Laboratory 4

Threaded applications in *Java SE* - Testing and implementing Petri nets and Time Petri nets using classic synchronization mechanisms

1. Laboratory objectives

- developing the following skills:
  - understanding of a discrete system described by Petri nets and / or time Petri nets;
  - the ability to design a system described by Petri nets and / or time Petri nets;
  - the ability to implement a system (*Java SE* application) described by Petri nets and / or time Petri nets.

2. Example of an application described by a Petri net

Specifications: Consider the time Petri nets in Figure 4.1. The application that will be implemented in *Java SE*, based on these nets, will use mutual exclusion as synchronization mechanisms for the synchronization element from the P8 place.

The timings of places P6 and P7 correspond to some activities of the form:

```
int k = ...;// random nr. within
      //the specified interval
for (int i = 0; i < k * 100000; i++) {
      i++;
      i--;
}
```

Timed transitions T1 and T2 represent delays and will be implemented by calling the Thread.sleep(x) method.

Requirements: To design and implement in *Java SE* the application in the figure.

Design: The application will be designed by using: state machine diagram, class diagram and sequence diagram. The state machine diagram for the current application is shown in Figure 4.2.

Additional requirements:

- make the class diagram and the sequence diagram for the proposed application;

- modify the source code of the application so that mutual exclusion between the two threads can be tested during run time.

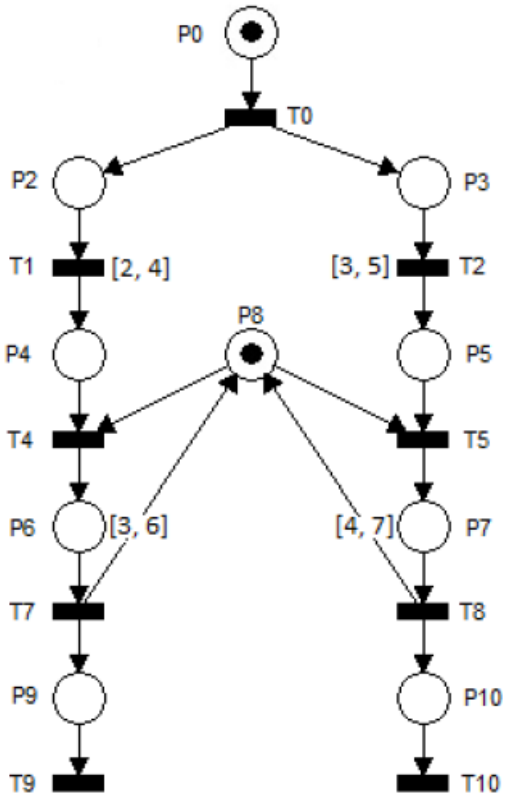

Figure 4.1 Time Petri nets

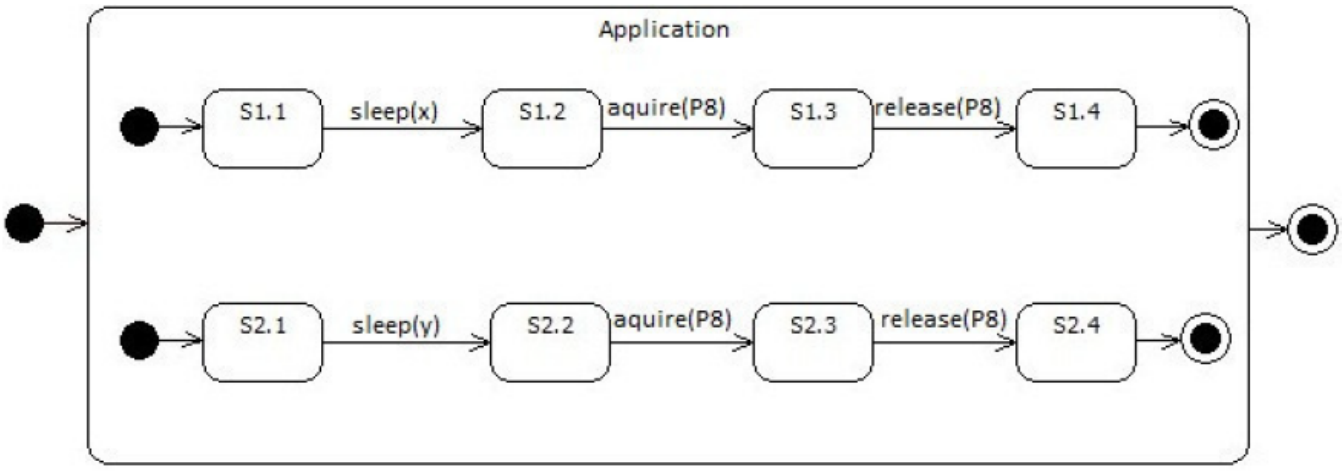

Figure 4.2 State machine diagram

The **implementation** of the application is presented in Code Sequence 1.

Code sequence 1: source code

```
public class ExecutionThread extends Thread {
      Integer monitor;
      int sleep_min, sleep_max, activity_min, activity_max;
      public ExecutionThread(Integer monitor, int sleep_min, int
            sleep_max, int activity_min, int activity_max) {
            this.monitor = monitor;
            this.sleep_min = sleep_min;
            this.sleep_max = sleep_max;
            this.activity_min = activity_min;
```

```
            this.activity_max = activity_max;
      }
      public void run() {
            System.out.println(this.getName() + " - STATE 1");
            try {
                  Thread.sleep(Math.round(Math.random() * (sleep_max
                  - sleep_min)+ sleep_min) * 500);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            System.out.println(this.getName() + " - STATE 2");
            synchronized (monitor) {
                  System.out.println(this.getName() + " - STATE 3");
                  int k = (int) Math.round(Math.random()*(activity_max
                  - activity_min) + activity_min);
                  for (int i = 0; i < k * 100000; i++) {
                        i++; i--;
                  }
            }
            System.out.println(this.getName() + " - STATE 4");
      }
}
public class Main {
      public static void main(String[] args) {
            Integer monitor = new Integer(1);
            new ExecutionThread(monitor, 2, 4, 3, 6).start();
            new ExecutionThread(monitor, 3, 5, 4, 7).start();
      }
}
```

3. Application

Problem statement:

The design and implementation in *Java SE* of the following systems (applications) described by time Petri nets (Figures 4.3, 4.4, 4.5 and 4.6) are required.

For the design phase the system will be described using the following diagrams types:

- state machine diagrams;
- class diagrams;
- communication diagrams between objects.

Conventions:

Timings on places correspond to activities that will be implemented using form of code sequences:

```
int k = ...;//random no. in specific interval
for (int i = 0; i < k * 100000; i++) {
      i++;
      i--;
}
```

Timed transitions are pure delays (which do not load the processor) and will be implemented by using the *Thread.sleep*(*x*) method.

Testing applications: Applications will be implemented so that synchronization between threads can be tested during run time. Tip: Use additional *sleep*() method calls and display text messages in the console.

3.1. Application 1:

The Petri net of Figure 4.3 is given. The *mutual exclusion* will be used for the synchronizaition elements represented by P9 and P10 places.

3.2 Application 2:

The Petri net of Figure 4.4 is given. The *mutual exclusion* will be used for the synchronization elements represented by the P9 and P10 places.

Is there a risk that the network will interlock? If so, modify the net and the implementaion so that there is no interlocking.

3.3 Application 3:

The Petri net of Figure 4.5 is given. The *mutual exclusion* will be used for the synchronization element represented by the P8 place.

3.4 Application 4:

The Petri net of Figure 4.6 is given. The *wait*() / *notify*() methods will be used for T6-P6-T7 and T6-P10-T12 synchronizations. The final synchronization from T11 will be implemented with the *join*() method.

Does the *join*() method work for the final synchronization? Explain.

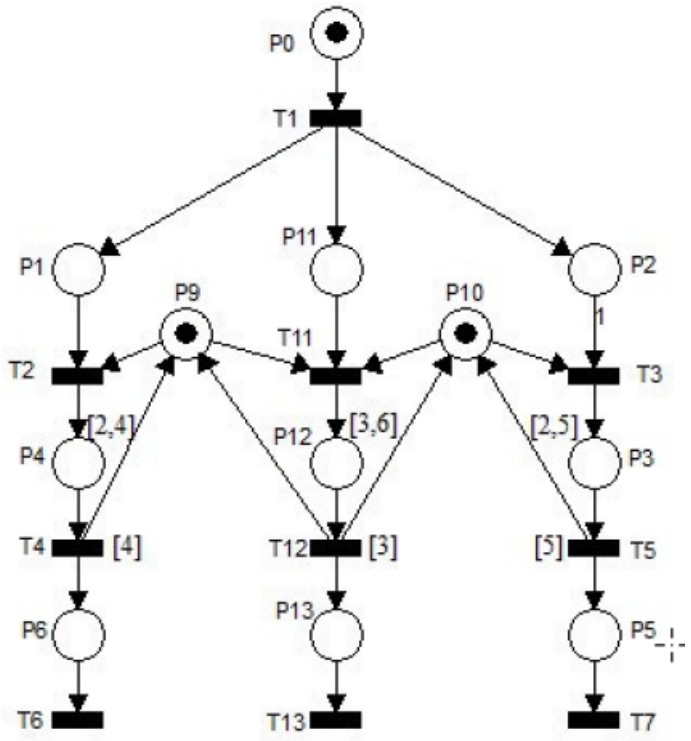

Figure 4.3 Application 1

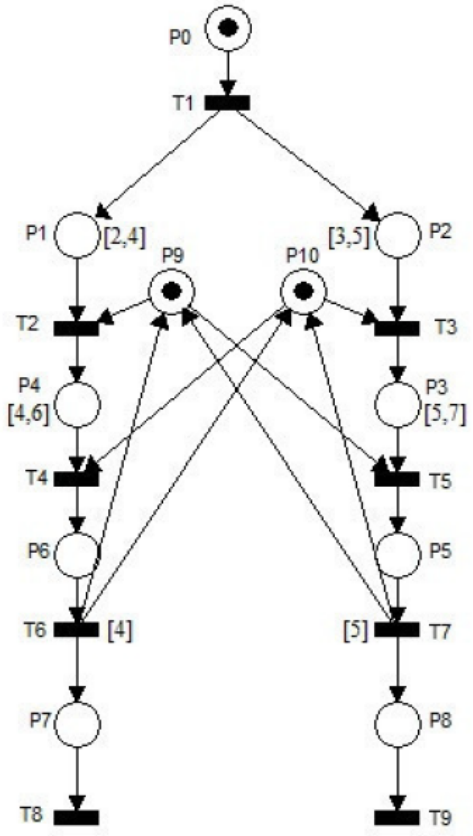

Figure 4.4 Application 2

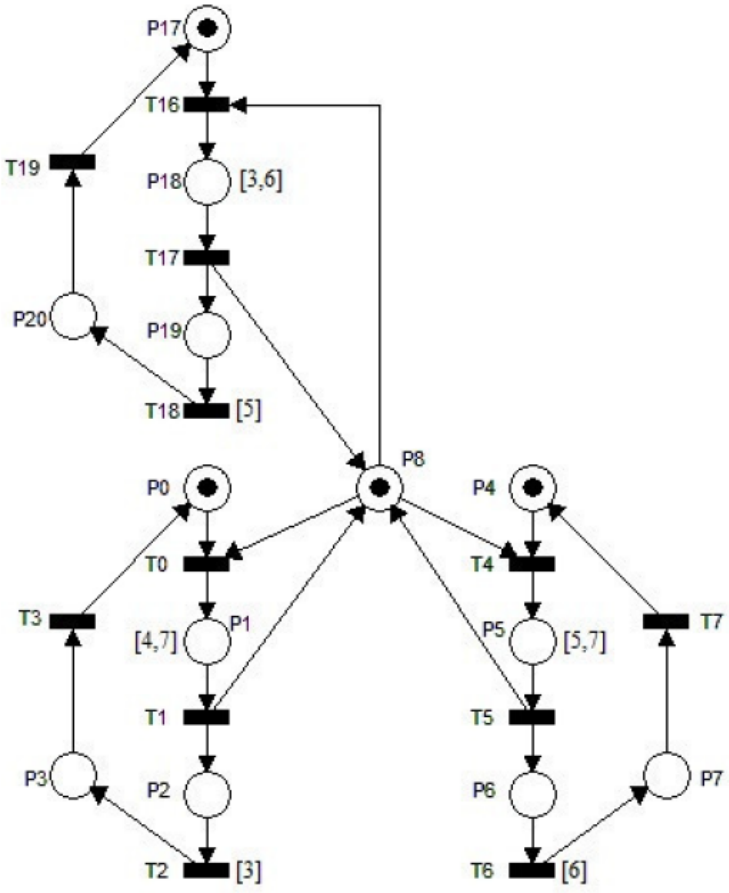

Figure 4.5 Application 3

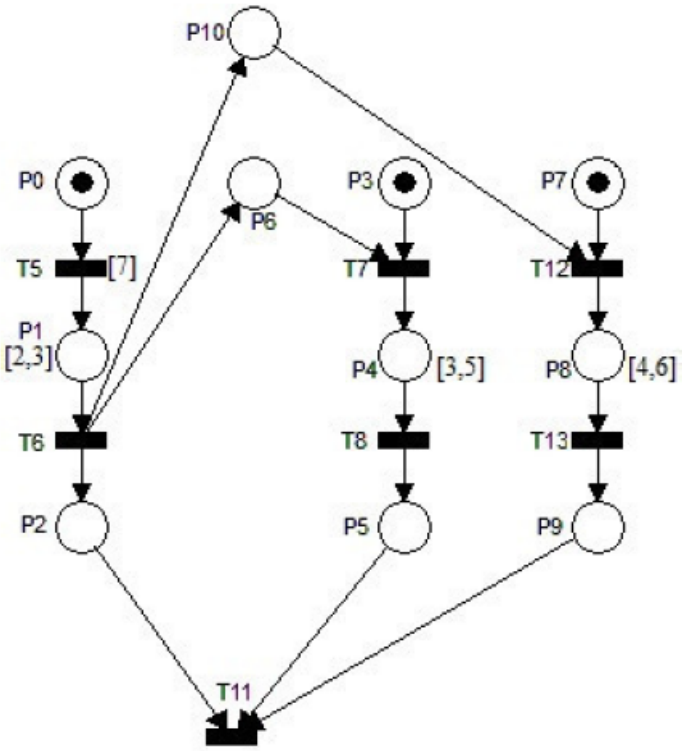

Figure 4.6 Application 4