from its input places and generates tokens in its output places according to the specified rules. For example:
    - The mix transition consumes ingredients from the flour, sugar, eggs, and butter places, mixes them, and produces batter in the mixer place.
    - The pour_into_pan transition consumes batter from the mixer place and pours it into the cake pan.
    - The bake transition consumes batter from the cake pan, bakes it in the oven for a specified time, and produces a cake.

3. **Simulation**:

- The simulation runs for a specified number of time steps.
- At each time step, the transitions are evaluated for firing based on the availability of ingredients and the specified rules.
- If a transition is enabled, it fires, updating the token counts in the places accordingly.
- The process continues for the specified number of time steps, simulating the entire baking process from mixing to baking.

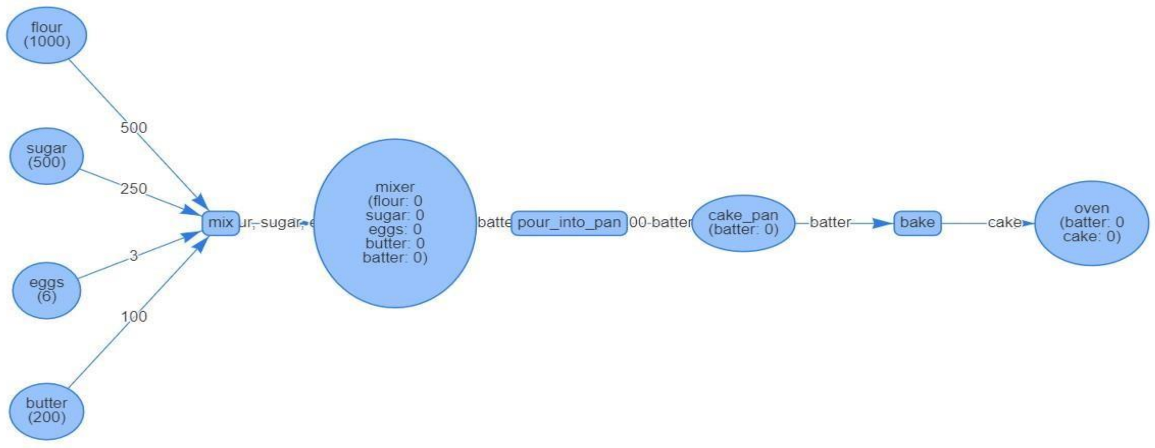
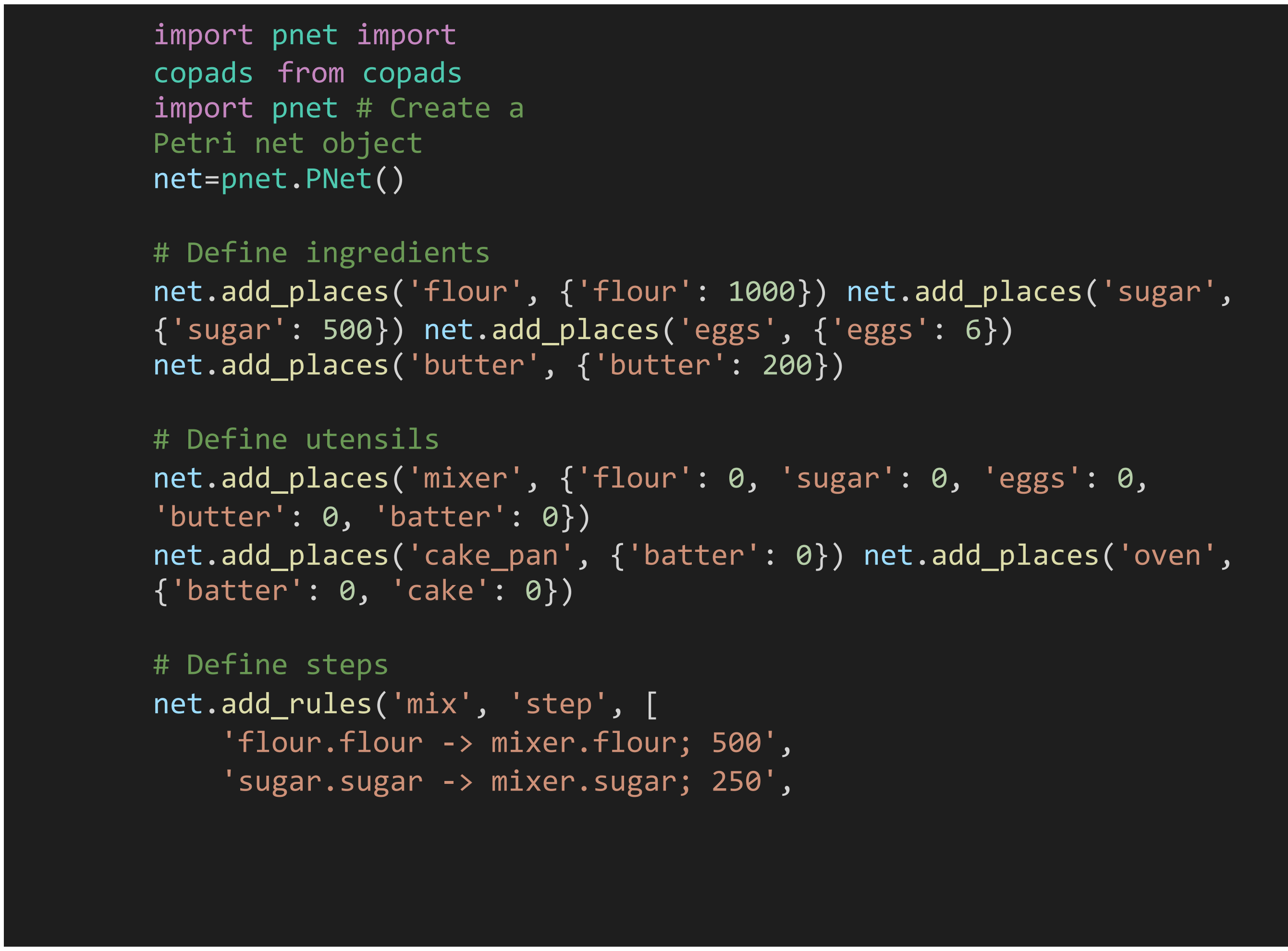


Fig 1: Petri Net Diagram of Cake Baking Model

3.1.1 Appendix A: Code for Baking a Cake



```
import pnet import
copads  from copads
import pnet # Create a
Petri net object
net=pnet.PNet()

# Define ingredients
net.add_places('flour', {'flour': 1000}) net.add_places('sugar',
{'sugar': 500}) net.add_places('eggs', {'eggs': 6})
net.add_places('butter', {'butter': 200})

# Define utensils
net.add_places('mixer', {'flour': 0, 'sugar': 0, 'eggs': 0,
'butter': 0, 'batter': 0})
net.add_places('cake_pan', {'batter': 0}) net.add_places('oven',
{'batter': 0, 'cake': 0})

# Define steps
net.add_rules('mix', 'step', [
    'flour.flour -> mixer.flour; 500',
    'sugar.sugar -> mixer.sugar; 250',
    'eggs.eggs -> mixer.eggs; 3',
    'butter.butter -> mixer.butter; 100'
])  net.add_rules('pour_into_pan', 'step', [
'mixer.batter -> cake_pan.batter; 800'
])
net.add_rules('bake', 'incubate', [
    '60; cake_pan.batter -> oven.batter; cake_pan.batter > 0' ])

# Simulate the baking process net.simulate(120, 1, 1)

# Generate results file data = net.report_tokens()
headers = ['timestep'] + data[0][1]

f = open('cake.csv', 'w')
f.write(','.join(headers) + '\n') for
tdata in data:   tdata = [tdata[0]] +
\  [str(x) for x in tdata[2]]
 f.write(','.join(tdata) + '\n')
f.close()
```

1. Introduction
   The above Python code implements a simulation of a baking process using Petri Nets, a mathematical modeling tool used in various fields including computer science and systems biology. The objective is to model the steps involved in baking a cake, including mixing ingredients, pouring the batter into a cake pan, and baking it in an oven.
2. Petri Net Setup

- Importing Modules: The code imports necessary modules including `pnet` for Petri Net simulation and `copads` for additional computational processes.
- Creating Petri Net Object: An instance of a Petri Net is created using the `PNet` class.

3. Model Definition

- Ingredients and Utensils: Places (nodes) representing ingredients like flour, sugar, eggs, and butter are defined along with their initial quantities. Additionally, places representing utensils such as mixer, cake pan, and oven are defined.
- Steps: Various steps involved in the baking process are defined using rules. These steps include mixing ingredients in the mixer, pouring the batter into a cake pan, and baking the cake in an oven. Each step specifies the transition of tokens (quantities) between places.

4. Simulation

- Simulating the Baking Process: The baking process is simulated using the defined Petri Net model. The simulation runs for 120 time units with 1 repetition and 1 time step.

5. Results Generation

- Generating Results File: The results of the simulation are extracted and formatted into a CSV file named "cake.csv". The file contains information about the quantities of ingredients and the state of utensils at each time step during the baking process.

6. Conclusion

In conclusion, this code demonstrates the use of Petri Nets for modeling and simulating a real-world process—in this case, the baking of a cake. By defining ingredients, utensils, and steps, and simulating the process, it provides insights into the dynamics of the baking process and facilitates analysis and optimization.

3.2 <u>Model on Epidemiological Disease</u>

Epidemiological disease models are conceptual frameworks that analyze ecological and epidemiological events, specifically focusing on the interactions between a host and a disease. Epidemiological models have demonstrated their utility in studying the evolutionary dynamics and forecasting characteristics of pathogen dissemination, such as duration and prevalence. Alphabet models are conceptual frameworks that represent a population, where individuals who are vulnerable are assumed to be infected by a contagious pathogen. The population is categorized into three epidemiological subclasses: S represents persons who are vulnerable to disease we use the variable "I" to represent the number of individuals who are infected, and "R" to represent the number of individuals who no longer contribute to the spread of diseases at a given moment. The Susceptible-Infectious-Susceptible (SIS) model is based on the premise that the pathogen infects individuals who are susceptible, leading to an infection. After recovering from the infection, these individuals return to the susceptible category once again. Hosts that are infected experience a consistent recovery rate per person, denoted as γ, whereas β represents the rate at which the susceptible class becomes infected. The SIS model is designed to simulate the spread of rapidly changing viruses and diseases that do not confer immunity. The Susceptible-Infectious-Recovered (SIR) model is analogous to the SIS model, with the distinction of the pathogen that results in permanent immunity. Those are immune to reinfection who have been infected and then recovered, exhibiting lasting immunity. This paradigm is engaged for viral illnesses including measles, mumps, and rubella. The SIRS model is similar to the SIR model, with the distinction that the acquired protection is transient. The ones who are not having immunity to reinfection are those who have been infected and then healed. TB is an example of an infection that can be modeled using the SIRS framework. Nevertheless, a set of ordinary differential equations (ODEs) can be used to execute the majority of epidemiological models and Petri Nets are widely used mathematical constructions in the field of mathematical modeling. Therefore, there is a requirement for a technique to express an Ordinary Differential Equation (ODE) using Petri Net notation. Soliman and Heiner have established a connection between ODEs and state-transition networks. In essence, an ordinary differential equation (ODE) represents the relationship between time progress and the evolution of a system's state, but a Petri Net captures the transitions that lead to changes in the system's condition w.r.t. time (Fig. 2). Within the framework of states (nodes) and transitions (arcs), this implies that ODEs symbolize the nodes, while Petri Nets symbolize the transitions. Consequently, it becomes straightforward to convert ODE representations to Petri Net representations, provided that the time unit remains consistent in both representations.

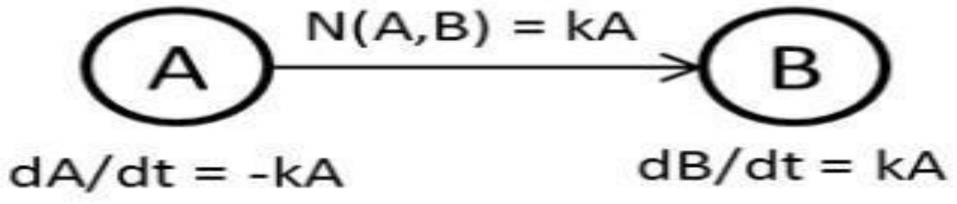


Fig 2: Petri Net Transition Rule and Ordinary Differential Equation Correspondence.

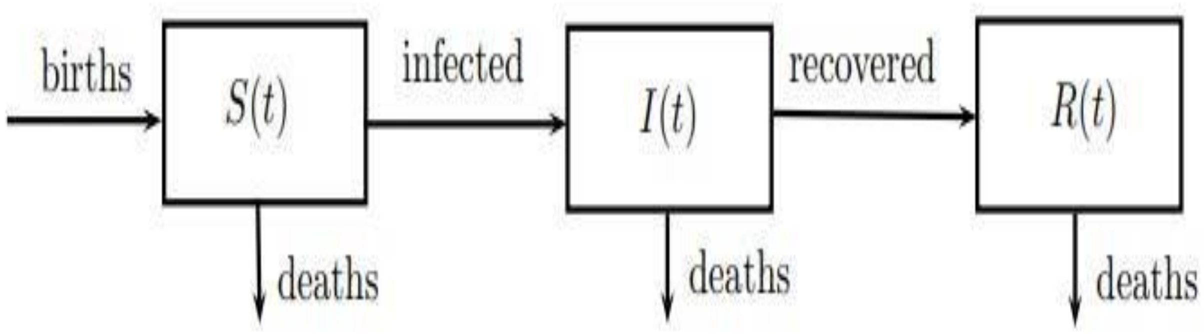


Fig 3: The input-output diagram represents the influenza epidemic model in a school setting without reinfection.

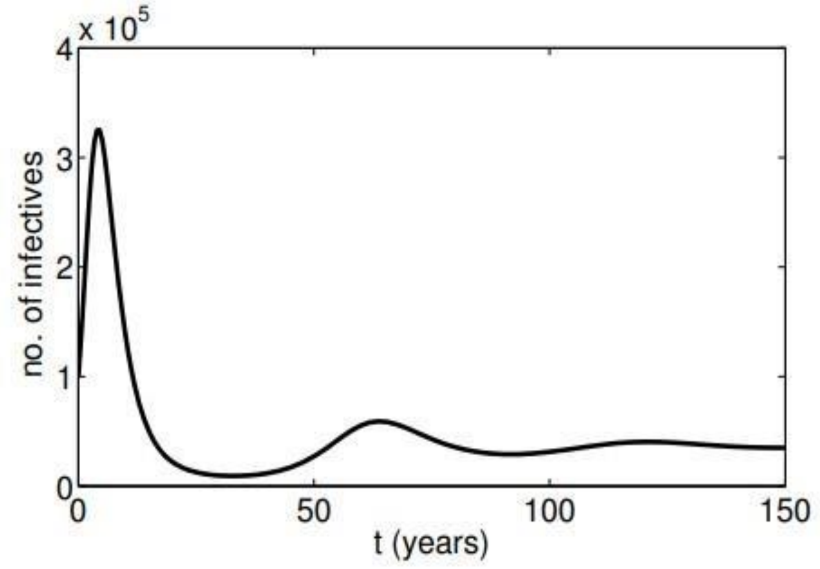


Fig 4: Using numerical methods, we can solve the differential equations that describe the spread of an infectious disease in a hypothetical population of N = 106. The parameter values utilized were β = 10−6 susceptibles−1 day−1, γ = 1/3 years−1, b = a = 1/50 years−1, with initial populations S(0) = 9 × 105 and I(0) = 105.

Our simulation results are indicating that the ratio of the population that is infected to the population that is susceptible stabilizes over time. Since, there is no immunity acquired after recovering from the infection, it is anticipated that there would be a consistent population of infected individuals, which is also referred to as an endemic population. This assumes that there are no births or deaths for the whole term, and that the sickness is not fatal. When individuals develop immunity after recovering from a disease, the SIS model transforms into the SIR model. In the SIR model, the population gradually becomes completely immune over time, assuming no new individuals are born. This is analogous to the situation with chickenpox, where most individuals who have recovered from the disease get lifetime immunity. As a result, children are more susceptible to chickenpox, while most adults are immune. Nevertheless, the possibility of reinfection arises if the acquired immunity is of a limited duration, resulting in a evolution from the SIR model to the SIRS model. We come to know by our findings that in situations when there is a consistent population of infected individuals (known as endemic) the SIRS model exhibits similar behavior to the SIS model. However, there is also a consistent group of individuals with immunity who have recently recovered from the condition. This occurrence is anticipated when the infectious agent has the ability to re-infect an individual who has previously recovered from the infection.

We are describing the dynamics of an epidemic using a Petri net structure and firing rules. Let's delve into its application and functioning:

Application using Petri Net Structure:

1. **Places**: These represent the different states of individuals within the population:
   - Susceptible: People who are vulnerable to the infection.
   - Infected: People who are presently contaminated with the illness.
   - Recovered: Individuals who have recovered from the disease and gained immunity.

2. **Transitions**: Transitions represent the events or actions that occur in the epidemic:
   - Infection: Represents the transmission of the disease from susceptible individuals to infected individuals.
   - Recovery: Represents the recovery of infected individuals.
   - Resusceptible: Represents the loss of immunity in recovered individuals, making them susceptible again.

Working using Firing Rules:

1. **Enabling Transitions**: Before a transition can occur, it must be enabled. A transition is considered enabled if there are a sufficient number of tokens available in the corresponding input places. In this simulation, a transition is enabled if there are sufficient susceptible individuals for infection, infected individuals for recovery, or recovered individuals for resusceptibility.

2. **Firing Transitions**: The act of extracting tokens from input places and generating tokens in output places according to the specified rules happens when a transition is said to be fired. The infection transition leads to consumption of susceptible individuals and leads to production of infected individuals in this simulation, the recovery transition leads to consumption of infected individuals and production of recovered individuals, and the resusceptible transition consumes recovered individuals and produces susceptible individuals.

3. **Simulation**:

- The simulation operates for a specified number of time steps.
- At each time step based on the conditions specified by the firing rules, the transitions are evaluated for firing.
- The enabling of a transition leads to it’s firing, updating the token counts in the places accordingly.
- For the specified number of time steps the process keeps on continuing which then allows the epidemic dynamics to unfold over time.

3.2.1 Appendix B: Code for SIRS Model

```
from copads import pnet
 # Parameters  infection_rate
 = 0.01  recovery_rate =
 0.005  resusceptible_rate =
 0.01

 # Initialize Petri
 net net = pnet.PNet()
 net.add_places('susceptible', {'susceptible': 100})
 net.add_places('infected', {'infected': 0})
 net.add_places('recovered', {'recovered': 0})

 # Define transition functions def
 susceptible_to_infected(places
 ):     susceptible =
 places['susceptible'].attributes['susceptible']     return
 infection_rate * susceptible
  def infected_to_recovered(places):     infected =
 places['infected'].attributes['infected']     return
 recovery_rate * infected
  def recovered_to_susceptible(places):     recovered =
 places['recovered'].attributes['recovered']     return
 resusceptible_rate * recovered

 # Add rules  net.add_rules('infection',
 'function', [
     'susceptible.susceptible ->
 infected.infected',     susceptible_to_infected,
 h'susceptible.susceptible > 0'
 ]
 )
   net.add_rules('recovery',
 'function', [
     'infected.infected -> recovered.recovered',
 infected_to_recovered,
 'infected.infected > 0'
 ]
 )

 net.add_rules('resusceptible', 'function', [
     'recovered.recovered -> susceptible.susceptible',
 recovered_to_susceptible,      'recovered.recovered
 > 0'
 ])

 # Simulate the Petri net net.simulate(500,
 1, 1)

 # Report tokens and save to CSV
 data = net.report_tokens()  headers
 = ['timestep'] + data[0][1]   with
 open('sirs.csv', 'w') as f:
     f.write(','.join(headers) + '\n')      for
 timestep_data in data:         row = [timestep_data[0]]
 + [str(x) for x in timestep_data[2]]
         f.write(','.join(row) + '\n')
```

3.3 Evolution of Molecule Count in a System

To simulate the evolution of molecule count in the system using a Petri net using python, we can write a custom implementation. Here's how we can do this:

1. Describe the places, transitions, and arcs for the Petri net.
2. Define functions to check transition enablement and firing.

3. Simulate the system over a number of steps, applying the transitions according to their probabilities.

We are unfolding a simple chemical reaction process using a Petri Net structure and it’s firing rules. Let's break down its application and working using Petri nets:

Application using Petri Net Structure:

1. **Places**: In the system, places symbolize different states or locations. In this simulation, there are three places:

- Molecules: Represents the pool of molecules in the system.
- Synthesis: Represents the process of synthesizing molecules.
- Dissociation: Represents the process of dissociating molecules.

2. **Transitions**: In the system, events or actions that can occur are represented by Transitions. In this simulation, there are two transitions:

- synthesize: Represents the synthesis process (combining different things).
- dissociate: Represents the dissociation process(breakdown into smaller components).

3. **Arcs**: The flow of tokens (molecules) between places and transitions is represented by Arcs. There are input arcs and output arcs:

- Places to transitions are connected by Input Arcs, indicating the tokens required for the transition to fire.
- Transitions to places are connected by Output Arcs, indicating where the tokens produced by the transition will go.

Working using Firing Rules:

1. **Enabling Transitions**: Enabling is necessary to make a transition fire. A transition is considered enabled if all of its input spots have enough tokens to satisfy their input arcs. The `is_enabled` method in the `Transition` class checks if a transition is enabled.

2. **Firing Transitions**: When a transition fires then the consumption of tokens from its input places according to the input arcs takes place which later produces tokens in its output places according to the output arcs. The `fire` method in the `Transition` class handles this process.

3. **Simulation**: o The simulation executes for a predetermined amount of iterations.
   - At each step, there's a probability of synthesis and dissociation taking place.
   - If the conditions for synthesis or dissociation are met (based on random probabilities and transition enabling), the corresponding transition fires.
   - The number of molecules in the 'Molecules' place is recorded at each step to track the evolution of the system.

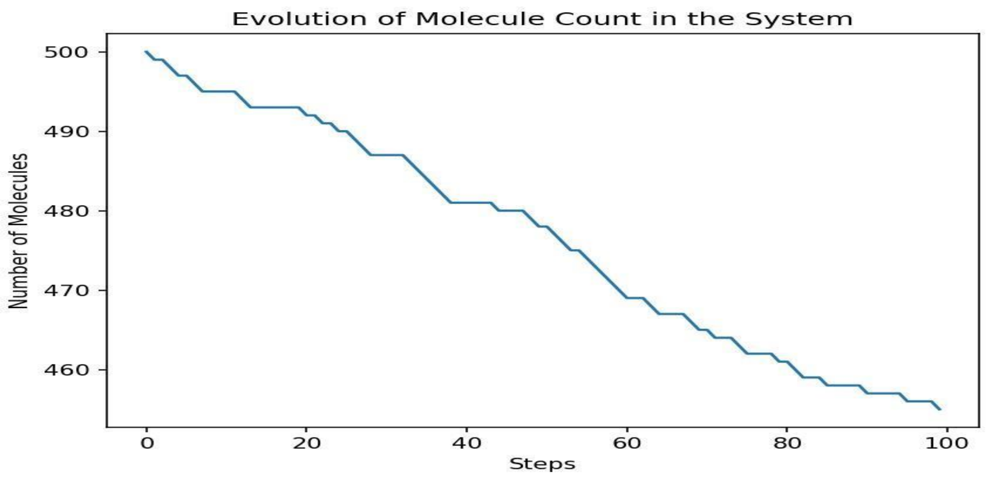
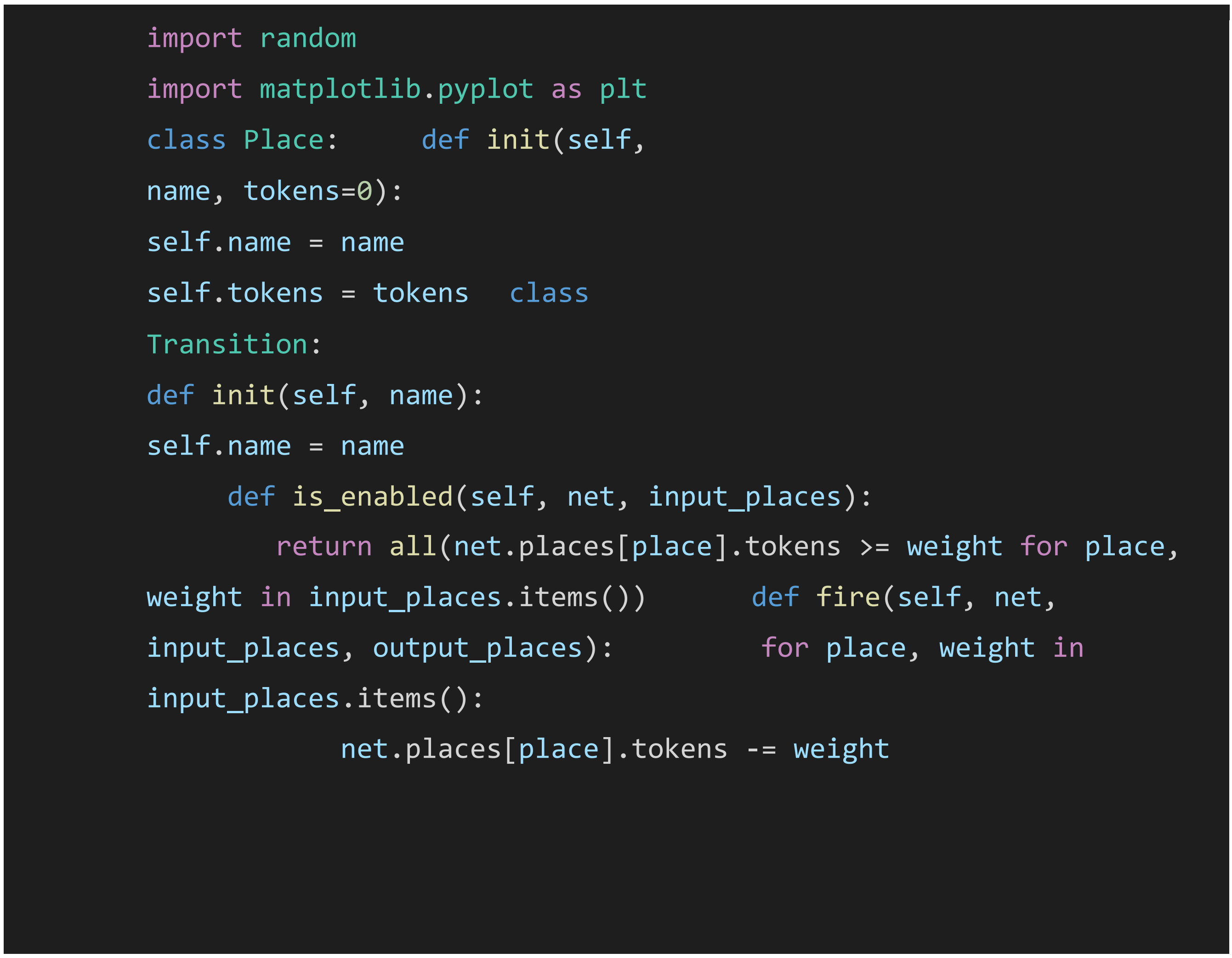


Fig 5: Evolution of molecule count in the system

3.3.1 Appendix C: Code for evolution of molecule count in a system



```
import random
import matplotlib.pyplot as plt
class Place:     def init(self,
name, tokens=0):
self.name = name
self.tokens = tokens   class
Transition:
def init(self, name):
self.name = name
     def is_enabled(self, net, input_places):
        return all(net.places[place].tokens >= weight for place,
weight in input_places.items())      def fire(self, net,
input_places, output_places):         for place, weight in
input_places.items():
            net.places[place].tokens -= weight
```

```
for place, weight in output_places.items():
net.places[place].tokens += weight
```

```
class PetriNet:     def
init(self, name):
self.name = name
self.places = {}
self.transitions = {}
self.input_arcs = {}
self.output_arcs = {}      def
add_place(self, place):
self.places[place.name] = place      def
add_transition(self, transition):
self.transitions[transition.name] = transition
self.input_arcs[transition.name] = {}
self.output_arcs[transition.name] = {}      def
add_input(self, place, transition, weight=1):
self.input_arcs[transition][place] = weight      def
add_output(self, place, transition, weight=1):
self.output_arcs[transition][place] = weight

# Define the Petri net
net = PetriNet('MoleculeSynthesisDissociation')

# Add places
net.add_place(Place('Molecules', tokens=500))  # Starting with
500 molecules
net.add_place(Place('Synthesis', tokens=0))
net.add_place(Place('Dissociation', tokens=0))

# Add transitions
net.add_transition(Transition('synthesize'))
net.add_transition(Transition('dissociate'))
```

```python
# Add arcs
net.add_input('Synthesis', 'synthesize', weight=1)
net.add_output('Molecules', 'synthesize', weight=1)
net.add_input('Molecules', 'dissociate', weight=1)
net.add_output('Dissociation', 'dissociate', weight=1)
```

```
# Define the probabilities for synthesis and dissociation
synthesis_probability = 0.6  # Probability of synthesis
dissociation_probability = 0.4  # Probability of dissociation
# Simulation parameters
steps = 100  # Number of simulation steps
molecule_counts = []  # To store the number of molecules at each
step

# Run the simulation for step in range(steps):     if
random.random() < synthesis_probability:
        if net.transitions['synthesize'].is_enabled(net,
net.input_arcs['synthesize']):
net.transitions['synthesize'].fire(net,
net.input_arcs['synthesize'], net.output_arcs['synthesize'])
if random.random() < dissociation_probability:        if
net.transitions['dissociate'].is_enabled(net,
net.input_arcs['dissociate']):
net.transitions['dissociate'].fire(net,
net.input_arcs['dissociate'], net.output_arcs['dissociate'])
molecule_counts.append(net.places['Molecules'].tokens)
# Plot the results plt.plot(molecule_counts)
plt.xlabel('Steps')
plt.ylabel('Number of Molecules')
plt.title('Evolution of Molecule Count in the System') plt.show()
```

CHAPTER-4

CONCLUSION

Winding up, we have given a definition of all the necessary and required rules used for developing a model using PNET library in python. We also come to a conclusion that it's easier to build up a Petri net complex model using PNET library in python and lesser time is consumed to come to conclusions than understanding a complex model and concluding from that model.

We did an exploration of Petri Net Theory (PNET) through it's practical applications such as baking a cake project, a model of epidemiological illness, and a model of the evolution of molecule count in a system which demonstrates the usefulness and power of PNET for modeling complex systems. At this juncture, we summarize the findings and insights gained from these projects.

1. Baking a Cake Project

Objective: To model the process of baking a cake in order to understand the sequence of steps and resource dependencies.
Findings:

- Sequential and Parallel Processes: Effectively both sequential and parallel processes involved in cake baking are captured by the Petri Net model, such as mixing ingredients (sequential) and preheating the oven while preparing the batter (parallel).
- Resource Management: Resource constraints, such as limited mixing bowls or baking pans, are highlighted in the model and ensures these resources are optimally utilized.

- Process Optimization: The simulation of the Petri Net leads to the identification of bottlenecks in the process, which allows for optimization of the workflow, such as reduction in idle time for resources or streamlining steps to save time.

2. Model of Epidemiological Disease

Objective: Simulation of the spread of an infectious disease for understandong the dynamics of infection and recovery in a population.
Findings:

- Compartmental Modeling: For compartmental models in epidemiology Petri Nets are well-suited, such as the SIR (Susceptible-Infectious-Recovered) model. The transitions between compartments are naturally represented by transitions in the Petri Net.
- Stochastic Simulations: The stochastic nature of Petri Nets helps in allowing the simulation of random events, such as the infection rate and recovery rate, providing a realistic representation of spread of disease.
- Intervention Strategies: The model can be used to test various intervention strategies (e.g., vaccination, quarantine) by adding/removing places and transitions, thus helping in planning effective control measures.

3. Evolution of Molecule Count in a System

Objective: To model the dynamics of chemical reactions and the evolution of molecule counts over time.
Findings:

- Reaction Dynamics: Petri Nets efficiently model chemical reaction networks, capturing how the reactants transform to products through transitions that represent chemical reactions.

- Conservation Laws: The model enforces laws of conversation, ensuring that the total number of molecules is preserved, reflecting real-world chemical processes.
- Complex Systems: For systems with more than one reactions and intermediate compounds, Petri Nets provide a clear and manageable way to analyze and visualize the complex interactions and dependencies.

Overall Conclusion

The use of the PNET library in Python for these diverse projects underscores its heftiness and elasticity in modeling various types of systems. Main conclusions are:

- Versatility: Petri Nets can be applied to a wide range of fields, from simple processes like cake baking to complex systems like epidemiological spread and chemical reaction networks.
- Clarity and Manageability: A clear graphical representation is offered by them, making it easier to understand, manage, and communicate the modeled systems.
- Simulation and Optimization: The ability to simulate these models allows for dynamic analysis and optimization, providing valuable insights and aiding in decision-making.
- Real-World Applications: The insights gained from these models have practical implications, such as improving efficiency in workflows, planning public health interventions, and understanding chemical processes.

Overall, the projects demonstrate that Petri Net Theory, implemented through the PNET library in Python, is a powerful tool for analyzing, modeling, and optimizing complex systems across various domains. Hence, the entire paper demonstrates the power of PNET as a versatile modeling framework in Python for simulating complex systems, enabling researchers, engineers, and enthusiasts to gain valuable insights, make informed decisions, and drive innovation across various domains.

REFERENCES

[1] W. Reisig, “Petri Nets: An Introduction”, Springer– Verlag, 1985.

[2] W. Reisig, “Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies”, Springer–Verlag, 2013.

[3] Koutny, M. (2019). Petri Nets and Petri’s Nets: A Personal Perspective. In: Reisig, W., Rozenberg, G. (eds) Carl Adam Petri: Ideas, Personality, Impact. Springer, Cham.

[4] Bobbio, “System Modelling with Petri Nets”. Instituto Elettrotecnico Nazionale Galileo Ferraris Strada Delle Cacce 91, 10135 Torino, Italy, 1990, 14-15.

[5] Koch I, Reisig W, Schreiber F (eds) (2011) Modeling in systems biology. the Petri net approach. Computational biology, vol 16. Springer, BerlinD. Gilbert and M. Heiner M.

[6] C. Chaouiya, “Petri Net Modelling of Biological Networks”, Briefings in Bioinformatics 8, 2007, pp 210219.

[7] H. Matsuno and A. Doi, “Hybrid Petri Net Representation of Gene Regulatory Network”, Pacific Symposium on Biocomputing 5, 2000, 338-349.

[8] Murata T (1989) Petri nets: properties, analysis and applications. Proc IEEE 77(4):541–580S.

[9] Wingender E (ed) (2011) Biological Petri nets. Studies in health technology and informatics.vol 162. IOS Press, Lansdale

[10] M. Matcovschi, C. Popescu, and O. Pastravanu, A new approach to hybrid system simulation: Development of a simulink library for Petri net models," Journal of Control Engineering and Applied Informatics 7, 2005, 55-62.

[11] F. Pommereau, “SNAKES: A Flexible High-Level Petri Nets Library (Tool Paper)”, Proceedings of the 36th International Conference on Petri Nets (PETRI NETS 2015), 2015, 254-265

[12] F. Brauer, “Compartmental Models in Epidemiology”, In Lecture Notes in Mathematics 1945, 2008, 19-79.

[13] Silva M (2013) Half a century after Carl Adam Petri’s Ph.D. thesis: a perspective on the field. Annu Rev Control 37(2):191–219M.

[14] J. Keeling and K. T.D. Eames, “Networks and epidemic models”, Journal of the Royal Society Interface 2, 2005, 295-307.

[15] C. Ozcaglar, A. Shabbeer, S. L. Vandenberg, B. Yener, K. P. Bennett,“Epidemiological models of Mycobacterium Tuberculosis complex infections”, Mathematical Biosciences 236, 2012, 77 – 96.

[16] P. Munz, I. Hudea, J. Imad and R. J. Smith, “When Zombies Attack!: Mathematical Modelling of Outbreak of Zombie Infection”, Infectious Disease Modelling Research Progress, 4, 2009, 133-150.

[17] M. Ling, “COPADS IV: Fixed Time-Step ODE Solvers for a System of Equations Implemented as a Set of Python Functions”, Advances in Computer Sciences 5, 2016, xxxxx.

[18] J.P. Aparicio and M. Pascual, “Building Epidemiological Models from R(0): An Implicit Treatment of Transmission in Networks”, Proceedings of the Royal Society B: Biological Sciences 274, 2007, 505-512.

[19] L. Kong, J. Wang, W. Han and Z. Cao, “Modeling Heterogeneity in Direct Infectious Disease Transmission in a Compartmental Model”, International Journal of Environmental Research and Public Health 13, 2016, 253.

[20] M. Ling, “Of (Biological) Models and Simulations”, MOJ Proteomics & Bioinformatics 3, 2016, 00093.

[21] S. Soliman and M. Heiner M, “A Unique Transformation from Ordinary Differential Equations to Reaction Networks”, PLoS One 5, 2010, Article e14284.

[22] H.W. Hethcote and P. van den Driessche, “An SIS Epidemic Model with Variable Population Size and a Delay”, Journal of Mathematical Biology 34, 1995, 177194.

[23] “Facts about chickenpox”, Paediatrics & Child Health 10, 2005, 413-414.

[24] Murata, T.: Petri nets: properties, analysis and applications. Proceedings IEEE 77(4), 541–580 (1989)

[25] https://johncarlosbaez.wordpress.com/2020/10/19/epidemiologicalmodeling-with-structured-cospans/

[26] https://johncarlosbaez.wordpress.com/2012/10/01/petrinet-programming/

[27] https://johncarlosbaez.wordpress.com/2012/12/20/petrinet-programmingpart-2/

[28] Mathematical Modelling with Case Studies: A differential equation approach using Maple and MATLAB

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

PLAGIARISM VERIFICATION

Title of the thesis – PNet Module: Empowering Petri Net Modeling and Simulation

Total Pages 58

Name of Scholars – Charu Singh (2K22/MSCMAT/05) and Shefali (2K22/MSCMAT/36)

Supervisor- Dr. Payal

Department -Applied Mathematics

This is to report that the above thesis was scanned for similarity detection. Process and outcome is given below:

Software used: Turnitin

Similarity Index: 8%

Total Word Count: 10263

Date: 3 June, 2024

Similarity Report

PAPER NAME	AUTHOR
PNet_Thesis.docx	**Msc**
WORD COUNT	CHARACTER COUNT
10263 Words	**60764 Characters**
PAGE COUNT	FILE SIZE
57 Pages	**263.7KB**
SUBMISSION DATE	REPORT DATE
Jun 3, 2024 7:34 PM GMT+5:30	**Jun 3, 2024 7:35 PM GMT+5:30**

● 8% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

- 8% Internet database
- 1% Publications database
- Crossref database
- Crossref Posted Content database
- 2% Submitted Works database

● Excluded from Similarity Report

- Bibliographic material
- Quoted material
- Small Matches (Less then 10 words)

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

CERTIFICATE OF FINAL THESIS SUBMISSION

1. Name: Charu Singh and Shefali
2. Roll No.: 2K22/MSCMAT/05 and 2K22/MSCMAT/36
3. Thesis title: "PNet Module: Empowering Petri Net Modeling And Simulation".
4. Degree for which the thesis is submitted: M.Sc. Mathematics
5. Faculty of the University to which the thesis is submitted: Dr. Payal
6. Thesis Preparation Guide was referred to for preparing the thesis. YES ☐ NO ☐
7. Specifications regarding thesis format have been closely followed. YES ☐ NO ☐
8. The contents of the thesis have been organized based on the guidelines. YES ☐ NO ☐
9. The thesis has been prepared without resorting to plagiarism. YES ☐ NO ☐
10. All sources used have been cited appropriately. YES ☐ NO ☐
11. The thesis has not been submitted elsewhere for a degree. YES ☐ NO ☐
12. All the correction has been incorporated. YES ☐ NO ☐
13. Submitted 2 hard bound copies plus one CD. YES ☐ NO ☐

(Signature of Candidate(s))
Name(s): Charu Singh and Shefali
Roll No.: 2K22/MSCMAT/05 and 2K22/MSCMAT/36