

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки

(повне найменування інституту, факультету)

кафедра Інформатики та програмної інженерії

(повна назва кафедри)

«До захисту допущено»

_____ Дифучин А. Ю. _____
(підпис) (ініціали, прізвище)

Курсова робота

з дисципліни Моделювання систем

за освітньо-професійною програмою «Програмне забезпечення
інформаційних управляючих систем та технологій»
спеціальності «121 Інженерія програмного забезпечення»

на тему Універсальний алгоритм імітації класичної мережі Петрі
(тема ВС1 зі списку додаткових тем)

Виконав: студент III курсу, групи ІІІ-01 Пашковський Євгеній Сергійович
(прізвище, ім'я, по батькові)

(підп
ис)

Керівник

асистент Дифучин А. Ю.

посада, науковий ступінь, вчене звання, прізвище, і ім'я, по батькові

(підп
ис)

Члени комісії

асистент Дифучин А. Ю.

посада, науковий ступінь, вчене звання, прізвище, і ім'я, по батькові

(підп
ис)

професор, д.т.н., Стеценко І. В.

посада, науковий ступінь, вчене звання, прізвище, і ім'я, по батькові

(підп
ис)

Засвідчую, що у цій курсовій роботі немає
запозичень з праць інших авторів без
відповідних посилань.

Студент

(підпис)

ПОСТАНОВКА ЗАВДАННЯ

Розробити універсальний алгоритм імітації класичної мережі Петрі мовою TypeScript. Перевірити роботу універсального алгоритму імітації класичної мережі Петрі на основі деяких тестових моделей. Навести графічні схеми отриманих моделей у формалізмі мереж Петрі. Провести верифікацію та експерименти над створеними моделями з метою визначення та виправлення недоліків у отриманому програмному забезпеченні (зокрема, щодо неправильної обробки конфліктних переходів) та інших параметрів моделей.

АНОТАЦІЯ

Структура та обсяг роботи. Пояснювальна записка курсової роботи складається з 5 розділів, містить 23 рисунки, 2 таблиці, 1 додаток, 2 джерела.

Мета. Розробити універсальний алгоритм імітації, описати концептуальні та формалізовані моделі, провести експериментування над вказаними моделями.

У розділі розробки концептуальної моделі було окреслено концептуальну модель та визначено необхідні властивості імітаційних алгоритмів.

У розділі аналіз розробки формалізованої моделі було описано моделі у рамках формалізму мереж Петрі.

У розділі алгоритмізації моделі та її реалізації було описано усі деталі щодо алгоритму та його реалізації.

У розділі експериментів на моделі було проведено верифікацію моделі та факторний експеримент.

У розділі інтерпретації результатів моделювання та експериментів було описано отримані результати в рамках проведеного моделювання та експериментування.

КЛЮЧОВІ СЛОВА: МОДЕЛЮВАННЯ СИСТЕМ, МЕРЕЖІ ПЕТРІ, УНІВЕРСАЛЬНИЙ АЛГОРИТМ

ЗМІСТ

ВСТУП.....	6
1. РОЗРОБКА КОНЦЕПТУАЛЬНОЇ МОДЕЛІ.....	7
2. РОЗРОБКА ФОРМАЛІЗОВАНОЇ МОДЕЛІ.....	8
3. АЛГОРИТМІЗАЦІЯ МОДЕЛІ ТА ЇЇ РЕАЛІЗАЦІЯ.....	11
3.1. Опис складових алгоритму імітації.....	11
3.1.1. Вибір алгоритму просування модельного часу.....	11
3.1.2. Вибір алгоритму просування стану моделі в залежності від часу....	11
3.1.3. Вибір алгоритму збирання інформації про поведінку моделі у процесі імітації.....	12
3.2. Загальний опис реалізації алгоритму.....	12
3.3. Опис засобів розробки алгоритму.....	13
3.4. Опис тестових моделей.....	13
3.4.1. Тестова модель для визначення правильності роботи алгоритму....	13
3.4.2. Задача про філософів.....	14
3.4.3. Задача про філософів з явищем дедлоку.....	15
3.5. Детальний опис компонентів реалізації алгоритму.....	16
3.6. Верифікація моделі.....	30
4. ЕКСПЕРИМЕНТИ НА МОДЕЛІ.....	33
4.1. Дерево досяжності.....	33
4.2. Опис вхідних та вихідних параметрів для проведення факторного експерименту.....	36
4.3. Факторний експеримент.....	37
4.3.1. Тактичне планування.....	37
4.3.2. Стратегічне планування.....	38
5. ІНТЕРПРЕТАЦІЯ РЕЗУЛЬТАТІВ МОДЕЛЮВАННЯ ТА ЕКСПЕРИМЕНТІВ.....	41
ВИСНОВКИ.....	43
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	44

ДОДАТКИ.....45

Додаток А. Текст програмного коду..... 45

ВСТУП

Імітаційне моделювання, або моделювання на основі імітації, є потужним інструментом в сучасному світі, де велика кількість складних інтеракцій відбувається в різноманітних сферах, від бізнес-процесів і виробництва до транспортних систем та соціальних об'єктів. Актуальність імітаційного моделювання визначається необхідністю аналізу та вдосконалення функціонування складних систем, в яких важко або неможливо провести експерименти в реальному середовищі.

Застосування імітаційного моделювання дозволяє вирішувати різноманітні завдання: від оптимізації бізнес-процесів та планування ресурсів до аналізу транспортних потоків і управління ланцюгами постачання. Цей підхід дозволяє розглядати системи в їхній динаміці, враховуючи взаємодію різних компонентів та випадковість в їхній поведінці.

В умовах стрімкого технологічного розвитку і високого ступеня складності бізнес-процесів і технічних систем, імітаційне моделювання стає невід'ємною частиною прийняття управлінських рішень та розробки оптимальних стратегій. В цьому контексті, важливим стає не лише створення точних моделей, але й вміння адекватно відтворювати динаміку реальних процесів для отримання коректних та достовірних результатів.

Таким чином, імітаційне моделювання визначається як ключовий інструмент для дослідження, вдосконалення та оптимізації різноманітних систем, що підкреслює його актуальність у сучасному науковому та практичному контексті.

Мережі Петрі займають важливе місце у світі імітаційного моделювання та аналізу систем. Їх важливість полягає в тому, що вони надають потужний та гнучкий інструмент для моделювання складних паралельних процесів, що зустрічаються у різних областях, від індустріального виробництва до інформаційних систем.

1. РОЗРОБКА КОНЦЕПТУАЛЬНОЇ МОДЕЛІ

Будь-яка імітаційна система може бути представлена у вигляді мережі Петрі, де обслуговуючі пристрої або одиниці інформації відповідають певним маркерам або місцям у мережі. Загальний принцип функціонування мережі Петрі полягає у токенах, які представляють вимоги для обробки, їхній обробці пристроями та виході вимог із системи.

При потраплянні вимоги на обробку може також виникати черга, якщо вона присутня (тобто, перед обробником існує таке місце, яке не має інших альтернативних дуг для виходу маркерів). У класичних мережах Петрі обробка (активація на переходах) відбувається миттєво, після чого вимога покидає пристрій і надходить до наступної системи за певними правилами.

Щоб створити імітаційну модель класичної мережі Петрі, можна використовувати алгоритм моделювання, який враховує динаміку токенів та переходів у мережі. Цей алгоритм має включати у себе введення нових токенів, їх переміщення по місцях та переходах, а також дуже рідко у специфічних випадках – видалення токенів після завершення обробки. Для кожного етапу імітації слід враховувати стан системи, кількість токенів у кожному місці, а також виконувати переходи відповідно до правил мережі Петрі.

Головною метою є створення точної та ефективної імітаційної моделі будь-якої імітаційної системи за допомогою мережі Петрі, яка дозволить отримати необхідні статистичні характеристики для подальшого аналізу та верифікації створеної моделі.

2. РОЗРОБКА ФОРМАЛІЗОВАНОЇ МОДЕЛІ

Мережа Петрі – це математична модель, яка використовується для опису паралельних процесів та систем. Вона була винайдена Карлом Адамом Петрі в 1962 році та знаходить застосування в області теорії систем, інженерії програмного забезпечення, дизайну систем управління та інших галузях. Мережі Петрі дозволяють моделювати і аналізувати паралельні та розподілені системи, виявляти можливі конфлікти та блокування. Вони знайшли широке застосування в індустрії для проектування та аналізу процесів, де важливо враховувати паралельні дії та взаємодію різних компонентів системи [1].

Основними елементами мережі Петрі є:

- Місце (англ. “Place”). Представляє стан системи. В місцях зберігаються токени, що вказують на наявність ресурсів або зазначають поточний стан системи.
- Перехід (англ. “Transition”). Представляє подію або дію, яка може відбутися в системі.
- Дуга (англ. “Arc”). З'єднує місця та переходи, вказуючи напрямок переміщення токенів.
- Токен (англ. “Token”). Позначає одиницю інформації чи ресурсу. Токени переміщаються вздовж дуг між місцями та переходами під час виконання подій.

Також основні елементи мереж Петрі вказані на рисунку 2.1 [2].

Е Л Е М Е Н Т И М Е Р Е Ж І П Е Т Р І

Перехід		позначає подію
Позиція	○	позначає умову
Дуга	○ → ⇨ ○	позначає зв'язки між подіями та умовами
Маркер(один)	○ •	позначає виконання (або не виконання) умови
Багато фішок	(12)	позначає багатократне виконання умови
Багато дуг	16 →	позначає велику кількість зв'язків

Рисунок 2.1 – Основні елементи мереж Петрі [2]

Для того, щоб представити систему засобами мереж Петрі необхідно [2]:

- виділити події, що виникають в системі, і поставити у відповідність кожній події перехід мережі Петрі;
- з'ясувати умови, при яких виникає кожна з подій, і поставити у відповідність кожній умові позицію мережі Петрі;
- визначити кількість фішок у позиції мережі Петрі, що символізує виконання умови;
- з'єднати позиції та переходи відповідно до логіки виникнення подій у системі: якщо умова передуює виконанню події, то з'єднати в мережі Петрі відповідну позицію з відповідним переходом;
- якщо умова являється наслідком виконання події, то з'єднати в мережі Петрі відповідний перехід з відповідною позицією;
- з'ясувати зміни, які відбуваються в системі при здійсненні кожної події, і поставити у відповідність змінам переміщення визначеної кількості фішок із позицій в переходи та з переходів у позиції;

- визначити числові значення часових затримок в переходах мережі Петрі;
- визначити стан мережі Петрі на початку моделювання.

Для запуску мережі Петрі необхідно, щоб вона була правильно побудована, тобто, щоб усі переходи мали вхідні та вихідні дуги, а усі позиції вхідні і/або вихідні дуги, а також щоб були розставлені маркери у тих позиціях, де це цього потребує модель. Також важливо, щоб дугою були з'єднані лише позиція і перехід, а не дві позиції чи два переходи.

Також необхідно описати правило активації переходів. Перехід може бути ініційований лише у випадку, коли на вході (у місцях, з яких йде дуга до цього переходу) присутні маркери у кількості, рівної кількості зв'язків. Якщо ця умова виконується, то перехід активується, після чого із всіх вхідних позицій маркери видаляються у кількості, яка відповідає кількості зв'язків, та у той же момент на всі вихідні позиції додаються маркери у кількості, рівній кількості зв'язків [2].

3. АЛГОРИТМІЗАЦІЯ МОДЕЛІ ТА ЇЇ РЕАЛІЗАЦІЯ

У рамках цього курсового проекту було розроблено універсальний алгоритм імітації класичної мережі Петрі мовою TypeScript. У цьому розділі буде описано усі технічні моменти щодо цієї розробки, у тому числі сам алгоритм та його реалізація.

3.1. Опис складових алгоритму імітації

Перш за все, необхідно визначитись із складовими алгоритму імітації. Найважливішими складовими є [2]:

- спосіб просування модельного часу;
- спосіб просування стану моделі в часі;
- спосіб збору інформації про модель в процесі імітації.

3.1.1. Вибір алгоритму просування модельного часу

Існують три способи просування модельного часу [2]:

- за принципом Δt ;
- за принципом найближчої події;
- за принципом послідовного проведення об'єктів уздовж моделі.

У цьому курсовому проекті буде використовуватись алгоритм просування модельного часу за принципом найближчої події.

3.1.2. Вибір алгоритму просування стану моделі в залежності від часу

Існують три способи просування стану моделі в залежності від часу [2]:

- орієнтований на події;
- орієнтований на дії;
- процесно-орієнтований.

У цьому курсовому проекті буде використовуватись алгоритм просування стану моделі в залежності від часу орієнтований на події.

3.1.3. Вибір алгоритму збирання інформації про поведінку моделі у процесі імітації

Статистично у кінці кожної ітерації для кожного місця визначається середня кількість маркерів, а для переходів – середня навантаженість та кількість спрацювань.

3.2. Загальний опис реалізації алгоритму

Алгоритм передбачає обмеження кількості ітерацій (тіків) роботи симуляції. Стан мережі Петрі у кожній такій ітерації описується наступними речами:

- кількість токенів у кожній позиції;
- стан кожного переходу (активований чи не активований).

Як вже описано у попередньому розділі, правилами зміни стану є виконання умов активації переходу. Тобто, при наявності достатньої кількості маркерів у вхідних позиціях для цього переходу відбувається видалення маркерів з вхідних позицій у такій кількості, що дорівнює кількості зв'язків та додаються маркери у вихідні позиції у такій кількості, що відповідають кількості зв'язків.

Кожна ітерація складається з наступних частин:

- вихід маркерів з активованих переходів;
- активація переходів з алгоритмом вирішення конфліктів, що представляє собою рівноймовірний відбір переходів у випадковому порядку для розгляду на активацію з числа конфліктуючих;
- якщо умови для активації переходу виконані, вхід маркерів до переходу;
- формування статистики за результатами ітерації;
- оновлення часу (перехід до наступної ітерації);
- якщо не досягнуте обмеження кількості ітерацій, повторюємо для нового часу (нової ітерації).

Детальніше алгоритм вирішення конфліктів працює наступним чином:

1. створюється копія масиву переходів;
2. якщо масив пустий, закінчуємо роботу алгоритму;
3. обираємо випадковий індекс у цьому масиві, під яким міститься перехід;
4. активуємо цей випадково обраний перехід, якщо він задовольняє умовам активації, описаним вище;
5. видаляємо перехід з масиву;
6. повертаємось до пункту 2.

3.3. Опис засобів розробки алгоритму

Розробка буде виконуватись мовою TypeScript. У якості платформи для виконання коду було обрано Node.js із стандартним пакетним менеджером npm. У якості редактора коду буде використовуватись Visual Studio Code.

3.4. Опис тестових моделей

У рамках цього курсового проекту було вирішено провести дослідження правильності роботи універсального алгоритму імітації класичної мережі Петрі на трьох моделях.

3.4.1. Тестова модель для визначення правильності роботи алгоритму

Першою моделлю є спеціальна тестова модель для визначення правильності роботи алгоритму імітації, зокрема, правильності вирішення конфліктів. Вона складається з генератора запитів, обробника запитів (має свій обмежений ресурс, який блокується під час обробки запиту) та 4 списки, в які рівномірно мають розподілятися запити за правилом вирішення конфліктів за замовчуванням. Графічна схема у формалізмі мереж Петрі цієї моделі показана на рисунку 2.2. Для побудови цієї графічної схеми використовувалось програмне забезпечення PetriObjModelPaint [3].

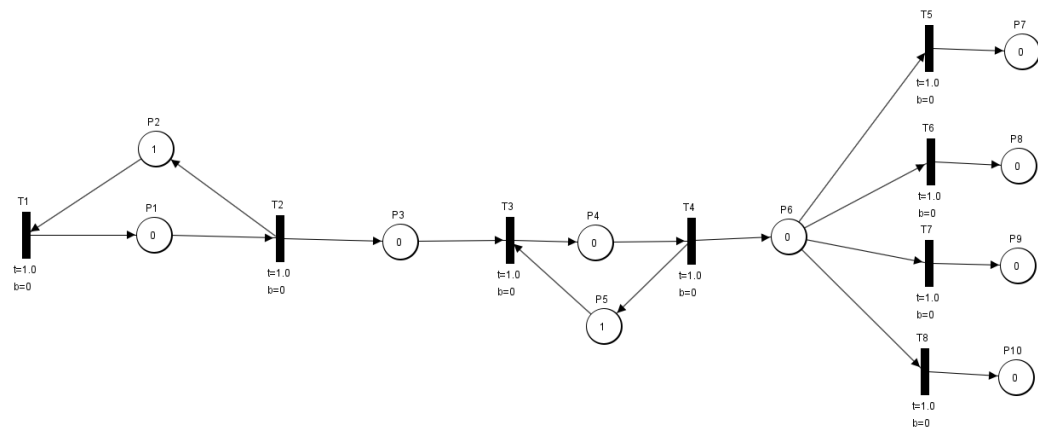


Рисунок 2.2 – Тестова модель для визначення правильності роботи імітаційного алгоритму

3.4.2. Задача про філософів

Наступною моделлю є модель задачі про філософів. Передусім, ця задача є гарним прикладом таких негативних явищ у паралельному програмуванні, як дедлоки. Умови цієї задачі звучать наступним чином. За круглим столом сидять п'ять філософів. Зліва та справа кожного з них лежить паличка для їжі таким чином, що одна і та ж паличка є сусідньою для двох філософів (тобто, паличок всього п'ять, як і філософів). На початку усі філософи знаходяться у стані роздумів. Як тільки починається симуляція, філософи намагаються взяти палочку зліва і справа від себе, після чого переходять у стан коштування (беруть одну рисинку та їдять) та потім кладуть обидві палички на свої місця. Далі такий алгоритм дій повторюється. Ідея в тому, що палички представляють собою спільний ресурс, а філософи – це процеси, які змагаються за цей ресурс, намагаючись його захопити для виконання задачі. У цьому прикладі філософи захоплюють обидві палички одночасно, тож явище дедлоку у цьому випадку спостерігатись не буде. Графічна схема у формалізмі мереж Петрі цієї моделі показана на рисунку 2.3. Для побудови цієї графічної схеми використовувалось програмне забезпечення PetriObjModelPaint [3].

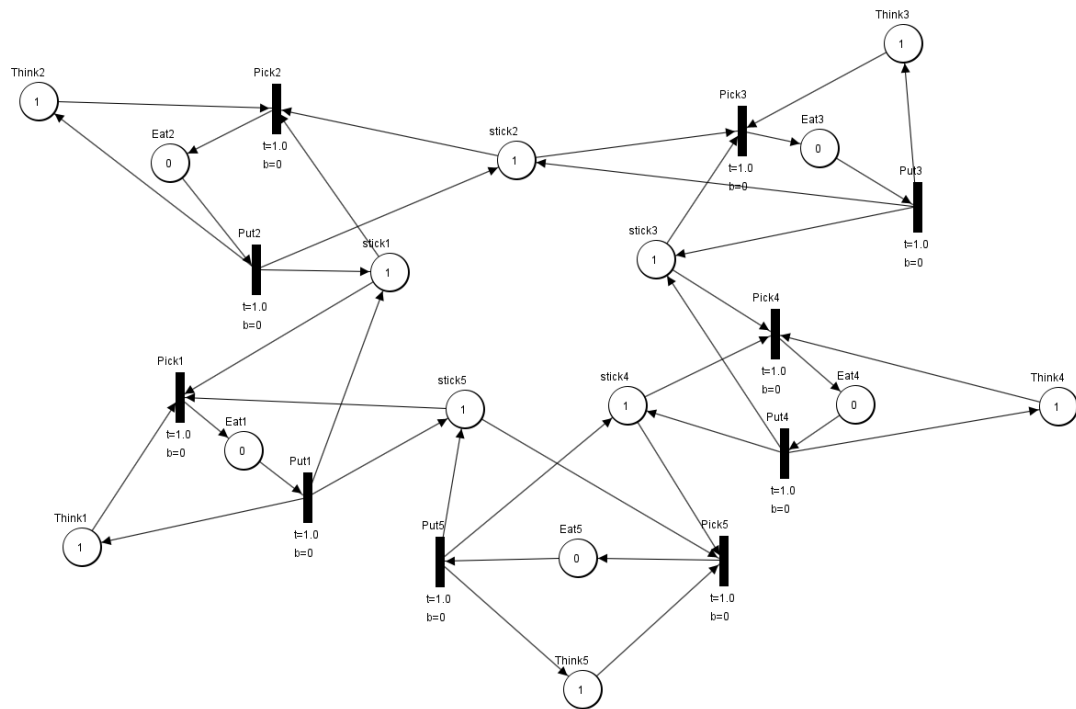


Рисунок 2.3 – Модель для вирішення задачі про філософів

3.4.3. Задача про філософів з явищем дедлоку

У цьому випадку спеціально допущено помилку, що призведе до дедлоку. Цього разу філософи захоплюють обидві палички не одночасно, а по одній. У такому випадку цілком імовірним є те, що кожен філософ візьме по одній паличці та буде нескінченно очікувати на другу, що і представляє собою явище дедлоку. Графічна схема у формалізмі мереж Петрі цієї моделі показана на рисунку 2.4. Для побудови цієї графічної схеми використовувалось програмне забезпечення PetriObjModelPaint [3].

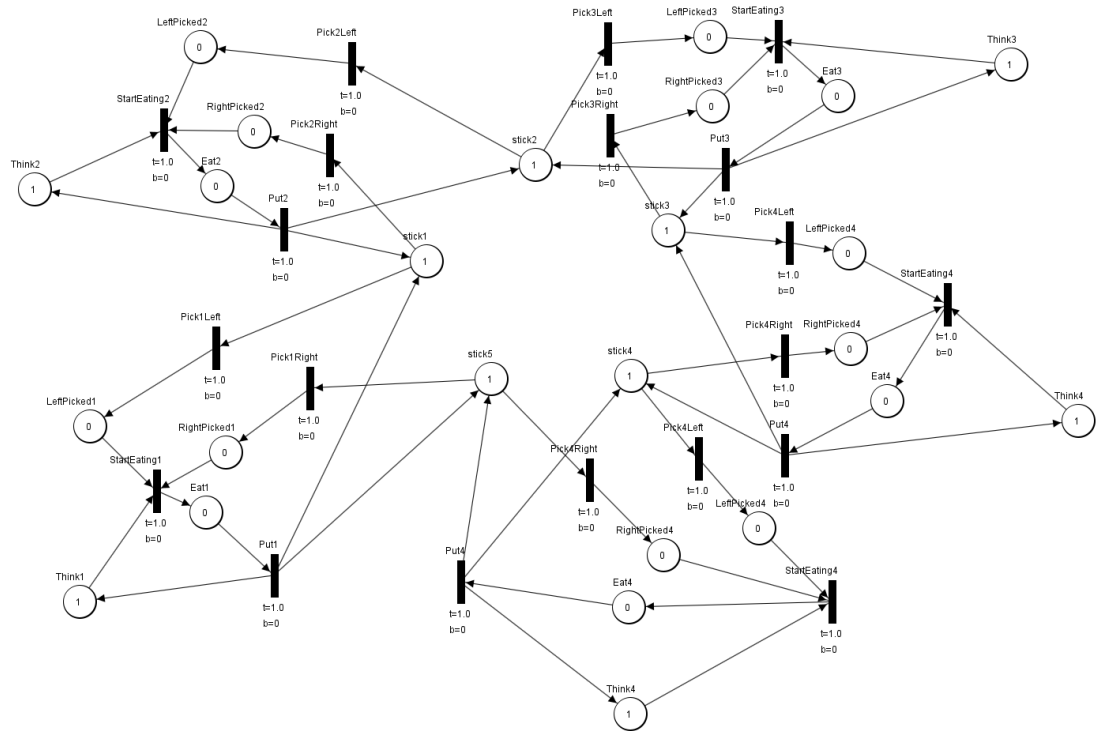


Рисунок 2.4 – Модель для задачі про філософів з явищем дедлоку

3.5. Детальний опис компонентів реалізації алгоритму

Реалізація алгоритму складається з наступних компонентів:

- інтерфейси, що описують дуги (рисунок 3.1);


```

You, 3 дня назад | 1 author (You)
1  import { Place } from './Place';
2  import { Transition } from './Transition';
3
You, 3 дня назад | 1 author (You)
4  export interface ArcIn {
5      readonly source: Place;
6      readonly target: Transition;
7      readonly multiplicity: number;
8  }
9
You, 3 дня назад | 1 author (You)
10 export interface ArcOut {
11     readonly target: Place;
12     readonly source: Transition;
13     readonly multiplicity: number;
14 }
15
16 export type Arc = ArcIn | ArcOut;
17

```

Рисунок 3.1 – Інтерфейси, що описують дуги

– клас місця (рисунок 3.2);

```

You, 3 дня назад | 1 author (You)
1  export class Place {
2      private static nextPlaceId = 1;
3
4      public meanValueParts = 0;
5
6      constructor(
7          public markers = 0,
8          public readonly name = `P${Place.nextPlaceId++}`
9      ) {}
10 }
11

```

Рисунок 3.2 – Клас місця

– клас переходу (рисунок 3.3);

```
you, 2 Часа Назад | T author (you)
1  export class Transition {
2      private static nextTransitionId = 1;
3
4      public quantity = 0;
5      public meanBusinessParts = 0;
6
7      constructor(
8          public readonly name = `T${Transition.nextTransitionId++}`,
9          public processing = false
10     ) {}
11 }
12 |
```

Рисунок 3.3 – Клас переходу

– допоміжний клас ArcsMap для налаштування зв'язків між місцями та переходами, який допоможе описати зв'язки кожного переходу декларативно (рисунок 3.4);

```

1 import { ArcIn, ArcOut } from './Arc';
2 import { Place } from './Place';
3 import { Transition } from './Transition';
4
5 export class ArcsMap extends Map<
6   Transition,
7   { arcsIn: ArcIn[]; arcsOut: ArcOut[] }
8 > {
9   public connectIn(source: Place, target: Transition, multiplicity: number) {
10     const arcs = this.get(target) || { arcsIn: [], arcsOut: [] };
11
12     arcs.arcsIn.push({ source, target, multiplicity });
13
14     this.set(target, arcs);
15
16     return this;
17   }
18
19   public connectOut(source: Transition, target: Place, multiplicity: number) {
20     const arcs = this.get(source) || { arcsIn: [], arcsOut: [] };
21
22     arcs.arcsOut.push({ source, target, multiplicity });
23
24     this.set(source, arcs);
25
26     return this;
27   }
28
29   public setupTransitionConnections(
30     transition: Transition,
31     options: {
32       sources?: { place: Place; multiplicity: number }[];
33       targets?: { place: Place; multiplicity: number }[];
34     }
35   ) {
36     const sources = options.sources || [];
37     const targets = options.targets || [];
38
39     for (const { place: source, multiplicity } of sources) {
40       this.connectIn(source, transition, multiplicity);
41     }
42
43     for (const { place: target, multiplicity } of targets) {
44       this.connectOut(transition, target, multiplicity);
45     }
46
47     return this;
48   }
49 }
50

```

Рисунок 3.4 – Допоміжний клас ArcsMap

– клас PetriNet, що містить декілька методів: запуск симуляції (рисунок 3.5), розрахунку статистики та логування результатів (рисунок 3.6).

```

14 public simulate(ticks: number) {
15     while (this.currentTick <= ticks) {
16         // OUT
17         for (const transition of this.transitions) {
18             const transitionArcs = this.arcs.get(transition);
19             if (!transitionArcs) continue;
20
21             const { arcsOut } = transitionArcs;
22
23             if (transition.processing) {
24                 for (const arcOut of arcsOut) {
25                     arcOut.target.markers += arcOut.multiplicity;
26                 }
27                 transition.processing = false;
28                 transition.quantity++;
29             }
30         }
31
32         // IN
33         const transitionsToBeActivated: Transition[] = this.transitions.slice();
34
35         while (transitionsToBeActivated.length !== 0) {
36             const randomIndex = Math.floor(
37                 Math.random() * transitionsToBeActivated.length
38             );
39
40             const transition = transitionsToBeActivated.splice(randomIndex, 1)[0];
41
42             const transitionArcs = this.arcs.get(transition);
43             if (!transitionArcs) continue;
44
45             const { arcsIn } = transitionArcs;
46
47             if (
48                 !arcsIn.every((arcIn) => arcIn.source.markers >= arcIn.multiplicity)
49             )
50                 continue;
51
52             for (const arcIn of arcsIn) {
53                 arcIn.source.markers -= arcIn.multiplicity;
54             }
55
56             transition.processing = true;
57         }
58
59         this.doStatistics();
60
61         // if (this.currentTick % 1000 === 0) {
62         //     console.log(`!!!${this.currentTick}!!!`);
63         //     this.logResults();
64         // }
65         this.currentTick++;
66     }
67 }

```

Рисунок 3.5 – Метод simulate класу PetriNet

```

149 public doStatistics() {
150     for (const place of this.places) {
151         place.meanValueParts += place.markers;
152     }
153
154     for (const transition of this.transitions) {
155         transition.meanBusinessParts += transition.processing ? 1 : 0;
156     }
157 }
158
159 public logResults() {
160     console.log('PLACES');
161
162     for (const place of this.places) {
163         console.log(`${place.name}:`);
164         console.log(`current markers: ${place.markers}`);
165         console.log(`mean value: ${place.meanValueParts / this.currentTick}`);
166     }
167
168     console.log();
169
170     console.log('TRANSITIONS');
171     for (const transition of this.transitions) {
172         console.log(`${transition.name}:`);
173         console.log(`current processing: ${transition.processing}`);
174         console.log(`quantity: ${transition.quantity}`);
175         console.log(
176             `mean business: ${transition.meanBusinessParts / this.currentTick}`
177         );
178     }
179 }

```

Рисунок 3.6 – Методи doStatistics та logResults класу PetriNet

Тепер давайте детальніше розглянемо реалізацію алгоритму симуляції.

Перш за все, алгоритм починається з обмеження на кількість часу роботи (або ж кількість ітерацій) алгоритму (рисунок 3.7).

```

13
14 public simulate(ticks: number) {
15     while (this.currentTick <= ticks) {
16         // OUT

```

Рисунок 3.7 – Обмеження на час роботи (кількість ітерацій) алгоритму

Наступним кроком є виконання виходу маркерів з активованих переходів (рисунок 3.8).

```

16      // OUT
17      for (const transition of this.transitions) {
18          const transitionArcs = this.arcs.get(transition);
19          if (!transitionArcs) continue;
20
21          const { arcsOut } = transitionArcs;
22
23          if (transition.processing) {
24              for (const arcOut of arcsOut) {
25                  arcOut.target.markers += arcOut.multiplicity;
26              }
27              transition.processing = false;
28              transition.quantity++;
29          }
30      }

```

Рисунок 3.8 – Виконання виходу маркерів з активованих переходів

Далі відбувається вхід маркерів разом з вирішенням конфліктів переходів (рисунок 3.9). Варто звернути увагу на те, що переходи розглядаються у випадковому порядку, що забезпечує вирішення конфліктів переходів шляхом рівноймовірного вибору певного переходу для розгляду на активацію з числа конфліктуючих.

```

32 // IN
33 const transitionsToBeActivated: Transition[] = this.transitions.slice();
34
35 while (transitionsToBeActivated.length !== 0) {
36     const randomIndex = Math.floor(
37         Math.random() * transitionsToBeActivated.length
38     );
39
40     const transition = transitionsToBeActivated.splice(randomIndex, 1)[0];
41
42     const transitionArcs = this.arcs.get(transition);
43     if (!transitionArcs) continue;
44
45     const { arcsIn } = transitionArcs;
46
47     if (
48         !arcsIn.every((arcIn) => arcIn.source.markers >= arcIn.multiplicity)
49     )
50         continue;
51
52     for (const arcIn of arcsIn) {
53         arcIn.source.markers -= arcIn.multiplicity;
54     }
55
56     transition.processing = true;
57 }

```

Рисунок 3.9 – Виконання входу маркерів та вирішення конфліктів між переходами

Під кінець ітерації оновлюється поточний час (номер ітерації) та збирається статистика (рисунок 3.10).

```

59 this.doStatistics();
60
61 // if (this.currentTick % 1000 === 0) {
62 //     console.log(`!!!${this.currentTick}!!!`);
63 //     this.logResults();
64 // }
65 this.currentTick++;

```

Рисунок 3.10 – Оновлення часу та збір статистики

Тепер ознайомимось безпосередньо з функціями симуляції формалізованих моделей. Код для створення та запуску симуляції тестової моделі зображено на рисунку 3.11. Результат запуску цієї моделі зображено на рисунку 3.12.

```

You, 4 years ago | 1 author (You)
1 import { ArcsMap } from './PetriNet/ArcsMap';
2 import { PetriNet } from './PetriNet/PetriNet';
3 import { Place } from './PetriNet/Place';
4 import { Transition } from './PetriNet/Transition';
5
6 const runTestTask = () => {
7   const place1 = new Place();
8   const place2 = new Place(1);
9   const place3 = new Place();
10  const place4 = new Place();
11  const place5 = new Place(1);
12  const place6 = new Place();
13  const place7 = new Place();
14  const place8 = new Place();
15  const place9 = new Place();
16  const place10 = new Place();
17
18  const places = [
19    place1,
20    place2,
21    place3,
22    place4,
23    place5,
24    place6,
25    place7,
26    place8,
27    place9,
28    place10,
29  ];
30
31  const tr1 = new Transition();
32  const tr2 = new Transition();
33  const tr3 = new Transition();
34  const tr4 = new Transition();
35  const tr5 = new Transition();
36  const tr6 = new Transition();
37  const tr7 = new Transition();
38  const tr8 = new Transition();
39
40  const transitions = [tr1, tr2, tr3, tr4, tr5, tr6, tr7, tr8];
41
42  const arcsMap = new ArcsMap();
43
44  arcsMap
45    .setupTransitionConnections(tr1, {
46      sources: [{ place: place2, multiplicity: 1 }],
47      targets: [{ place: place1, multiplicity: 1 }],
48    })
49    .setupTransitionConnections(tr2, {
50      sources: [{ place: place1, multiplicity: 1 }],
51      targets: [
52        { place: place3, multiplicity: 1 },
53        { place: place2, multiplicity: 1 },
54      ],
55    })
56    .setupTransitionConnections(tr3, {
57      sources: [
58        { place: place3, multiplicity: 1 },
59        { place: place5, multiplicity: 1 },
60      ],
61      targets: [{ place: place4, multiplicity: 1 }],
62    })
63    .setupTransitionConnections(tr4, {
64      sources: [{ place: place4, multiplicity: 1 }],
65      targets: [
66        { place: place5, multiplicity: 1 },
67        { place: place6, multiplicity: 1 },
68      ],
69    })
70    .setupTransitionConnections(tr5, {
71      sources: [{ place: place6, multiplicity: 1 }],
72      targets: [{ place: place7, multiplicity: 1 }],
73    })
74    .setupTransitionConnections(tr6, {
75      sources: [{ place: place6, multiplicity: 1 }],
76      targets: [{ place: place8, multiplicity: 1 }],
77    })
78    .setupTransitionConnections(tr7, {
79      sources: [{ place: place6, multiplicity: 1 }],
80      targets: [{ place: place9, multiplicity: 1 }],
81    })
82    .setupTransitionConnections(tr8, {
83      sources: [{ place: place6, multiplicity: 1 }],
84      targets: [{ place: place10, multiplicity: 1 }],
85    });
86
87  const petriNet = new PetriNet(places, transitions, arcsMap);
88
89  petriNet.simulate(1000);
90  petriNet.logResults();
91 };
92
93 export default runTestTask;
94

```

Рисунок 3.11 – Код для створення та симуляції тестової моделі


```

P5 E:\Projects\system-modelling-coursework> npx ts-node .\src\index.ts\
PLACES
P1:
current markers: 0
mean value: 0
P2:
current markers: 0
mean value: 0
P3:
current markers: 0
mean value: 0
P4:
current markers: 0
mean value: 0
P5:
current markers: 0
mean value: 0.00001999980000199998
P6:
current markers: 0
mean value: 0
P7:
current markers: 12718
mean value: 6313.4682453175465
P8:
current markers: 12385
mean value: 6262.366246337537
P9:
current markers: 12461
mean value: 6221.974870251297
P10:
current markers: 12434
mean value: 6199.940700592994

TRANSITIONS
T1:
current processing: false
quantity: 50000
mean business: 0.4999950000499995
T2:
current processing: true
quantity: 49999
mean business: 0.4999950000499995
T3:
current processing: false
quantity: 49999
mean business: 0.4999850001499985
T4:
current processing: true
quantity: 49998
mean business: 0.4999850001499985
T5:
current processing: false
quantity: 12718
mean business: 0.12717872821271786
T6:
current processing: false
quantity: 12385
mean business: 0.12384876151238487
T7:
current processing: false
quantity: 12461
mean business: 0.12460875391246087
T8:
current processing: false
quantity: 12434
mean business: 0.12433875661243388

```

Рисунок 3.12 – Результат запуску тестової моделі

Код для створення та запуску симуляції моделі для вирішення задачі про філософів зображено на рисунках 3.13 і 3.14. Результат запуску цієї моделі зображено на рисунку 3.15.

```

You, 4 дня назад | 1 author (You)
1 import { ArcsMap } from './PetriNet/ArcsMap';
2 import { PetriNet } from './PetriNet/PetriNet';
3 import { Place } from './PetriNet/Place';
4 import { Transition } from './PetriNet/Transition';
5
6 const runPhilosophyTask = () => {
7   const stick1 = new Place(1, 'stick1');
8   const stick2 = new Place(1, 'stick2');
9   const stick3 = new Place(1, 'stick3');
10  const stick4 = new Place(1, 'stick4');
11  const stick5 = new Place(1, 'stick5');
12
13  const philosopher1Eat = new Place(0, 'Eat1');
14  const philosopher1Think = new Place(1, 'Think1');
15
16  const philosopher2Eat = new Place(0, 'Eat2');
17  const philosopher2Think = new Place(1, 'Think2');
18
19  const philosopher3Eat = new Place(0, 'Eat3');
20  const philosopher3Think = new Place(1, 'Think3');
21
22  const philosopher4Eat = new Place(0, 'Eat4');
23  const philosopher4Think = new Place(1, 'Think4');
24
25  const philosopher5Eat = new Place(0, 'Eat5');
26  const philosopher5Think = new Place(1, 'Think5');
27
28  const places = [
29    stick1,
30    stick2,
31    stick3,
32    stick4,
33    stick5,
34    philosopher1Eat,
35    philosopher1Think,
36    philosopher2Eat,
37    philosopher2Think,
38    philosopher3Eat,
39    philosopher3Think,
40    philosopher4Eat,
41    philosopher4Think,
42    philosopher5Eat,
43    philosopher5Think,
44  ];
45
46  const philosopher1Pick = new Transition('Pick1');
47  const philosopher1Put = new Transition('Put1');
48
49  const philosopher2Pick = new Transition('Pick2');
50  const philosopher2Put = new Transition('Put2');
51
52  const philosopher3Pick = new Transition('Pick3');
53  const philosopher3Put = new Transition('Put3');
54
55  const philosopher4Pick = new Transition('Pick4');
56  const philosopher4Put = new Transition('Put4');
57
58  const philosopher5Pick = new Transition('Pick5');
59  const philosopher5Put = new Transition('Put5');
60
61  const transitions = [
62    philosopher1Pick,
63    philosopher1Put,
64    philosopher2Pick,
65    philosopher2Put,
66    philosopher3Pick,
67    philosopher3Put,
68    philosopher4Pick,
69    philosopher4Put,
70    philosopher5Pick,
71    philosopher5Put,
72  ];
73
74  const arcsMap = new ArcsMap();
75
76  arcsMap
77    .setupTransitionConnections(philosopher1Pick, {
78      sources: [
79        { place: stick1, multiplicity: 1 },
80        { place: stick5, multiplicity: 1 },
81        { place: philosopher1Think, multiplicity: 1 },

```

Рисунок 3.13 – Код для створення та симуляції моделі для вирішення задачі про філософів (перша частина)

```

82     ], You, 4 дня назад * Add philosophy task
83     targets: [{ place: philosopher1Eat, multiplicity: 1 }],
84   })
85   .setupTransitionConnections(philosopher1Put, {
86     sources: [{ place: philosopher1Eat, multiplicity: 1 }],
87     targets: [
88       { place: stick1, multiplicity: 1 },
89       { place: stick5, multiplicity: 1 },
90       { place: philosopher1Think, multiplicity: 1 },
91     ],
92   })
93   .setupTransitionConnections(philosopher2Pick, {
94     sources: [
95       { place: stick2, multiplicity: 1 },
96       { place: stick1, multiplicity: 1 },
97       { place: philosopher2Think, multiplicity: 1 },
98     ],
99     targets: [{ place: philosopher2Eat, multiplicity: 1 }],
100   })
101   .setupTransitionConnections(philosopher2Put, {
102     sources: [{ place: philosopher2Eat, multiplicity: 1 }],
103     targets: [
104       { place: stick2, multiplicity: 1 },
105       { place: stick1, multiplicity: 1 },
106       { place: philosopher2Think, multiplicity: 1 },
107     ],
108   })
109   .setupTransitionConnections(philosopher3Pick, {
110     sources: [
111       { place: stick3, multiplicity: 1 },
112       { place: stick2, multiplicity: 1 },
113       { place: philosopher3Think, multiplicity: 1 },
114     ],
115     targets: [{ place: philosopher3Eat, multiplicity: 1 }],
116   })
117   .setupTransitionConnections(philosopher3Put, {
118     sources: [{ place: philosopher3Eat, multiplicity: 1 }],
119     targets: [
120       { place: stick3, multiplicity: 1 },
121       { place: stick2, multiplicity: 1 },
122       { place: philosopher3Think, multiplicity: 1 },
123     ],
124   })
125   .setupTransitionConnections(philosopher4Pick, {
126     sources: [
127       { place: stick4, multiplicity: 1 },
128       { place: stick3, multiplicity: 1 },
129       { place: philosopher4Think, multiplicity: 1 },
130     ],
131     targets: [{ place: philosopher4Eat, multiplicity: 1 }],
132   })
133   .setupTransitionConnections(philosopher4Put, {
134     sources: [{ place: philosopher4Eat, multiplicity: 1 }],
135     targets: [
136       { place: stick4, multiplicity: 1 },
137       { place: stick3, multiplicity: 1 },
138       { place: philosopher4Think, multiplicity: 1 },
139     ],
140   })
141   .setupTransitionConnections(philosopher5Pick, {
142     sources: [
143       { place: stick5, multiplicity: 1 },
144       { place: stick4, multiplicity: 1 },
145       { place: philosopher5Think, multiplicity: 1 },
146     ],
147     targets: [{ place: philosopher5Eat, multiplicity: 1 }],
148   })
149   .setupTransitionConnections(philosopher5Put, {
150     sources: [{ place: philosopher5Eat, multiplicity: 1 }],
151     targets: [
152       { place: stick5, multiplicity: 1 },
153       { place: stick4, multiplicity: 1 },
154       { place: philosopher5Think, multiplicity: 1 },
155     ],
156   });
157
158   const petriNet = new PetriNet(places, transitions, arcsMap);
159
160   petriNet.simulate(1000);
161   petriNet.logResults();
162 }

```

Рисунок 3.14 – Код для створення та симуляції моделі для вирішення задачі про філософів (друга частина)

```

PS E:\Projects\system-modelling-coursework> npx ts-node .\src\index.ts\
PLACES
stick1:
current markers: 1
mean value: 0.20217978202179782
stick2:
current markers: 0
mean value: 0.19618038196180382
stick3:
current markers: 0
mean value: 0.21057894210578942
stick4:
current markers: 0
mean value: 0.19838016198380162
stick5:
current markers: 0
mean value: 0.19258074192580743
Eat1:
current markers: 0
mean value: 0
Think1:
current markers: 1
mean value: 0.6053394660533946
Eat2:
current markers: 0
mean value: 0
Think2:
current markers: 1
mean value: 0.5967403259674032
Eat3:
current markers: 0
mean value: 0
Think3:
current markers: 0
mean value: 0.5993400659934006
Eat4:
current markers: 0
mean value: 0
Think4:
current markers: 1
mean value: 0.6111388861113889
Eat5:
current markers: 0
mean value: 0
Think5:
current markers: 0
mean value: 0.5871412858714129

TRANSITIONS
Pick1:
current processing: false
quantity: 1973
mean business: 0.1972802719728027
Put1:
current processing: false
quantity: 1973
mean business: 0.1972802719728027
Pick2:
current processing: false
quantity: 2016
mean business: 0.20157984201579843
Put2:
current processing: false
quantity: 2016
mean business: 0.20157984201579843
Pick3:
current processing: false
quantity: 2003
mean business: 0.20027997200279973
Put3:
current processing: true
quantity: 2002
mean business: 0.20027997200279973
Pick4:
current processing: false
quantity: 1944
mean business: 0.19438056194380562
Put4:
current processing: false
quantity: 1944
mean business: 0.19438056194380562
Pick5:
current processing: false
quantity: 2064
mean business: 0.20637936206379362
Put5:
current processing: true
quantity: 2063
mean business: 0.20637936206379362

Mean time thinking: 0.6
Mean eaten: 1999.6

```

Рисунок 3.15 – Результат запуску моделі для вирішення задачі про філософів

Код для створення та запуску симуляції моделі для вирішення задачі про філософів із явищем дедлоку можна побачити у додатку А. Результат запуску цієї моделі зображено на рисунку 3.16.

```

PS E:\Projects\system-modelling-coursework> npx ts-node .\src\index.ts\
PLACES
stick1:
current markers: 0
mean value: 0
stick2:
current markers: 0
mean value: 0
stick3:
current markers: 0
mean value: 0
stick4:
current markers: 0
mean value: 0
stick5:
current markers: 0
mean value: 0
LeftPicked1:
current markers: 1
mean value: 0.998001998001998
RightPicked1:
current markers: 0
mean value: 0
Eat1:
current markers: 0
mean value: 0
Think1:
current markers: 1
mean value: 0.999000999000999
LeftPicked2:
current markers: 1
mean value: 0.998001998001998
RightPicked2:
current markers: 0
mean value: 0
Eat2:
current markers: 0
mean value: 0
Think2:
current markers: 1
mean value: 0.999000999000999
LeftPicked3:
current markers: 1
mean value: 0.998001998001998
RightPicked3:
current markers: 0
mean value: 0
Eat3:
current markers: 0
mean value: 0
Think3:
current markers: 1
mean value: 0.999000999000999
LeftPicked4:
current markers: 1
mean value: 0.998001998001998
RightPicked4:
current markers: 0
mean value: 0
Eat4:
current markers: 0
mean value: 0
Think4:
current markers: 1
mean value: 0.999000999000999
LeftPicked5:
current markers: 1
mean value: 0.998001998001998
RightPicked5:
current markers: 0
mean value: 0
Eat5:
current markers: 0
mean value: 0
Think5:
current markers: 1
mean value: 0.999000999000999

TRANSITIONS
Pick1Left:
current processing: false
quantity: 1
mean business: 0.000999000999000999
Pick1Right:
current processing: false
quantity: 0
mean business: 0
StartEating1:
current processing: false
quantity: 0
mean business: 0
Put1:
current processing: false
quantity: 0
mean business: 0
Pick2Left:
current processing: false
quantity: 1
mean business: 0.000999000999000999
Pick2Right:
current processing: false
quantity: 0
mean business: 0
StartEating2:
current processing: false
quantity: 0
mean business: 0

```

Рисунок 3.16 – Результат запуску моделі для вирішення задачі про філософів з явищем дедлоку

Як бачимо із результатів запуску симуляції тестової моделі (рисунок 3.12), можна стверджувати, що алгоритм вирішення конфліктів працює правильно, адже запити в останніх місцях розподілені рівномірно. Також, порівнюючи результати запусків симуляції моделі для вирішення задачі про філософів (рисунок 3.15) та такої ж, але з явищем дедлоку (рисунок 3.16), можна стверджувати про наявність дедлоку у другому випадку, адже середнє значення місця “Think” у першому випадку близько 0.6, у другому – 0.99, що свідчить про недоступність ресурсу у вигляді обох паличок для їжі через взаємне блокування ресурсів.

3.6. Верифікація моделі

З метою верифікації кількість ітерацій була обмежена до 10000 ітерацій. Таблиця верифікації моделі зображена на рисунку 3.17. Для отримання статистично значимого значення, для кожного прогону було проведено 5 запусків і внесено в таблицю середнє значення.

Прогін	Вхідні змінні										Вихідні змінні				
	Кількість паличок між філософами					Початкова кількість токенів у позиції Think філософа					Середня кількість токенів у позиції Think філософа				
	1-2	2-3	3-4	4-5	5-1	1	2	3	4	5	1	2	3	4	5
1	2	2	2	2	2	1	1	1	1	1	0	0	0	0	0
2	1	2	2	2	2	1	1	1	1	1	0,5	0,5	0	0	0
3	1	1	2	2	2	1	1	1	1	1	0,33	0,67	0,33	0	0
4	1	1	1	2	2	1	1	1	1	1	0,37	0,63	0,63	0,37	0
5	1	1	1	1	2	1	1	1	1	1	0,37	0,63	0,54	0,63	0,37
6	1	1	1	1	1	1	1	1	1	1	0,6	0,6	0,6	0,6	0,6
7	1	1	1	1	1	0	1	1	1	1	0	0,37	0,62	0,63	0,37
8	1	1	1	1	1	0	0	1	1	1	0	0	0,33	0,67	0,33
9	1	1	1	1	1	0	0	0	1	1	0	0	0	0,51	0,49
10	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0
11	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0

Рисунок 3.17 – Таблиця верифікації моделі

Враховуючи таблицю верифікації можна зробити наступні висновки:

- палички представляють собою певний ресурс, за захват яких кожен філософ конкурує з його сусідами, через що їх кількість впливає на середнє значення у позиції Think (позиція очікування);
- при однаковій кількості паличок між філософами та присутності у кожного з них мітки в позиції Think, філософи мають середню кількість токенів у позиції Think, рівну 0.6;
- коли умови між філософами стають такими, що не вимагають від них конкуренції з сусідами за захват ресурсів (або паличок вистачає на всіх, або

сусіди мають 0 у позиції Think), філософи майже не знаходяться у стані очікування;

— зміна параметра одного філософа найбільше впливає на вихідні параметри його самого та його сусідів, на інших філософів така зміна впливає набагато менше.

4. ЕКСПЕРИМЕНТИ НА МОДЕЛІ

У рамках цієї курсової роботи експериментування буде відбуватись на моделі для вирішення задачі про філософів. Оскільки особливість класичної мережі Петрі полягає у тому, що переходи активуються негайно і, не змінюючи структуру (або кількість маркерів), неможливо досягти змін у вихідних параметрах моделі, то у рамках цього розділу було вирішено виконати дослідження дерева досяжності, а для виконання факторного експерименту додати до алгоритму імітації часові затримки на переходах. Передусім, це необхідно для визначення вхідних параметрів моделі при факторному експерименті.

4.1. Дерево досяжності

Для побудови дерева досяжності було створено окремий метод, у якому відбувається збирання унікальних станів моделі. Його можна побачити на рисунку 4.1 та у додатку А. Стан токенизується у вигляді рядка з цифрами, що представляють кількість маркерів у позиціях та знаходяться у такому порядку:

1. паличка 1;
2. паличка 2;
3. паличка 3;
4. паличка 4;
5. паличка 5;
6. куштування філософа 1;
7. роздуми філософа 1;
8. куштування філософа 2;
9. роздуми філософа 2;
10. куштування філософа 3;
11. роздуми філософа 3;
12. куштування філософа 4;
13. роздуми філософа 4;

14. куштування філософа 5;
15. роздуми філософа 5.

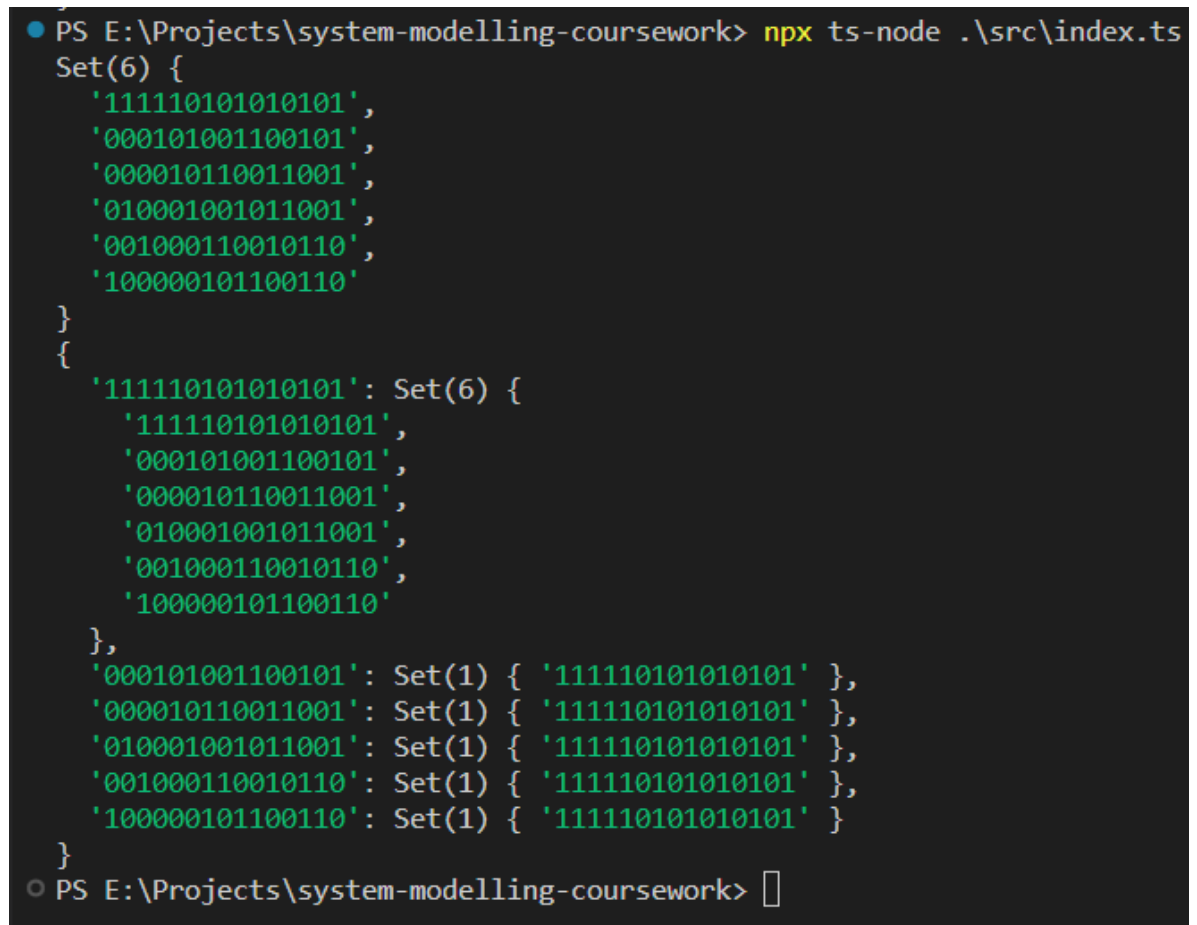
```

69 public traceTree(ticks: number) {
70     const treeNodeDictionary = new Set<string>();
71
72     const initialMarking = this.getMarkingsToken();
73
74     treeNodeDictionary.add(initialMarking);
75     const markingTargets: Record<string, Set<string>> = {
76         [initialMarking]: new Set(),
77     };
78
79     let prevMarking = this.getMarkingsToken();
80
81     while (this.currentTick <= ticks) {
82         // OUT
83         for (const transition of this.transitions) {
84             const transitionArcs = this.arcs.get(transition);
85             if (!transitionArcs) continue;
86
87             const { arcsOut } = transitionArcs;
88
89             if (transition.processing) {
90                 for (const arcOut of arcsOut) {
91                     arcOut.target.markers += arcOut.multiplicity;
92                 }
93                 transition.processing = false;
94                 transition.quantity++;
95             }
96         }
97
98         const nextMarking = this.getMarkingsToken();
99
100         if (!markingTargets[prevMarking]) markingTargets[prevMarking] = new Set();
101         markingTargets[prevMarking].add(nextMarking);
102
103         treeNodeDictionary.add(nextMarking);
104         prevMarking = nextMarking;
105
106         // IN
107         const transitionsToBeActivated: Transition[] = this.transitions.slice();
108
109         while (transitionsToBeActivated.length !== 0) {
110             const randomIndex = Math.floor(
111                 Math.random() * transitionsToBeActivated.length
112             );
113
114             const transition = transitionsToBeActivated.splice(randomIndex, 1)[0];
115
116             const transitionArcs = this.arcs.get(transition);
117             if (!transitionArcs) continue;
118
119             const { arcsIn } = transitionArcs;
120
121             if (
122                 !arcsIn.every((arcIn) => arcIn.source.markers >= arcIn.multiplicity)
123             )
124                 continue;
125
126             for (const arcIn of arcsIn) {
127                 arcIn.source.markers -= arcIn.multiplicity;
128             }
129
130             transition.processing = true;
131         }
132
133         this.doStatistics();
134         this.currentTick++;
135     }
136
137     console.log(treeNodeDictionary);
138     console.log(markingTargets);
139 }
140

```

Рисунок 4.1 – Метод traceTree класу PetriNet

У результаті запуску цього методу було отримано наступні результати (рисунок 4.2).



```

PS E:\Projects\system-modelling-coursework> npx ts-node .\src\index.ts
Set(6) {
  '111110101010101',
  '000101001100101',
  '000010110011001',
  '010001001011001',
  '001000110010110',
  '100000101100110'
}
{
  '111110101010101': Set(6) {
    '111110101010101',
    '000101001100101',
    '000010110011001',
    '010001001011001',
    '001000110010110',
    '100000101100110'
  },
  '000101001100101': Set(1) { '111110101010101' },
  '000010110011001': Set(1) { '111110101010101' },
  '010001001011001': Set(1) { '111110101010101' },
  '001000110010110': Set(1) { '111110101010101' },
  '100000101100110': Set(1) { '111110101010101' }
}
PS E:\Projects\system-modelling-coursework>

```

Рисунок 4.2. Результати роботи методу traceTree

Як бачимо, у результатах відображаються унікальні стани маркування моделі та показані унікальні стани, які є досяжними з цих станів. На основі цих результатів та з урахуванням одночасного виконання переходів можна побудувати дерево досяжності. На жаль, за цими результатами неможливо визначити в результаті яких саме переходів сталася зміна стану, проте ми можемо визначити це, врахувавши початковий та кінцевий стан.

Наприклад, можна орієнтуватись на перші п'ять цифр (стан позицій паличок). Беремо перший стан у списку досяжних переходів з початкового стану ("000101001100101"). Бачимо, що після активування деяких переходів залишилась паличка тільки у четвертій позиції. Співставивши цю інформацію із формалізованою графічною схемою, можна зрозуміти, що таке можливо тільки

при активації переходів “Pick1” та “Pick3”. Так само повторюємо з іншими унікальними станами.

У результаті отримуємо дерево досяжності, яке зображено на рисунку 4.3.

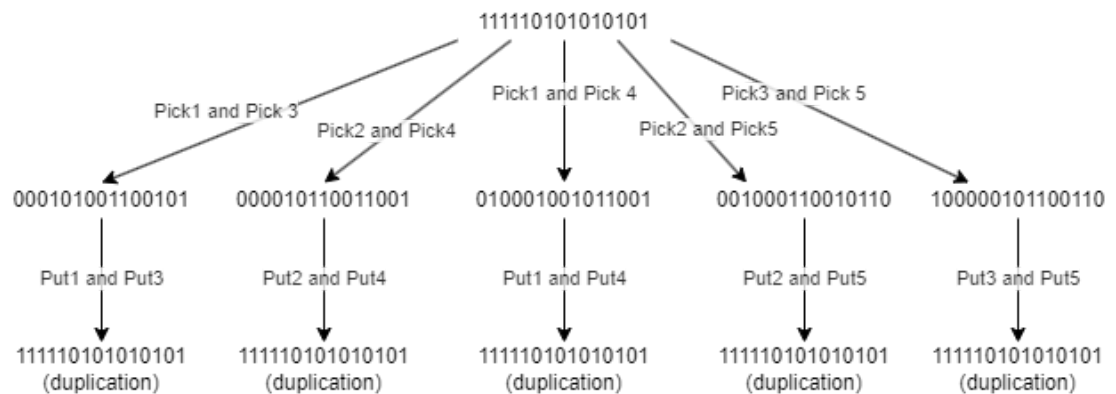


Рисунок 4.3 – Дерево досяжності

Як видно із дерева досяжності, з початкового стану модель переходить у один з 5 станів на основі активації двох переходів Pick (при чому ці переходи не мають стосуватись сусідніх філософів, адже між ними є конфлікт щодо спільного ресурсу). Після цього модель повертається у початковий стан шляхом активації двох відповідних переходів Put.

4.2. Опис вхідних та вихідних параметрів для проведення факторного експерименту

З метою виконання факторного експерименту було додано до алгоритму можливість встановлювати часові затримки.

Вхідними параметрами моделі є:

- часова затримка захвату паличок кожного філософа;
- часова затримка відпускання паличок кожного філософа.

Вихідними параметрами мережі є:

- час перебування філософів у стані роздумів;
- кількість з’їдених рисинок.

4.3. Факторний експеримент

Тепер перейдемо до факторного експерименту. У рамках факторного експерименту відгуком моделі буде вважатись час перебування у стані очікування першого філософа. У зв'язку з цим виберемо декілька факторів для дослідження їх впливу на модель:

- затримка захвату паличок першим філософом;
- затримка захвату паличок другим філософом;
- затримка звільнення паличок третім філософом.

4.3.1. Тактичне планування

У рамках тактичного планування необхідно визначитись з кількістю прогонів та часом моделювання. Зробимо 4 прогони із часом моделювання 10000 (рисунок 4.2). Побудуємо графік для дослідження перехідного періоду (рисунок 4.3).

час	Прогін 1	Прогін 2	Прогін 3	Прогін 4
1000	0,604	0,61	0,64	0,604
2000	0,631	0,619	0,634	0,601
3000	0,622	0,588	0,61	0,608
4000	0,631	0,588	0,603	0,6
5000	0,625	0,588	0,6	0,597
6000	0,617	0,581	0,606	0,602
7000	0,624	0,591	0,609	0,601
8000	0,613	0,588	0,601	0,603
9000	0,607	0,591	0,595	0,609
10000	0,608	0,602	0,604	0,611

Рисунок 4.4 – Результати прогонів під час тактичного планування

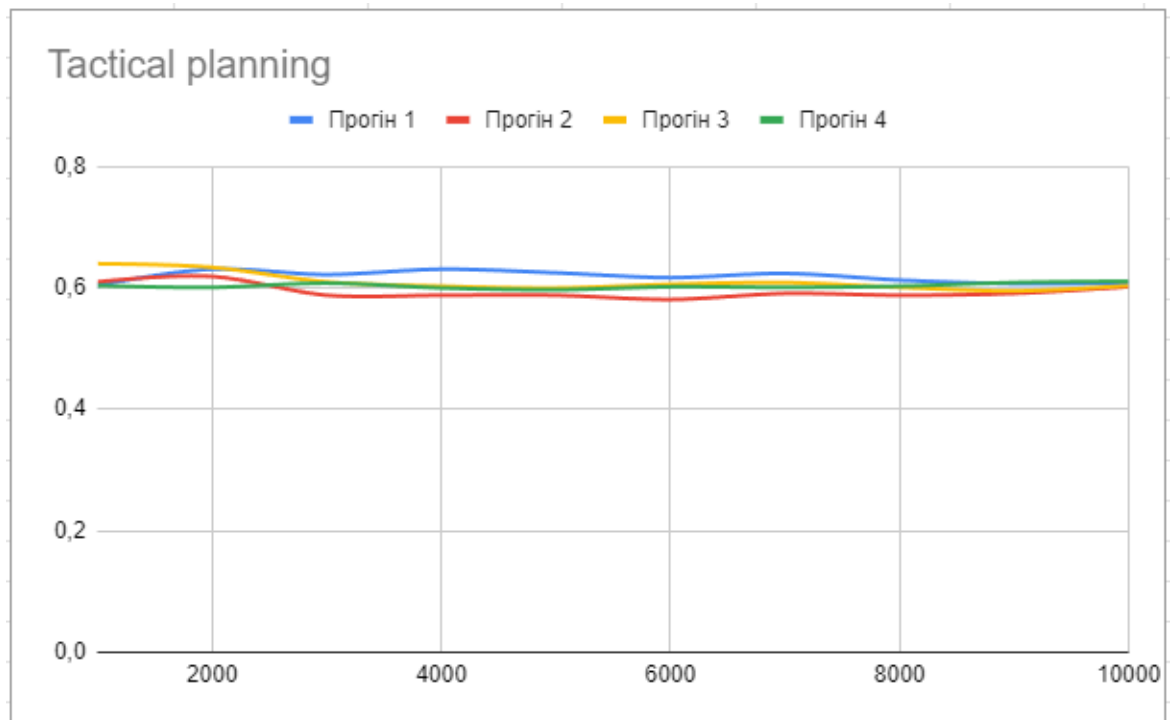


Рисунок 4.5 – Графік для дослідження перехідного періоду

Як бачимо з таблиць та графіків, збирати статистику варто починаючи зі значення 5000, адже графіки стабілізуються після 3000 (і додаємо ще 2000 для впевненості у правильності роботи).

4.3.2. Стратегічне планування

У рамках стратегічного планування необхідно зробити регресійний аналіз, адже оцінка є кількісною.

Введемо наступні позначення:

- x_1 – затримка захвату паличок першим філософом;
- x_2 – затримка захвату паличок другим філософом;
- x_3 – затримка звільнення паличок третім філософом.

Визначимо значення рівнів факторів. Необхідні обчислені значення показано у таблиці 4.1. З метою нормалізації використаємо формули (4.1), (4.2) і (4.3).

$$x_i = \frac{X_i - X_{i0}}{\Delta i} \quad (4.1)$$

$$X_{i0} = \frac{X_{i \max} + X_{i \min}}{2} \quad (4.2)$$

$$\Delta i = \frac{X_{i \max} - X_{i \min}}{2} \quad (4.3)$$

Таблиця 4.1 – Обчислені значення щодо факторів

Фактор	$X_{i \min}$	$X_{i \max}$	X_{i0}	Δi
x_1	1	3	2	1
x_2	1	3	2	1
x_3	1	3	2	1

Тепер виконаємо безпосередньо регресійний аналіз для визначення впливу факторів на відгук моделі. Запускаємо симуляцію моделі та отримуємо дані у проміжку 5000 - 10000 ітерацій. У таблиці 4.2 зображено необхідні фактори, їх знаки та відгук для регресійного аналізу.

Таблиця 4.2 - Регресійний аналіз

2^3	x_0	x_1	x_2	x_3	x_1x_2	x_1x_3	x_2x_3	$x_1x_2x_3$	y
1	+	+	+	+	+	+	+	+	0,6
2	+	-	+	+	-	-	+	-	0,49
3	+	+	-	+	-	+	-	-	0,44
4	+	-	-	+	+	-	-	+	0,5
5	+	+	+	-	+	-	-	-	0,54
6	+	-	+	-	-	+	-	+	0,52
7	+	+	-	-	-	-	+	+	0,49
8	+	-	-	-	+	+	+	-	0,51

Тепер отримаємо апроксимацію функції відгуку моделі, показану у формулі (4.4).

$$y = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + b_4x_1x_2 + b_5x_1x_3 + b_6x_2x_3 + b_7x_1x_2x_3 \quad (4.4)$$

Обчислимо коефіцієнти за формулою:

$$b_0 = \frac{\sum_{i=1}^8 y_i}{8} = 0.51125 \quad (4.5)$$

$$b_1 = \frac{\sum_{i=1}^8 y_i x_{i1}}{8} = 0,00625 \quad (4.6)$$

$$b_2 = \frac{\sum_{i=1}^8 y_i x_{i2}}{8} = 0,02625 \quad (4.7)$$

$$b_3 = \frac{\sum_{i=1}^8 y_i x_{i3}}{8} = -0,00375 \quad (4.8)$$

$$b_4 = \frac{\sum_{i=1}^8 y_i x_{i1} x_{i2}}{8} = 0,02625 \quad (4.9)$$

$$b_5 = \frac{\sum_{i=1}^8 y_i x_{i1} x_{i3}}{8} = 0,00625 \quad (4.10)$$

$$b_6 = \frac{\sum_{i=1}^8 y_i x_{i2} x_{i3}}{8} = 0,01125 \quad (4.11)$$

$$b_7 = \frac{\sum_{i=1}^8 y_i x_{i1} x_{i2} x_{i3}}{8} = 0,01625 \quad (4.12)$$

Як бачимо, фактори x_1 , x_2 та взаємодія факторів $x_1 x_2$ мають найбільший вплив на відгук моделі. Інші фактори (а особливо фактор x_3) не мають суттєвого впливу. Це підтверджує зроблений висновок у підрозділі верифікації моделі про те, що зміна вхідного параметра будь-якого філософа найбільше впливає на самого філософа та його сусідів, але мало впливає на інших філософів.

5. ІНТЕРПРЕТАЦІЯ РЕЗУЛЬТАТІВ МОДЕЛЮВАННЯ ТА ЕКСПЕРИМЕНТІВ

У рамках цього проекту було змодельовано декілька моделей та перевірено на них правильність роботи універсального алгоритму імітації. Було досягнуто розуміння, що алгоритм правильно обробляє активацію переходів та правильно вирішує конфлікти. Також було досліджено явище дедлоку на прикладі задачі про філософів. На основі моделі для тієї ж задачі було проведено експеримент, основними етапами якого були:

- верифікація;
- аналіз дерева досяжності;
- тактичне планування;
- стратегічне планування.

У рамках верифікації спостерігались наступні властивості цієї моделі:

- палички представляють собою певний ресурс, за захват яких кожен філософ конкурує з його сусідами, через що їх кількість впливає на середнє значення у позиції Think (позиція очікування);
- при однаковій кількості паличок між філософами та присутності у кожного з них мітки в позиції Think, філософи мають середню кількість токенів у позиції Think, рівну 0.6;
- коли умови між філософами стають такими, що не вимагають від них конкуренції з сусідами за захват ресурсів (або паличок вистачає на всіх, або сусіди мають 0 у позиції Think), філософи майже не знаходяться у стані очікування;
- зміна параметра одного філософа найбільше впливає на вихідні параметри його самого та його сусідів, на інших філософів така зміна впливає набагато менше.

У рамках аналізу дерева досяжностей було перевірено правильність структури моделі та роботи алгоритму шляхом дослідження досяжності станів. Протягом цього дослідження було побудовано дерево досяжностей та за

допомогою нього зроблено висновок щодо характеру поведінки класичної мережі Петрі.

У рамках тактичного планування було визначено кількість прогонів та час моделювання для факторного експерименту.

У рамках стратегічного планування було проаналізовано вплив деяких факторів на відгук моделі за допомогою методів регресійного аналізу (оскільки оцінка є кількісною).

Цікавим спостереженням є те, що характеристики певного філософа впливають на його сусідів, адже вони поділяють з кожним сусідом по паличці (спільний ресурс).

ВИСНОВКИ

У рамках цієї курсової роботи було розроблено універсальний алгоритм імітації класичної мережі Петрі. Було розглянуто концептуальну модель імітаційного алгоритму. На основі деяких моделей було протестовано роботу алгоритму, зокрема, перевірено правильність обробки конфліктних переходів. Описано ці моделі за допомогою формалізму мереж Петрі. Наступним кроком був опис самого алгоритму імітації та його конкретної реалізації мовою TypeScript. Після цього було проведено верифікацію моделі для вирішення задачі про філософів та проведено факторний експеримент для визначення впливу деяких факторів на відгук моделі із задіянням методів регресійного аналізу. Останнім кроком була інтерпретація та узагальнений опис результатів моделювання та проведених експериментів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Reisig W. Petri Nets and Algebraic Specifications [Електронний ресурс] / Wolfgang Reisig – Режим доступу до ресурсу:
<https://www.sciencedirect.com/science/article/pii/030439759190203E?via%3Dihub>.
2. Стеценко І. В. Моделювання систем [Електронний ресурс] / І. В. Стеценко. – 2011. – Режим доступу до ресурсу:
https://do.ipk.kpi.ua/pluginfile.php/112577/mod_resource/content/1/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8E%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%9D%D0%B0%D0%B2%D1%87%D0%9F%D0%BE%D1%81%D1%96%D0%B1%D0%BD%D0%B8%D0%BA_2011.pdf;
3. Stetsenko I. PetriObjModelPaint [Електронний ресурс] / Inna Stetsenko – Режим доступу до ресурсу:
<https://github.com/StetsenkoInna/PetriObjModelPaint>.

ДОДАТКИ

Додаток А. Текст програмного коду

Також доступ до коду можна отримати за посиланням:
<https://github.com/Secret333Boy/system-modelling-coursework>.

Arc.ts

```
import { Place } from './Place';
import { Transition } from './Transition';

export interface ArcIn {
  readonly source: Place;
  readonly target: Transition;
  readonly multiplicity: number;
}

export interface ArcOut {
  readonly target: Place;
  readonly source: Transition;
  readonly multiplicity: number;
}

export type Arc = ArcIn | ArcOut;
```

ArcMap.ts

```
import { ArcIn, ArcOut } from './Arc';
import { Place } from './Place';
import { Transition } from './Transition';

export class ArcsMap extends Map<
  Transition,
  { arcsIn: ArcIn[]; arcsOut: ArcOut[] }
> {
  public connectIn(source: Place, target: Transition, multiplicity:
number) {
    const arcs = this.get(target) || { arcsIn: [], arcsOut: [] };

    arcs.arcsIn.push({ source, target, multiplicity });

    this.set(target, arcs);

    return this;
  }
}
```

```

    }

    public connectOut(source: Transition, target: Place, multiplicity:
number) {
        const arcs = this.get(source) || { arcsIn: [], arcsOut: [] };

        arcs.arcsOut.push({ source, target, multiplicity });

        this.set(source, arcs);

        return this;
    }

    public setupTransitionConnections(
        transition: Transition,
        options: {
            sources?: { place: Place; multiplicity: number }[];
            targets?: { place: Place; multiplicity: number }[];
        }
    ) {
        const sources = options.sources || [];
        const targets = options.targets || [];

        for (const { place: source, multiplicity } of sources) {
            this.connectIn(source, transition, multiplicity);
        }

        for (const { place: target, multiplicity } of targets) {
            this.connectOut(transition, target, multiplicity);
        }

        return this;
    }
}

```

PetriNet.ts

```

import { ArcsMap } from './ArcsMap';
import { Place } from './Place';
import { Transition } from './Transition';

export class PetriNet {
    private currentTick = 1;

    constructor(

```

```

    private readonly places: Place[] = [],
    private readonly transitions: Transition[] = [],
    private readonly arcsMap: ArcsMap = new ArcsMap()
  ) {}

  public simulate(ticks: number) {
    while (this.currentTick <= ticks) {
      // OUT
      for (const transition of this.transitions) {
        const transitionArcs = this.arcsMap.get(transition);
        if (!transitionArcs) continue;

        const { arcsOut } = transitionArcs;

        if (transition.processing) {
          for (const arcOut of arcsOut) {
            arcOut.target.markers += arcOut.multiplicity;
          }
          transition.processing = false;
          transition.quantity++;
        }
      }

      // IN
      const transitionsToBeActivated: Transition[] =
this.transitions.slice(); // робимо копію масиву переходів

      while (transitionsToBeActivated.length !== 0) {
        const randomIndex = Math.floor(
          Math.random() * transitionsToBeActivated.length
        ); // генеруємо випадковий індекс у копії масиву переходів

        const transition =
transitionsToBeActivated.splice(randomIndex, 1)[0]; // "відриваємо" елемент
за випадковим індексом, згенерованим вище

        const transitionArcs = this.arcsMap.get(transition); // тут
відсутня багатоканальність, адже цей перехід може бути активований тільки
один раз

        if (!transitionArcs) continue;

        // а тут перевіряємо "відірваний" перехід на умови активації
        const { arcsIn } = transitionArcs;
        if (

```

```

                                !arcsIn.every((arcIn) => arcIn.source.markers >=
arcIn.multiplicity)
                                )
                                continue;

                                // у всіх вхідних позиціях забираємо маркери у кількості, що
дорівнює кількості зв'язків
                                for (const arcIn of arcsIn) {
                                    arcsIn.source.markers -= arcIn.multiplicity;
                                }
                                // вказуємо, що перехід активований
                                transition.processing = true;
                                }

                                this.doStatistics();

                                // if (this.currentTick % 1000 === 0) {
                                //     console.log(`!!!${this.currentTick}!!!`);
                                //     this.logResults();
                                // }
                                this.currentTick++;
                                }
                                }

public traceTree(ticks: number) {
    const treeNodeDictionary = new Set<string>();

    const initialMarking = this.getMarkingsToken();

    treeNodeDictionary.add(initialMarking);
    const markingTargets: Record<string, Set<string>> = {
        [initialMarking]: new Set(),
    };

    let prevMarking = this.getMarkingsToken();

    while (this.currentTick <= ticks) {
        // OUT
        for (const transition of this.transitions) {
            const transitionArcs = this.arcsMap.get(transition);
            if (!transitionArcs) continue;

            const { arcsOut } = transitionArcs;

```



```

        if (transition.processing) {
            for (const arcOut of arcsOut) {
                arcOut.target.markers += arcOut.multiplicity;
            }
            transition.processing = false;
            transition.quantity++;
        }
    }

    const nextMarking = this.getMarkingsToken();

    if (!markingTargets[prevMarking]) markingTargets[prevMarking] =
new Set();
    markingTargets[prevMarking].add(nextMarking);

    treeNodeDictionary.add(nextMarking);
    prevMarking = nextMarking;

    // IN
        const transitionsToBeActivated: Transition[] =
this.transitions.slice();

    while (transitionsToBeActivated.length !== 0) {
        const randomIndex = Math.floor(
            Math.random() * transitionsToBeActivated.length
        );

        const transition =
transitionsToBeActivated.splice(randomIndex, 1)[0];

        const transitionArcs = this.arcsMap.get(transition);
        if (!transitionArcs) continue;

        const { arcsIn } = transitionArcs;

        if (
            !arcsIn.every((arcIn) => arcIn.source.markers >=
arcIn.multiplicity)
        )
            continue;

        for (const arcIn of arcsIn) {
            arcIn.source.markers -= arcIn.multiplicity;
        }
    }

```

```

        transition.processing = true;
    }

    this.doStatistics();
    this.currentTick++;
}

console.log(treeNodeDictionary);
console.log(markingTargets);
}

public getMarkings() {
    return this.places.map((place) => place.markers);
}

public getMarkingsToken() {
    return this.getMarkings().join('');
}

public doStatistics() {
    for (const place of this.places) {
        place.meanValueParts += place.markers;
    }

    for (const transition of this.transitions) {
        transition.meanBusinessParts += transition.processing ? 1 : 0;
    }
}

public logResults() {
    console.log('PLACES');

    for (const place of this.places) {
        console.log(`${place.name}:`);
        console.log(`current markers: ${place.markers}`);
        console.log(`mean value:  ${place.meanValueParts} /
this.currentTick`);
    }

    console.log();

    console.log('TRANSITIONS');
    for (const transition of this.transitions) {

```

```

        console.log(`${transition.name}:`);
        console.log(`current processing: ${transition.processing}`);
        console.log(`quantity: ${transition.quantity}`);
        console.log(
            `mean business: ${transition.meanBusinessParts} /
this.currentTick)`
        );
    }
}
}

```

Place.ts

```

export class Place {
    private static nextPlaceId = 1;

    public meanValueParts = 0;

    constructor(
        public markers = 0,
        public readonly name = `P${Place.nextPlaceId++}`
    ) {}
}

```

Transition.ts

```

export class Transition {
    private static nextTransitionId = 1;

    public quantity = 0;
    public meanBusinessParts = 0;

    constructor(
        public readonly name = `T${Transition.nextTransitionId++}`,
        public processing = false
    ) {}
}

```

runTestTask.ts

```

import { ArcsMap } from '../PetriNet/ArcsMap';
import { PetriNet } from '../PetriNet/PetriNet';
import { Place } from '../PetriNet/Place';
import { Transition } from '../PetriNet/Transition';

const runTestTask = () => {

```

```

const place1 = new Place();
const place2 = new Place(1);
const place3 = new Place();
const place4 = new Place();
const place5 = new Place(1);
const place6 = new Place();
const place7 = new Place();
const place8 = new Place();
const place9 = new Place();
const place10 = new Place();

const places = [
    place1,
    place2,
    place3,
    place4,
    place5,
    place6,
    place7,
    place8,
    place9,
    place10,
];

const tr1 = new Transition();
const tr2 = new Transition();
const tr3 = new Transition();
const tr4 = new Transition();
const tr5 = new Transition();
const tr6 = new Transition();
const tr7 = new Transition();
const tr8 = new Transition();

const transitions = [tr1, tr2, tr3, tr4, tr5, tr6, tr7, tr8];

const arcsMap = new ArcsMap();

arcsMap
    .setupTransitionConnections(tr1, {
        sources: [{ place: place2, multiplicity: 1 }],
        targets: [{ place: place1, multiplicity: 1 }],
    })
    .setupTransitionConnections(tr2, {
        sources: [{ place: place1, multiplicity: 1 }],

```

```

        targets: [
            { place: place3, multiplicity: 1 },
            { place: place2, multiplicity: 1 },
        ],
    ))
    .setupTransitionConnections(tr3, {
        sources: [
            { place: place3, multiplicity: 1 },
            { place: place5, multiplicity: 1 },
        ],
        targets: [{ place: place4, multiplicity: 1 }],
    ))
    .setupTransitionConnections(tr4, {
        sources: [{ place: place4, multiplicity: 1 }],
        targets: [
            { place: place5, multiplicity: 1 },
            { place: place6, multiplicity: 1 },
        ],
    ))
    .setupTransitionConnections(tr5, {
        sources: [{ place: place6, multiplicity: 1 }],
        targets: [{ place: place7, multiplicity: 1 }],
    ))
    .setupTransitionConnections(tr6, {
        sources: [{ place: place6, multiplicity: 1 }],
        targets: [{ place: place8, multiplicity: 1 }],
    ))
    .setupTransitionConnections(tr7, {
        sources: [{ place: place6, multiplicity: 1 }],
        targets: [{ place: place9, multiplicity: 1 }],
    ))
    .setupTransitionConnections(tr8, {
        sources: [{ place: place6, multiplicity: 1 }],
        targets: [{ place: place10, multiplicity: 1 }],
    });

    const petriNet = new PetriNet(places, transitions, arcsMap);

    petriNet.simulate(1000);
    petriNet.logResults();
};

export default runTestTask;

```

runPhilosophyTask.ts

```

import { ArcsMap } from '../PetriNet/ArcsMap';
import { PetriNet } from '../PetriNet/PetriNet';
import { Place } from '../PetriNet/Place';
import { Transition } from '../PetriNet/Transition';

const runPhilosophyTask = () => {
  const stick1 = new Place(1, 'stick1');
  const stick2 = new Place(1, 'stick2');
  const stick3 = new Place(1, 'stick3');
  const stick4 = new Place(1, 'stick4');
  const stick5 = new Place(1, 'stick5');

  const philosopher1Eat = new Place(0, 'Eat1');
  const philosopher1Think = new Place(1, 'Think1');

  const philosopher2Eat = new Place(0, 'Eat2');
  const philosopher2Think = new Place(1, 'Think2');

  const philosopher3Eat = new Place(0, 'Eat3');
  const philosopher3Think = new Place(1, 'Think3');

  const philosopher4Eat = new Place(0, 'Eat4');
  const philosopher4Think = new Place(1, 'Think4');

  const philosopher5Eat = new Place(0, 'Eat5');
  const philosopher5Think = new Place(1, 'Think5');

  const places = [
    stick1,
    stick2,
    stick3,
    stick4,
    stick5,
    philosopher1Eat,
    philosopher1Think,
    philosopher2Eat,
    philosopher2Think,
    philosopher3Eat,
    philosopher3Think,
    philosopher4Eat,
    philosopher4Think,
    philosopher5Eat,
    philosopher5Think,
  ];
};

```

```

const philosopher1Pick = new Transition('Pick1');
const philosopher1Put = new Transition('Put1');

const philosopher2Pick = new Transition('Pick2');
const philosopher2Put = new Transition('Put2');

const philosopher3Pick = new Transition('Pick3');
const philosopher3Put = new Transition('Put3');

const philosopher4Pick = new Transition('Pick4');
const philosopher4Put = new Transition('Put4');

const philosopher5Pick = new Transition('Pick5');
const philosopher5Put = new Transition('Put5');

const transitions = [
  philosopher1Pick,
  philosopher1Put,
  philosopher2Pick,
  philosopher2Put,
  philosopher3Pick,
  philosopher3Put,
  philosopher4Pick,
  philosopher4Put,
  philosopher5Pick,
  philosopher5Put,
];

const arcsMap = new ArcsMap();

arcsMap
  .setupTransitionConnections(philosopher1Pick, {
    sources: [
      { place: stick1, multiplicity: 1 },
      { place: stick5, multiplicity: 1 },
      { place: philosopher1Think, multiplicity: 1 },
    ],
    targets: [{ place: philosopher1Eat, multiplicity: 1 }],
  })
  .setupTransitionConnections(philosopher1Put, {
    sources: [{ place: philosopher1Eat, multiplicity: 1 }],
    targets: [
      { place: stick1, multiplicity: 1 },

```

```

        { place: stick5, multiplicity: 1 },
        { place: philosopher1Think, multiplicity: 1 },
    ],
    })
    .setupTransitionConnections(philosopher2Pick, {
        sources: [
            { place: stick2, multiplicity: 1 },
            { place: stick1, multiplicity: 1 },
            { place: philosopher2Think, multiplicity: 1 },
        ],
        targets: [{ place: philosopher2Eat, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher2Put, {
        sources: [{ place: philosopher2Eat, multiplicity: 1 }],
        targets: [
            { place: stick2, multiplicity: 1 },
            { place: stick1, multiplicity: 1 },
            { place: philosopher2Think, multiplicity: 1 },
        ],
    })
    .setupTransitionConnections(philosopher3Pick, {
        sources: [
            { place: stick3, multiplicity: 1 },
            { place: stick2, multiplicity: 1 },
            { place: philosopher3Think, multiplicity: 1 },
        ],
        targets: [{ place: philosopher3Eat, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher3Put, {
        sources: [{ place: philosopher3Eat, multiplicity: 1 }],
        targets: [
            { place: stick3, multiplicity: 1 },
            { place: stick2, multiplicity: 1 },
            { place: philosopher3Think, multiplicity: 1 },
        ],
    })
    .setupTransitionConnections(philosopher4Pick, {
        sources: [
            { place: stick4, multiplicity: 1 },
            { place: stick3, multiplicity: 1 },
            { place: philosopher4Think, multiplicity: 1 },
        ],
        targets: [{ place: philosopher4Eat, multiplicity: 1 }],
    })

```



```

    .setupTransitionConnections(philosopher4Put, {
      sources: [{ place: philosopher4Eat, multiplicity: 1 }],
      targets: [
        { place: stick4, multiplicity: 1 },
        { place: stick3, multiplicity: 1 },
        { place: philosopher4Think, multiplicity: 1 },
      ],
    })
    .setupTransitionConnections(philosopher5Pick, {
      sources: [
        { place: stick5, multiplicity: 1 },
        { place: stick4, multiplicity: 1 },
        { place: philosopher5Think, multiplicity: 1 },
      ],
      targets: [{ place: philosopher5Eat, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher5Put, {
      sources: [{ place: philosopher5Eat, multiplicity: 1 }],
      targets: [
        { place: stick5, multiplicity: 1 },
        { place: stick4, multiplicity: 1 },
        { place: philosopher5Think, multiplicity: 1 },
      ],
    });

const petriNet = new PetriNet(places, transitions, arcsMap);

petriNet.simulate(1000);
petriNet.logResults();
};

export default runPhilosophyTask;

```

runPhilosophyDeadlockTask

```

import { ArcsMap } from './PetriNet/ArcsMap';
import { PetriNet } from './PetriNet/PetriNet';
import { Place } from './PetriNet/Place';
import { Transition } from './PetriNet/Transition';

const runPhilosophyDeadlockTask = () => {
  const stick1 = new Place(1, 'stick1');
  const stick2 = new Place(1, 'stick2');
  const stick3 = new Place(1, 'stick3');
  const stick4 = new Place(1, 'stick4');

```

```

const stick5 = new Place(1, 'stick5');

const philosopher1LeftPicked = new Place(0, 'LeftPicked1');
const philosopher1RightPicked = new Place(0, 'RightPicked1');
const philosopher1Eat = new Place(0, 'Eat1');
const philosopher1Think = new Place(1, 'Think1');

const philosopher2LeftPicked = new Place(0, 'LeftPicked2');
const philosopher2RightPicked = new Place(0, 'RightPicked2');
const philosopher2Eat = new Place(0, 'Eat2');
const philosopher2Think = new Place(1, 'Think2');

const philosopher3LeftPicked = new Place(0, 'LeftPicked3');
const philosopher3RightPicked = new Place(0, 'RightPicked3');
const philosopher3Eat = new Place(0, 'Eat3');
const philosopher3Think = new Place(1, 'Think3');

const philosopher4LeftPicked = new Place(0, 'LeftPicked4');
const philosopher4RightPicked = new Place(0, 'RightPicked4');
const philosopher4Eat = new Place(0, 'Eat4');
const philosopher4Think = new Place(1, 'Think4');

const philosopher5LeftPicked = new Place(0, 'LeftPicked5');
const philosopher5RightPicked = new Place(0, 'RightPicked5');
const philosopher5Eat = new Place(0, 'Eat5');
const philosopher5Think = new Place(1, 'Think5');

const places = [
  stick1,
  stick2,
  stick3,
  stick4,
  stick5,
  philosopher1LeftPicked,
  philosopher1RightPicked,
  philosopher1Eat,
  philosopher1Think,
  philosopher2LeftPicked,
  philosopher2RightPicked,
  philosopher2Eat,
  philosopher2Think,
  philosopher3LeftPicked,
  philosopher3RightPicked,
  philosopher3Eat,

```

```

philosopher3Think,
philosopher4LeftPicked,
philosopher4RightPicked,
philosopher4Eat,
philosopher4Think,
philosopher5LeftPicked,
philosopher5RightPicked,
philosopher5Eat,
philosopher5Think,
];

const philosopher1PickLeft = new Transition('Pick1Left');
const philosopher1PickRight = new Transition('Pick1Right');
const philosopher1StartEating = new Transition('StartEating1');
const philosopher1Put = new Transition('Put1');

const philosopher2PickLeft = new Transition('Pick2Left');
const philosopher2PickRight = new Transition('Pick2Right');
const philosopher2StartEating = new Transition('StartEating2');
const philosopher2Put = new Transition('Put2');

const philosopher3PickLeft = new Transition('Pick3Left');
const philosopher3PickRight = new Transition('Pick3Right');
const philosopher3StartEating = new Transition('StartEating3');
const philosopher3Put = new Transition('Put3');

const philosopher4PickLeft = new Transition('Pick4Left');
const philosopher4PickRight = new Transition('Pick4Right');
const philosopher4StartEating = new Transition('StartEating4');
const philosopher4Put = new Transition('Put4');

const philosopher5PickLeft = new Transition('Pick5Left');
const philosopher5PickRight = new Transition('Pick5Right');
const philosopher5StartEating = new Transition('StartEating5');
const philosopher5Put = new Transition('Put5');

const transitions = [
  philosopher1PickLeft,
  philosopher1PickRight,
  philosopher1StartEating,
  philosopher1Put,
  philosopher2PickLeft,
  philosopher2PickRight,
  philosopher2StartEating,

```

```

philosopher2Put,
philosopher3PickLeft,
philosopher3PickRight,
philosopher3StartEating,
philosopher3Put,
philosopher4PickLeft,
philosopher4PickRight,
philosopher4StartEating,
philosopher4Put,
philosopher5PickLeft,
philosopher5PickRight,
philosopher5StartEating,
philosopher5Put,
];

const arcsMap = new ArcsMap();

arcsMap
  .setupTransitionConnections(philosopher1PickLeft, {
    sources: [{ place: stick1, multiplicity: 1 }],
    targets: [{ place: philosopher1LeftPicked, multiplicity: 1 }],
  })
  .setupTransitionConnections(philosopher1PickRight, {
    sources: [{ place: stick5, multiplicity: 1 }],
    targets: [{ place: philosopher1RightPicked, multiplicity: 1 }],
  })
  .setupTransitionConnections(philosopher1StartEating, {
    sources: [
      { place: philosopher1LeftPicked, multiplicity: 1 },
      { place: philosopher1RightPicked, multiplicity: 1 },
      { place: philosopher1Think, multiplicity: 1 },
    ],
    targets: [{ place: philosopher1Eat, multiplicity: 1 }],
  })
  .setupTransitionConnections(philosopher1Put, {
    sources: [{ place: philosopher1Eat, multiplicity: 1 }],
    targets: [
      { place: stick1, multiplicity: 1 },
      { place: stick5, multiplicity: 1 },
      { place: philosopher1Think, multiplicity: 1 },
    ],
  })
  //
  .setupTransitionConnections(philosopher2PickLeft, {

```

```

        sources: [{ place: stick2, multiplicity: 1 }],
        targets: [{ place: philosopher2LeftPicked, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher1PickRight, {
        sources: [{ place: stick1, multiplicity: 1 }],
        targets: [{ place: philosopher2RightPicked, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher2StartEating, {
        sources: [
            { place: philosopher2LeftPicked, multiplicity: 1 },
            { place: philosopher2RightPicked, multiplicity: 1 },
            { place: philosopher2Think, multiplicity: 1 },
        ],
        targets: [{ place: philosopher2Eat, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher2Put, {
        sources: [{ place: philosopher2Eat, multiplicity: 1 }],
        targets: [
            { place: stick2, multiplicity: 1 },
            { place: stick1, multiplicity: 1 },
            { place: philosopher2Think, multiplicity: 1 },
        ],
    })
    //
    .setupTransitionConnections(philosopher3PickLeft, {
        sources: [{ place: stick3, multiplicity: 1 }],
        targets: [{ place: philosopher3LeftPicked, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher1PickRight, {
        sources: [{ place: stick2, multiplicity: 1 }],
        targets: [{ place: philosopher3RightPicked, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher3StartEating, {
        sources: [
            { place: philosopher3LeftPicked, multiplicity: 1 },
            { place: philosopher3RightPicked, multiplicity: 1 },
            { place: philosopher3Think, multiplicity: 1 },
        ],
        targets: [{ place: philosopher3Eat, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher3Put, {
        sources: [{ place: philosopher3Eat, multiplicity: 1 }],
        targets: [
            { place: stick3, multiplicity: 1 },

```

```

        { place: stick2, multiplicity: 1 },
        { place: philosopher3Think, multiplicity: 1 },
    ],
})
//
.setupTransitionConnections(philosopher4PickLeft, {
    sources: [{ place: stick4, multiplicity: 1 }],
    targets: [{ place: philosopher4LeftPicked, multiplicity: 1 }],
})
.setupTransitionConnections(philosopher1PickRight, {
    sources: [{ place: stick3, multiplicity: 1 }],
    targets: [{ place: philosopher4RightPicked, multiplicity: 1 }],
})
.setupTransitionConnections(philosopher4StartEating, {
    sources: [
        { place: philosopher4LeftPicked, multiplicity: 1 },
        { place: philosopher4RightPicked, multiplicity: 1 },
        { place: philosopher4Think, multiplicity: 1 },
    ],
    targets: [{ place: philosopher4Eat, multiplicity: 1 }],
})
.setupTransitionConnections(philosopher4Put, {
    sources: [{ place: philosopher4Eat, multiplicity: 1 }],
    targets: [
        { place: stick4, multiplicity: 1 },
        { place: stick3, multiplicity: 1 },
        { place: philosopher4Think, multiplicity: 1 },
    ],
})
//
.setupTransitionConnections(philosopher5PickLeft, {
    sources: [{ place: stick5, multiplicity: 1 }],
    targets: [{ place: philosopher5LeftPicked, multiplicity: 1 }],
})
.setupTransitionConnections(philosopher1PickRight, {
    sources: [{ place: stick4, multiplicity: 1 }],
    targets: [{ place: philosopher5RightPicked, multiplicity: 1 }],
})
.setupTransitionConnections(philosopher5StartEating, {
    sources: [
        { place: philosopher5LeftPicked, multiplicity: 1 },
        { place: philosopher5RightPicked, multiplicity: 1 },
        { place: philosopher5Think, multiplicity: 1 },
    ],

```

```

        targets: [{ place: philosopher5Eat, multiplicity: 1 }],
    })
    .setupTransitionConnections(philosopher1Put, {
        sources: [{ place: philosopher5Eat, multiplicity: 1 }],
        targets: [
            { place: stick5, multiplicity: 1 },
            { place: stick4, multiplicity: 1 },
            { place: philosopher5Think, multiplicity: 1 },
        ],
    });

    const petriNet = new PetriNet(places, transitions, arcsMap);

    petriNet.simulate(1000);
    petriNet.logResults();
};

export default runPhilosophyDeadlockTask;

```

index.ts

```

import runPhilosophyDeadlockTask from './runPhilosophyDeadlockTask';
import runPhilosophyTask from './runPhilosophyTask';
import runTestTask from './runTestTask';

// runTestTask();
// runPhilosophyTask();
runPhilosophyDeadlockTask();

```