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 ІІІ курсу, групи *ІП-01 Пашковський Євгеній Сергійович* _____
(прізвище, ім'я, по батькові) (підпис)

Керівник *асистент Дифучин А. Ю.* _____
посада,науковий ступінь,вчене звання,прізвище,і ім'я, по батькові (підпис)

Члени комісії *асистент Дифучин А. Ю.* _____
посада,науковий ступінь,вчене звання,прізвище,і ім'я, по батькові (підпис)

професор, д.т.н., Стеценко І. В. _____
посада,науковий ступінь,вчене звання,прізвище,і ім'я, по батькові (підпис)

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

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. Для спрощення подальшого викладу, будемо використовувати терміни CREATE для процесів створення елементів, PROCESS для процесів обробки та DISPOSE для процесів завершення обробки (зазвичай не використовується в мережах Петрі).

Обробка елементів може відбуватись у режимі з одним або кількома каналами. Багатоканальний режим дозволяє паралельну обробку вимог, що відповідає можливості одночасної обробки декількох вимог. 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). Обробка може відбуватись із визначеною часовою затримкою (у випадку мереж Петрі з часовими затримками), 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 (CREATE), їх переміщення по місцях та переходах (PROCESS), а також дуже рідко у специфічних випадках – видалення токенів після завершення обробки (DISPOSE). 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

2.1. Опис формалізму мереж Петрі

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].

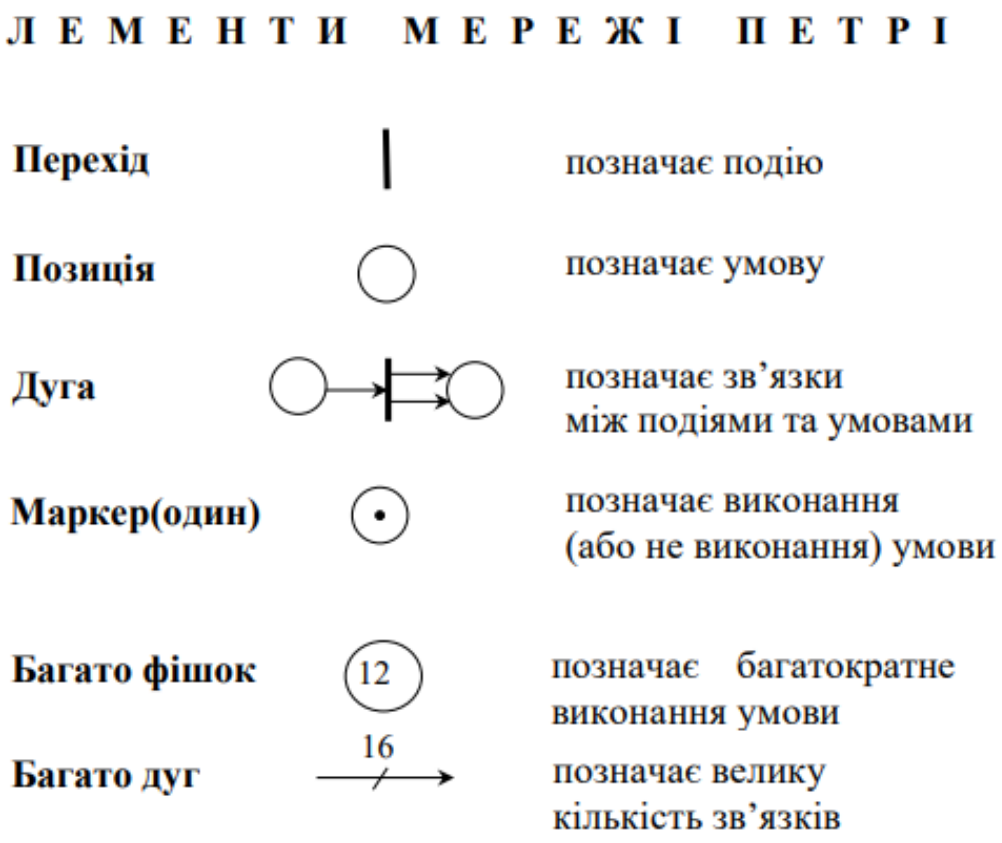


Рисунок 2.1 – Основні елементи мереж Петрі [2]

Для того, щоб представити систему засобами мереж Петрі небхідно [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 передує 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.

2.2. Опис формалізованих моделей

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

2.2.1. 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 розподілятись запити за правилом вирішення конфліктів за замовчуванням. Графічна схема у формалізмі мереж Петрі цієї моделі показана на рисунку 2.2.

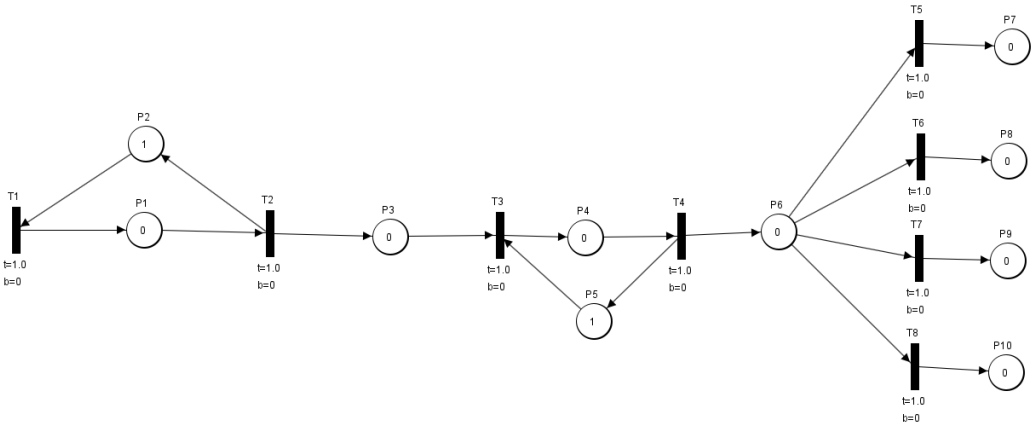


Рисунок 2.2 – Тестова модель для визначення правильності роботи імітаційного алгоритму

2.2.2. 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.

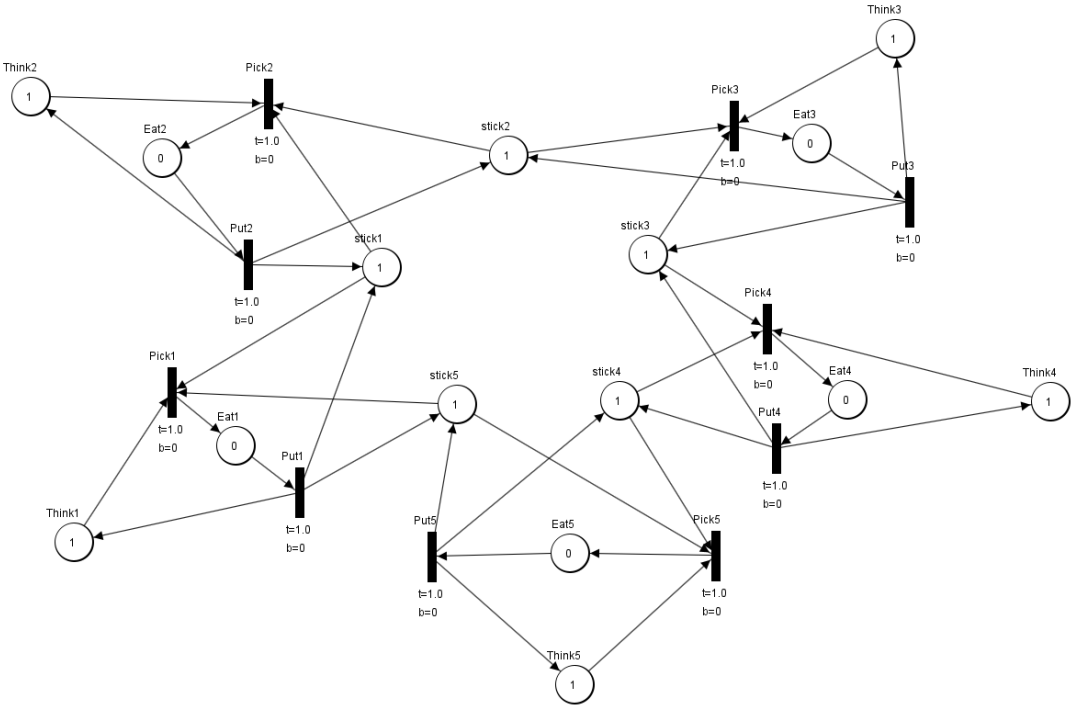


Рисунок 2.3 – Модель для вирішення задачі про філософів

2.2.3. 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 поличці 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.

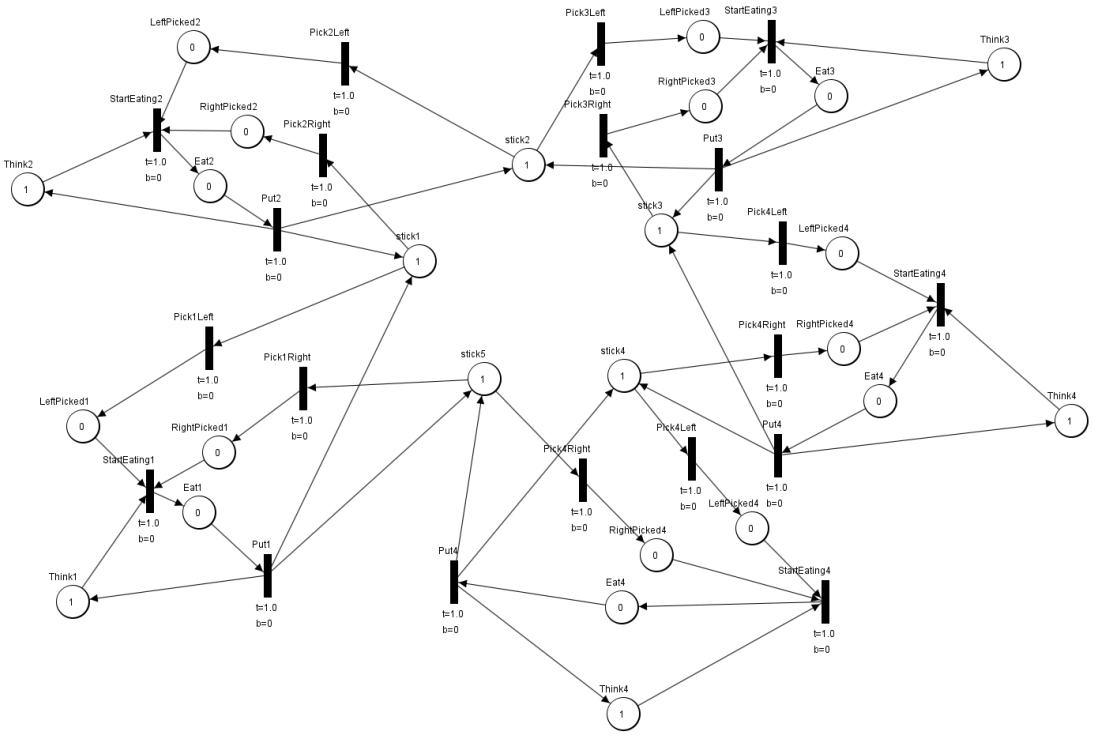


Рисунок 2.4 – Модель для задачі про філософів з явищем дедлоку

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. Кожна така ітерація складається з наступних частин:

– оновлення стану елементів у залежності від поточного часу;

– вихід маркерів з активованих переходів;

– активація переходів з алгоритмом вирішення конфліктів та вхід маркерів;

– формування статистики за результатами ітерації;

– оновлення часу (перехід до наступної ітерації);

– якщо не досягнуте обмеження кількості ітерацій, повторюємо для нового часу (нової ітерації).

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. 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  }       You, 3 дня назад • setup petri net
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);

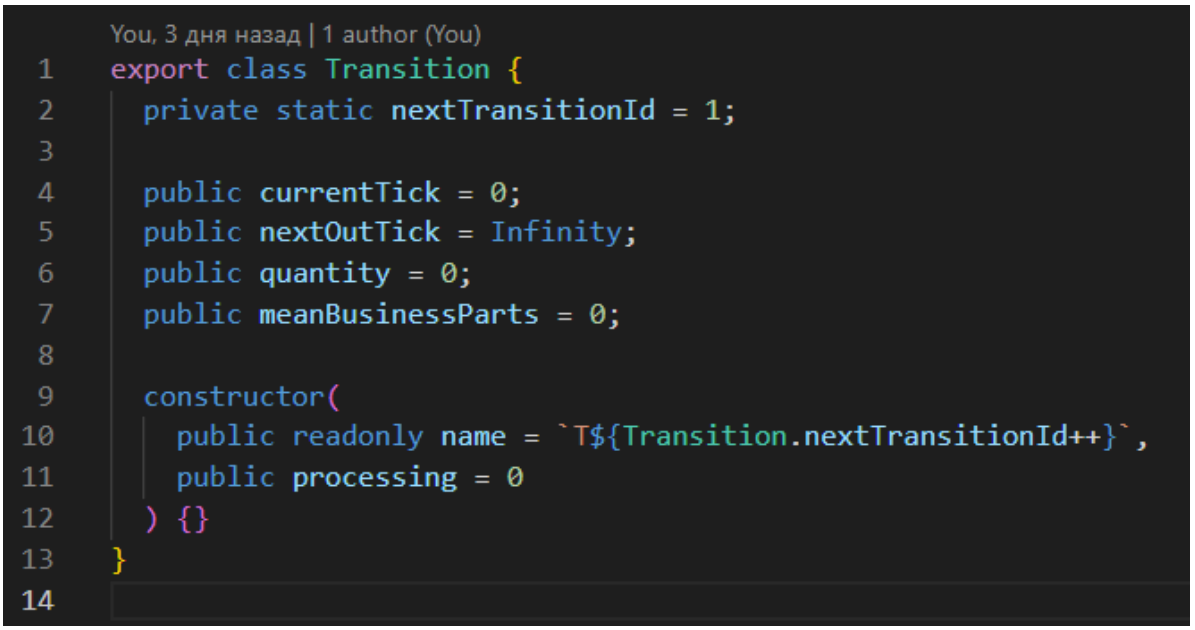


Рисунок 3.3 – Клас переходу

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

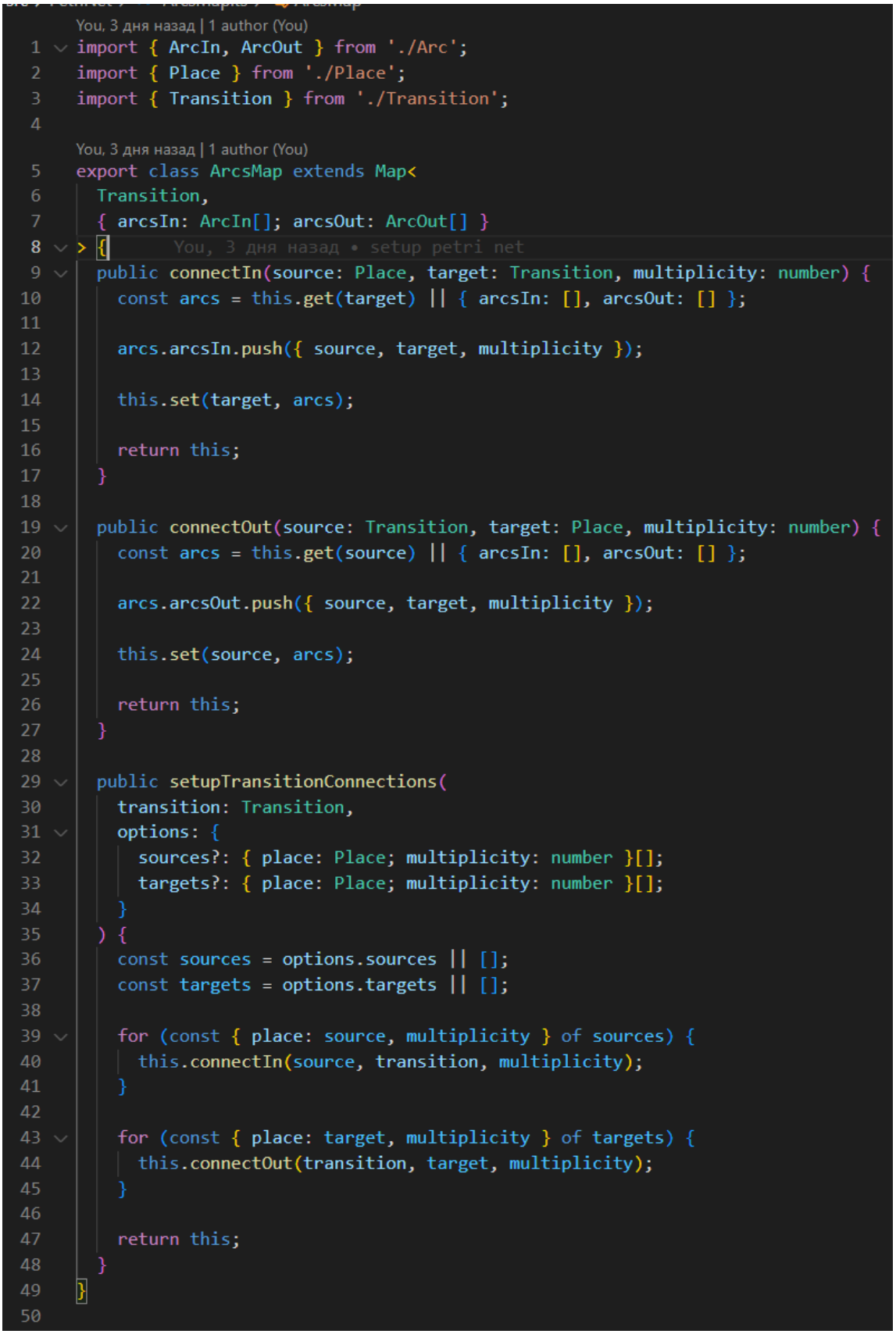


Рисунок 3.4 – Допоміжний клас ArcsMap

– 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          transition.currentTick = this.currentTick;
19
20          const transitionArcs = this.arcs.get(transition);
21          if (!transitionArcs) continue;
22
23          const { arcsOut } = transitionArcs;
24
25          if (
26            transition.nextOutTick === transition.currentTick &&
27            transition.processing > 0
28          ) {
29            for (const arcOut of arcsOut) {
30              arcOut.target.markers += arcOut.multiplicity;       You, 3 дня назад • s
31            }
32            transition.processing--;
33            transition.quantity++;
34          }
35        }
36
37        // IN
38        const transitionsToBeActivated: Transition[] = this.transitions.slice();
39
40        while (transitionsToBeActivated.length !== 0) {
41          const randomIndex = Math.floor(
42            Math.random() * transitionsToBeActivated.length
43          );
44
45          const transition = transitionsToBeActivated.splice(randomIndex, 1)[0];
46
47          const transitionArcs = this.arcs.get(transition);
48          if (!transitionArcs) continue;
49
50          const { arcsIn } = transitionArcs;
51
52          if (
53            !arcsIn.every((arcIn) => arcIn.source.markers >= arcIn.multiplicity)
54          )
55            continue;
56
57          for (const arcIn of arcsIn) {
58            arcIn.source.markers -= arcIn.multiplicity;
59          }
60
61          transition.processing++;
62          transition.nextOutTick = transition.currentTick + 1;
63        }
64
65        this.doStatistics(1);
66        this.currentTick++;
67      }
68    }
```

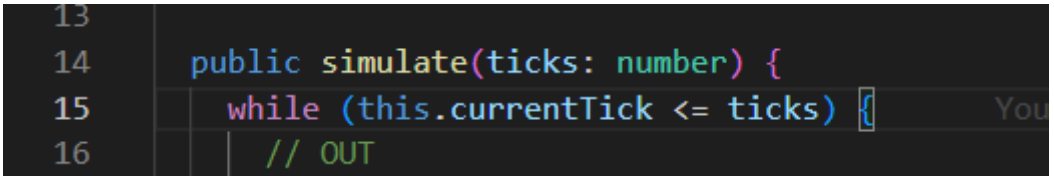
Рисунок 3.5 – Метод simulate класу PetriNet

```
  public doStatistics(delta: number) {
    for (const place of this.places) {
      place.meanValueParts += place.markers / delta;
    }

    for (const transition of this.transitions) {
      transition.meanBusinessParts += transition.processing / delta;
    }
  }

  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}`
      );
    }
  }
}
```

Рисунок 3.6 – Методи doStatistics та logResults класу PetriNet

Тепер давайте детальніше розглянемо реалізацію алгоритму симуляції.

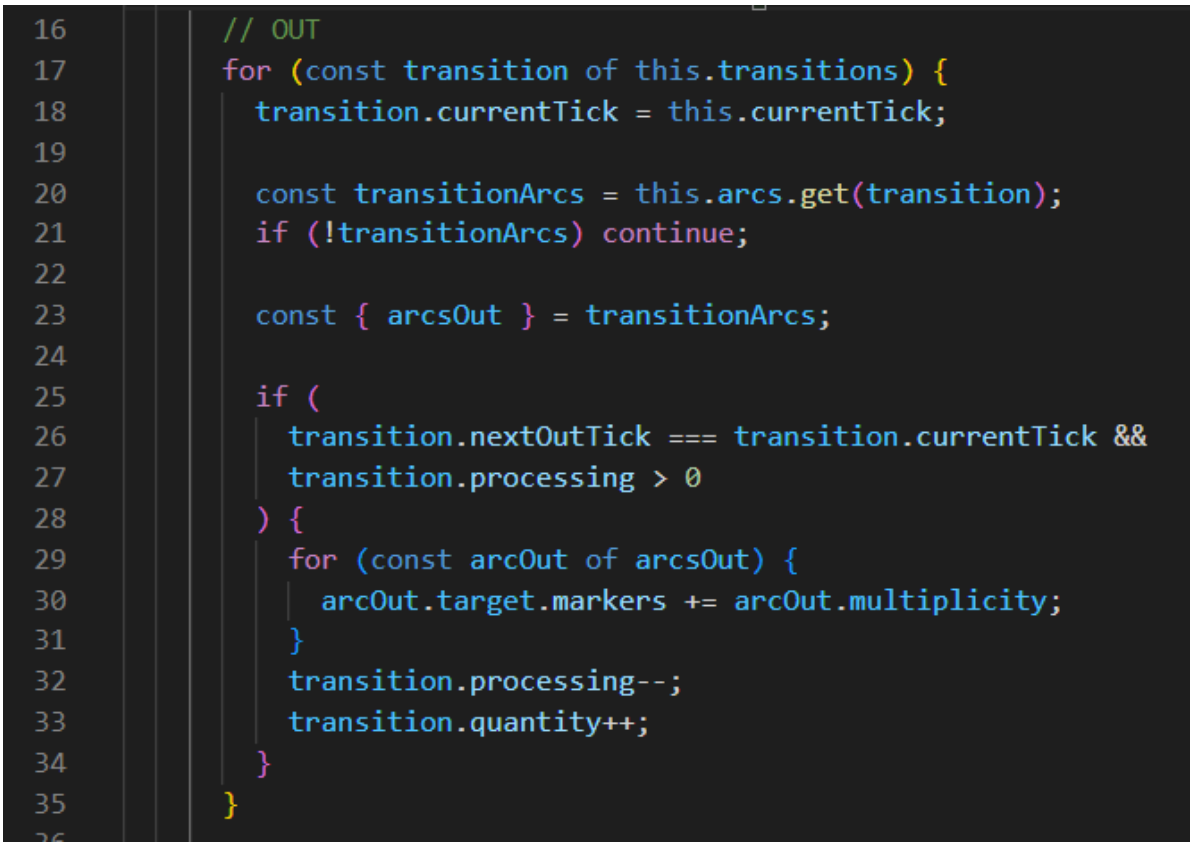
Перш за все, алгоритм починається з обмеження на кількість часу роботи (або ж кількість ітерацій) алгоритму (рисунок 3.7).



```
  public simulate(ticks: number) {
    while (this.currentTick <= ticks) {
      // 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        transition.currentTick = this.currentTick;
19
20        const transitionArcs = this.arcs.get(transition);
21        if (!transitionArcs) continue;
22
23        const { arcsOut } = transitionArcs;
24
25        if (
26          transition.nextOutTick === transition.currentTick &&
27          transition.processing > 0
28        ) {
29          for (const arcOut of arcsOut) {
30            arcOut.target.markers += arcOut.multiplicity;
31          }
32          transition.processing--;
33          transition.quantity++;
34        }
35      }
```

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.

```
37    // IN
38    const transitionsToBeActivated: Transition[] = this.transitions.slice();
39
40    while (transitionsToBeActivated.length !== 0) {
41      const randomIndex = Math.floor(
42        Math.random() * transitionsToBeActivated.length
43      );
44
45      const transition = transitionsToBeActivated.splice(randomIndex, 1)[0];
46
47      const transitionArcs = this.arcs.get(transition);
48      if (!transitionArcs) continue;
49
50      const { arcsIn } = transitionArcs;
51
52      if (
53        !arcsIn.every((arcIn) => arcIn.source.markers >= arcIn.multiplicity)
54      )
55        continue;
56
57      for (const arcIn of arcsIn) {
58        arcIn.source.markers -= arcIn.multiplicity;
59      }
60
61      transition.processing++;
62      transition.nextOutTick = transition.currentTick + 1;
63    }
```

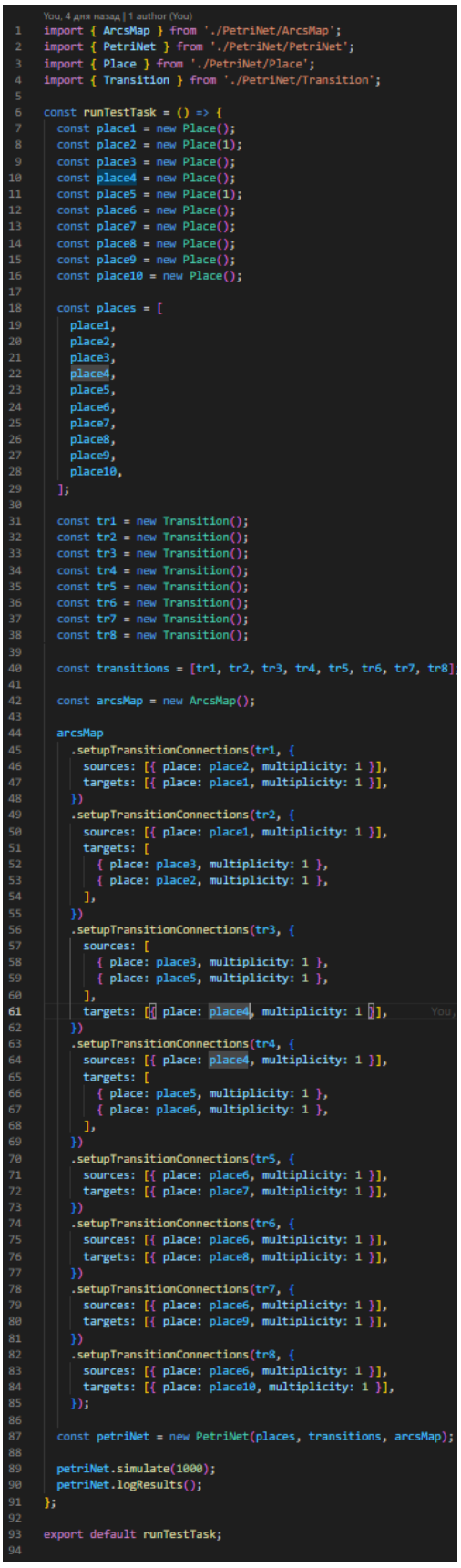
Рисунок 3.9 – Виконання входу маркерів та вирішення конфліктів між переходами

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

```
64
65        this.doStatistics(1);
66        this.currentTick++;
67      }      You, 3 дня назад • setup petri net
68    }
```

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 дня назад | 1 author (You)
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;
```

Рисунок 3.11 – Код для створення та симуляції тестової моделі

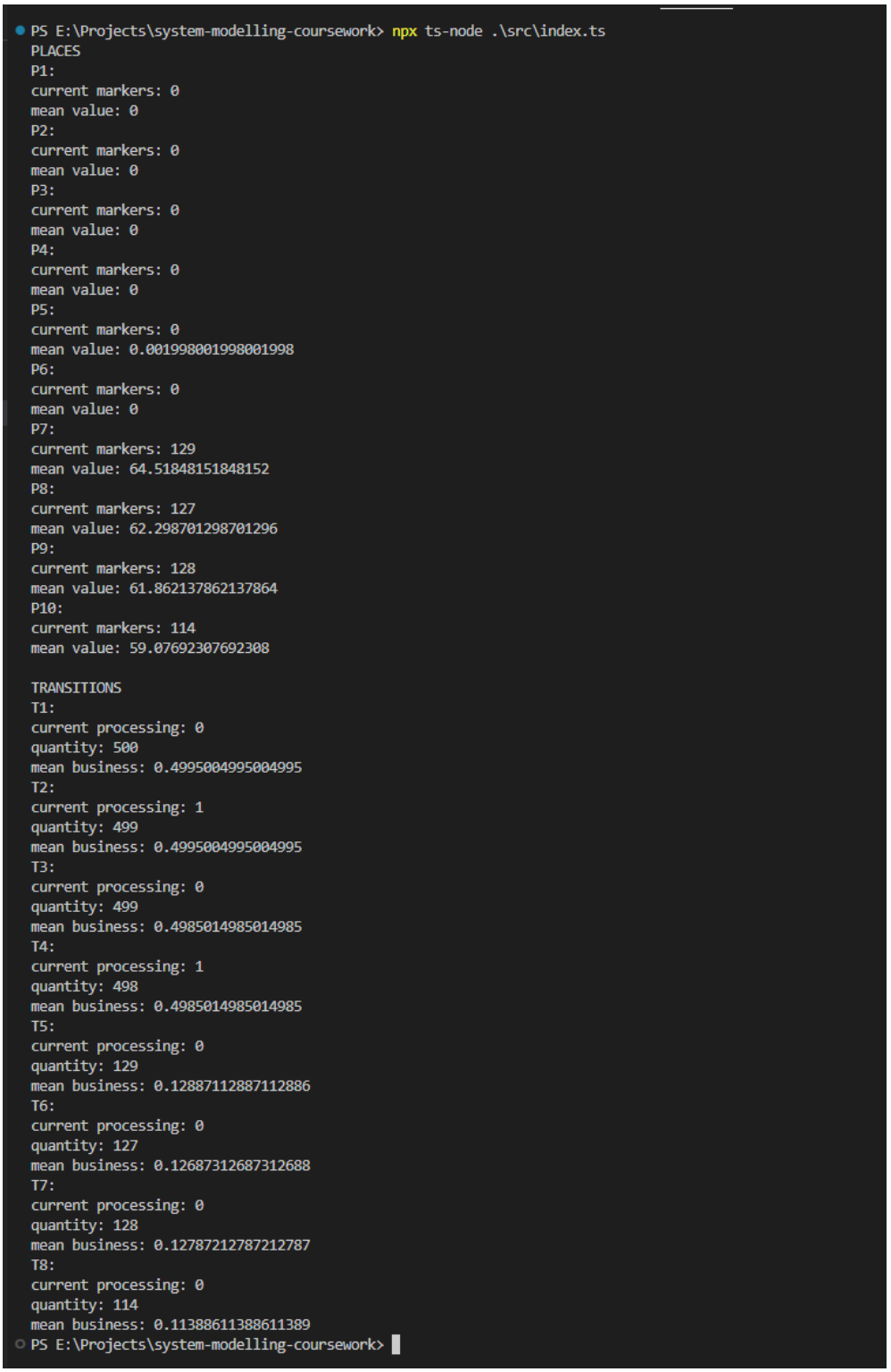


Рисунок 3.12 – Результат запуску тестової моделі

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.

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

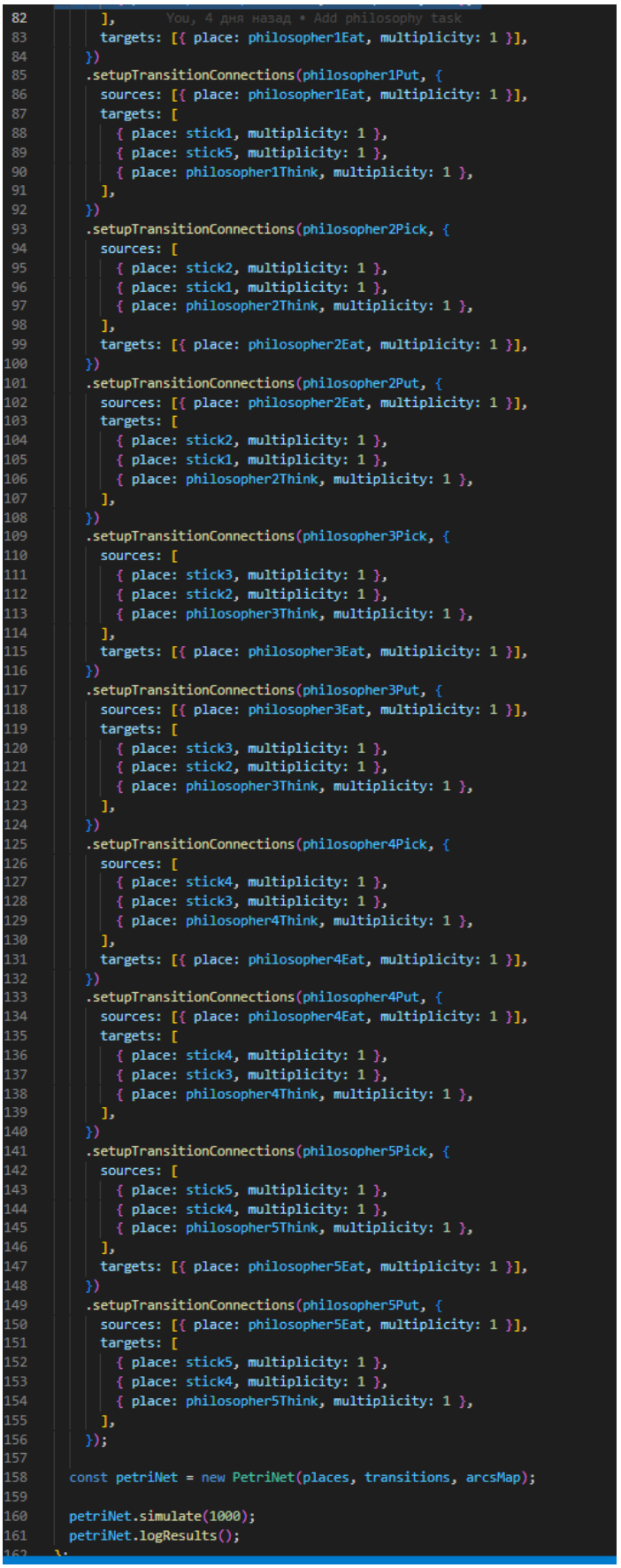


Рисунок 3.14 – Код для створення та симуляції моделі для вирішення задачі про філософів (друга частина)

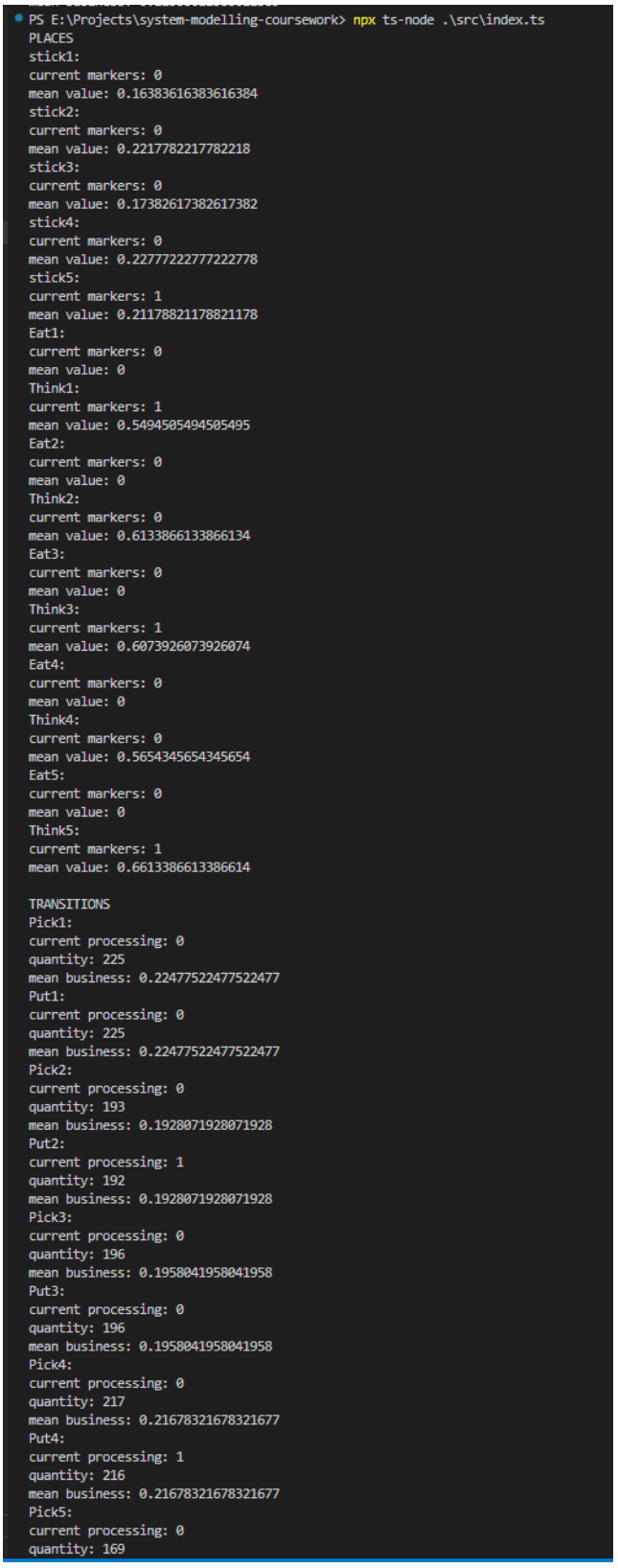


Рисунок 3.15 – Результат запуску моделі для вирішення задачі про філософів

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.

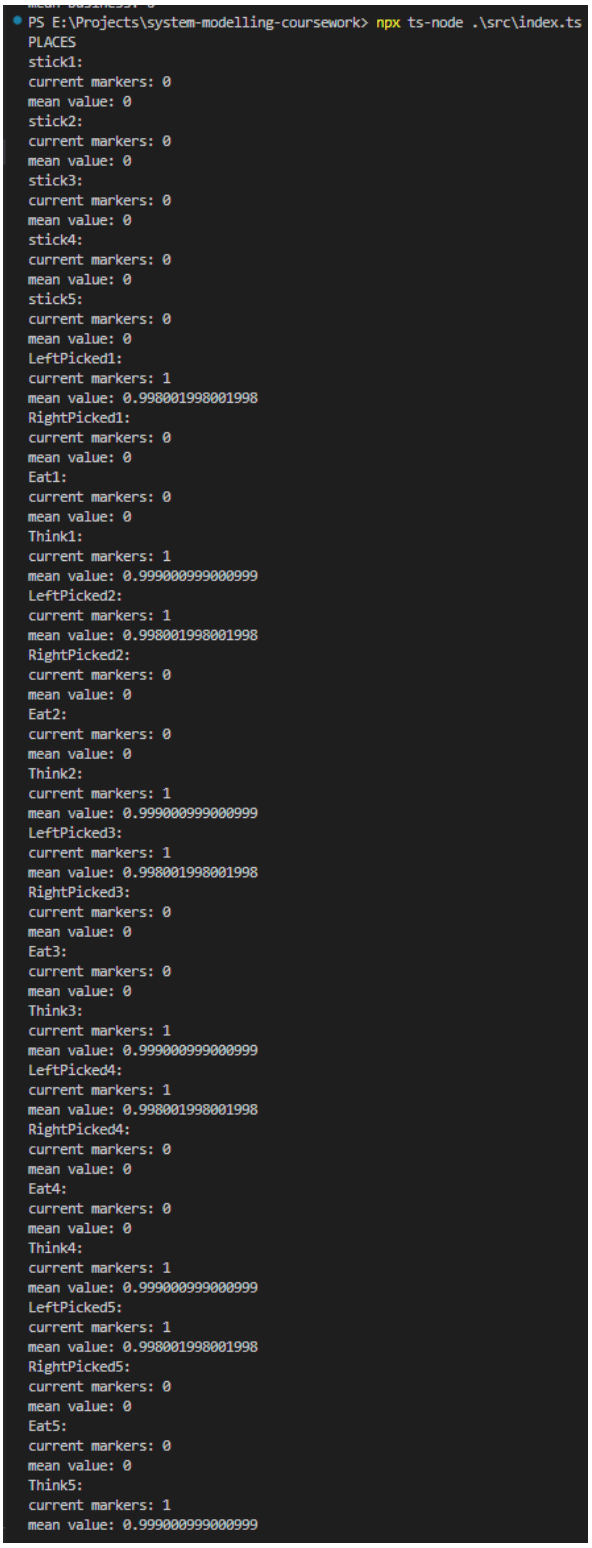


Рисунок 3.16 – Результат запуску моделі для вирішення задачі про філософів з явищем дедлоку

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 близко 0.6, у другому – 0.99, що свідчить про недоступність ресурсу у вигляді обох паличок для їжі через взаємне блокування ресурсів.

3.5. Верифікація моделі

З метою верифікації час симуляції був обмежений 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 параметри										Вихідні параметри									
	Затримка захвату паличок філософом, ітерацій					Затримка звільнення паличок філософом, ітерацій					Час у стані очікування філософа					Кількість з'їдених рисинок				
	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	1	1	1	1	1	1	1	1	1	1	0,6	0,6	0,6	0,6	0,6	2011	1967	2014	1994	1982
2	2	1	1	1	1	1	1	1	1	1	0,49	0,67	0,58	0,58	0,68	1697	1645	2084	2100	1624
3	3	1	1	1	1	1	1	1	1	1	0,44	0,71	0,58	0,57	0,7	1404	1439	2111	2150	1490
4	3	2	1	1	1	1	1	1	1	1	0,5	0,62	0,62	0,56	0,69	1229	1274	1881	2191	1557
5	3	3	1	1	1	1	1	1	1	1	0,54	0,56	0,67	0,56	0,68	1158	1089	1664	2219	1621
6	3	3	2	1	1	1	1	1	1	1	0,52	0,63	0,56	0,62	0,67	1205	929	1457	1887	1657
7	3	3	3	1	1	1	1	1	1	1	0,49	0,68	0,49	0,67	0,67	1270	812	1275	1648	1635
8	3	3	3	2	1	1	1	1	1	1	0,51	0,63	0,58	0,58	0,7	1230	920	1054	1406	1481
9	3	3	3	3	1	1	1	1	1	1	0,5	0,62	0,61	0,52	0,74	1237	946	968	1197	1302
10	3	3	3	3	2	1	1	1	1	1	0,58	0,59	0,62	0,56	0,65	1061	1020	940	1108	1160
11	3	3	3	3	3	1	1	1	1	1	0,62	0,58	0,62	0,59	0,59	942	1061	952	1027	1016
12	3	3	3	3	3	2	1	1	1	1	0,54	0,64	0,6	0,57	0,65	927	908	995	1070	867
13	3	3	3	3	3	3	1	1	1	1	0,5	0,67	0,58	0,59	0,66	828	832	1049	1027	848
14	3	3	3	3	3	3	2	1	1	1	0,53	0,62	0,62	0,58	0,65	779	751	958	1054	879
15	3	3	3	3	3	3	3	1	1	1	0,55	0,59	0,64	0,57	0,65	742	689	906	1084	862
16	3	3	3	3	3	3	3	2	1	1	0,56	0,59	0,6	0,6	0,64	728	679	790	1006	894
17	3	3	3	3	3	3	3	3	1	1	0,54	0,64	0,53	0,65	0,63	765	593	780	868	923
18	3	3	3	3	3	3	3	3	2	1	0,57	0,61	0,58	0,59	0,65	717	645	698	819	885
19	3	3	3	3	3	3	3	3	3	1	0,55	0,62	0,6	0,54	0,69	748	634	660	768	784
20	3	3	3	3	3	3	3	3	3	2	0,6	0,58	0,64	0,55	0,63	682	695	598	744	735
21	3	3	3	3	3	3	3	3	3	3	0,6	0,6	0,6	0,61	0,59	667	667	658	651	689

Рисунок 3.17 – Таблиця верифікації моделі

Враховуючи таблицю верифікації можна зробити наступні висновки:

– при збільшенні будь-якого вхідного параметру, середня кількість з'їдених рисинок знижується;

– при однакових вхідних параметрах, філософи мають час очікування близький 0.6;

– при будь-яких вхідних параметрах середній час очікування філософів дорівнює 0.6;

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

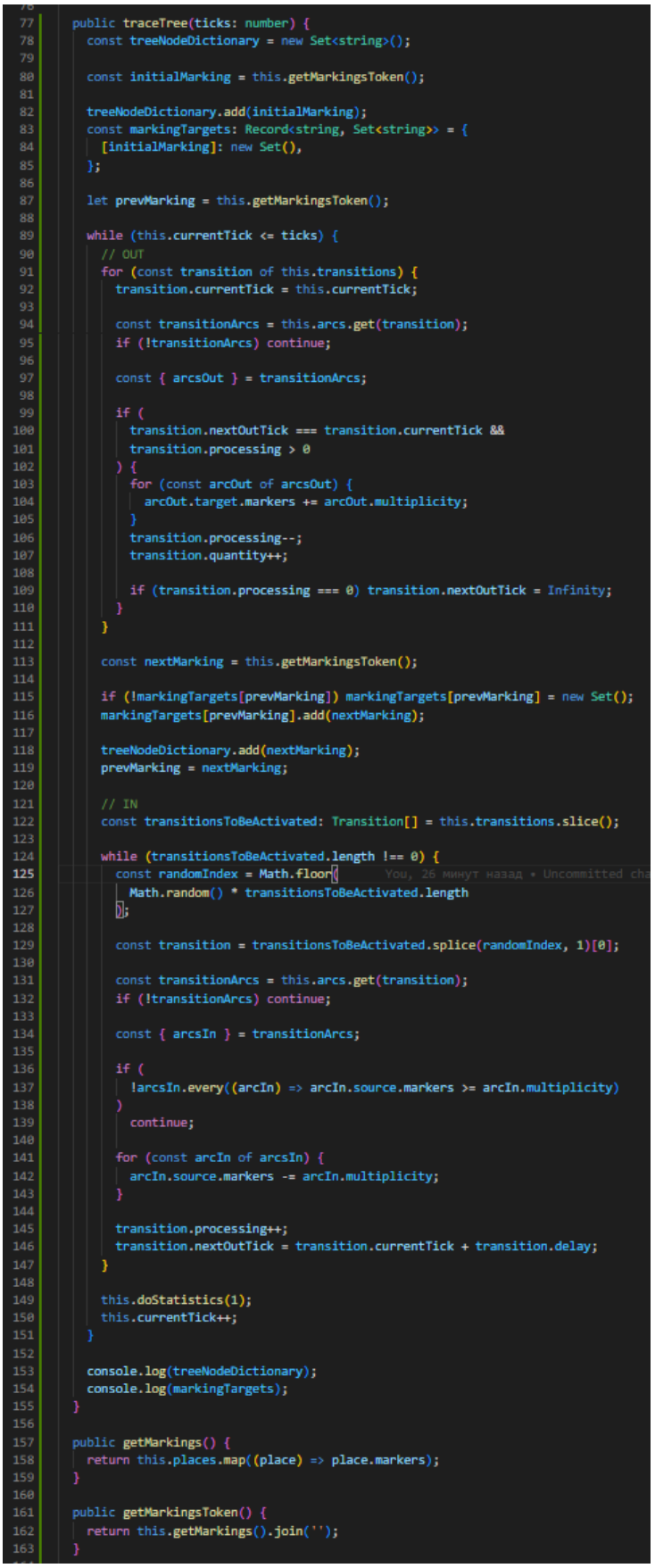
4. ЕКСПЕРИМЕНТИ НА МОДЕЛІ

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 додати до алгоритму імітації часові затримки на переходах. Передусім, це необхідно для визначення вхідних параметрів моделі.

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 токенізується у вигляді рядка такому порядку:

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



```
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) {
      transition.currentTick = this.currentTick;

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

      const { arcsOut } = transitionArcs;

      if (
        transition.nextOutTick === transition.currentTick &&
        transition.processing > 0
      ) {
        for (const arcOut of arcsOut) {
          arcOut.target.markers += arcOut.multiplicity;
        }
        transition.processing--;
        transition.quantity++;

        if (transition.processing === 0) transition.nextOutTick = Infinity;
      }
    }

    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(      You, 26 минут назад • Uncommitted cha
        Math.random() * transitionsToBeActivated.length
      );

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

      const transitionArcs = this.arcs.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++;
      transition.nextOutTick = transition.currentTick + transition.delay;
    }

    this.doStatistics(1);
    this.currentTick++;
  }

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

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

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

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, яке зображено на рисунку 4.3.

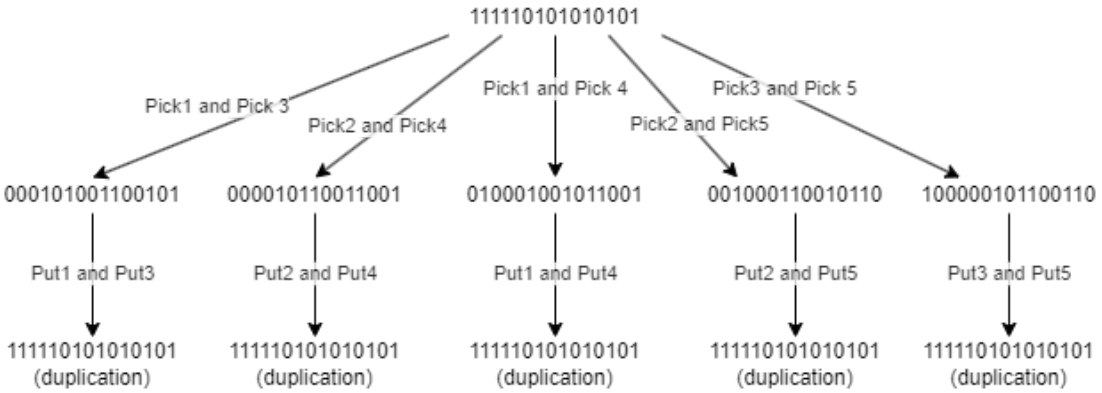


Рисунок 4.3 – Дерево досяжності

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

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

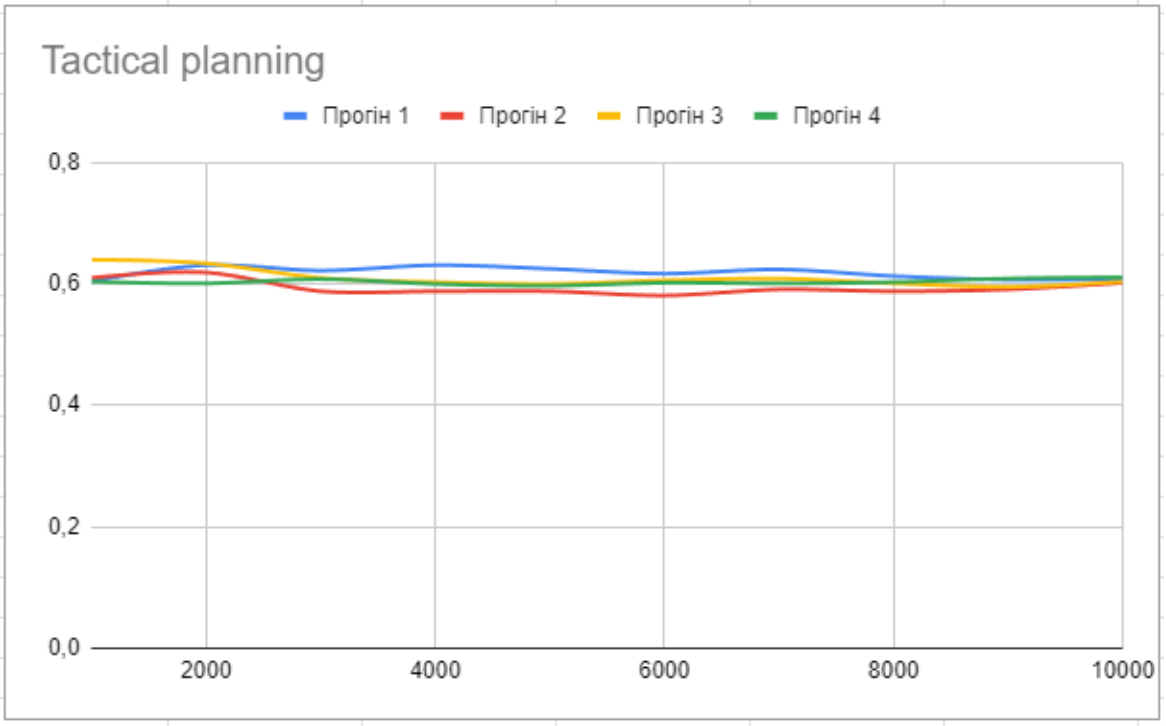


Рисунок 4.5 – Графік для дослідження перехідного періоду

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:

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

$$b_1 = \frac{\sum_{i=1}^{8} y_i x_1}{8} = 0,00625 \quad (4.6)$$

$$b_2 = \frac{\sum_{i=1}^{8} y_i x_2}{8} = 0,02625 \quad (4.7)$$

$$b_3 = \frac{\sum_{i=1}^{8} y_i x_3}{8} = -0,00375 \quad (4.8)$$

$$b_4 = \frac{\sum_{i=1}^{8} y_i x_1 x_2}{8} = 0,02625 \quad (4.9)$$

$$b_5 = \frac{\sum_{i=1}^{8} y_i x_1 x_3}{8} = 0,00625 \quad (4.10)$$

$$b_6 = \frac{\sum_{i=1}^{8} y_i x_2 x_3}{8} = 0,01125 \quad (4.11)$$

$$b_7 = \frac{\sum_{i=1}^{8} y_i x_1 x_2 x_3}{8} = 0,01625 \quad (4.12)$$

Як бачимо, фактори x_1, x_2 та взаємодія факторів x_1x_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:

– при збільшенні будь-якого вхідного параметру, середня кількість з'їдених рисинок знижується;
– при однакових вхідних параметрах, філософи мають час очікування близький 0.6;
– при будь-яких вхідних параметрах середній час очікування філософів дорівнює 0.6;
– зміна параметра одного філософа найбільше впливає на вихідні параметри його самого та його сусідів, на інших філософів така зміна впливає набагато менше.

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), при цьому зберігаючи загальний середній час очікування – 60%.

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. Мережі Петрі [Електронний ресурс] // Wikipedia – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9C%D0%B5%D1%80%D0%B5%D0%B6%D1%96_%D0%9F%D0%B5%D1%82%D1%80%D1%96.

2. Стеценко І. В. Моделювання систем [Електронний ресурс] / І. В. Стеценко. – 2011. – Режим доступу до ресурсу: https://do.ipo.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.

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 arcs: ArcsMap = new ArcsMap()
        ) {}

        public simulate(ticks: number) {
          while (this.currentTick <= ticks) {
            // OUT
            for (const transition of this.transitions) {
              transition.currentTick = this.currentTick;

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

              const { arcsOut } = transitionArcs;

              if (
                transition.nextOutTick === transition.currentTick &&
                transition.processing > 0
              ) {
                for (const arcOut of arcsOut) {
                  arcOut.target.markers += arcOut.multiplicity;
                }
                transition.processing--;
                transition.quantity++;

                    if (transition.processing === 0) transition.nextOutTick =
Infinity;
              }
            }

            // 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.arcs.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++;
                    transition.nextOutTick = transition.currentTick +
transition.delay;
          }

          this.doStatistics(1);

          // 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) {
```

```
            transition.currentTick = this.currentTick;

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

            const { arcsOut } = transitionArcs;

            if (
              transition.nextOutTick === transition.currentTick &&
              transition.processing > 0
            ) {
              for (const arcOut of arcsOut) {
                arcOut.target.markers += arcOut.multiplicity;
              }
              transition.processing--;
              transition.quantity++;

                 if (transition.processing === 0) transition.nextOutTick =
Infinity;
            }
          }

          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.arcs.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++;
                    transition.nextOutTick = transition.currentTick +
transition.delay;
          }

          this.doStatistics(1);
          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(delta: number) {
        for (const place of this.places) {
          place.meanValueParts += place.markers / delta;
        }

        for (const transition of this.transitions) {
          transition.meanBusinessParts += transition.processing / delta;
        }
      }
```

```
    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 currentTick = 0;
    public nextOutTick = Infinity;
```

```
    public quantity = 0;
    public meanBusinessParts = 0;

    constructor(
      public readonly name = `T${Transition.nextTransitionId++}`,
      public processing = 0
    ) {}
  }
```

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();
```