4 Some application of Petri nets

4.1 Petri net model of Five Dining Philosophers problem

In 1965, Dijkstra posed and solved a synchronization problem called the dining philosophers problem. The problem can be stated as follows. Five philosophers are sitting around a table. Each philosopher has a plate of spaghetti. A philosopher needs two forks to eat it. There is a fork between each couple of plates. The life of a philosopher consists of alternate periods of eating and thinking. When a philosopher gets hungry, he/she tries to acquire his/her left and right forks, one at a time, in either time. If successful in acquiring two forks, a philosopher eats for a while, then puts down the forks and continues to think. The key question is: can you write a program for each philosopher that does what it is supposed to do and never gets stuck? Unfortunately, there may be situation when all five philosophers take their left forks simultaneously.
None will be able to take their right forks, and there will be a deadlock. Deadlock is a situation when all processes would stay blocked forever and no more work would be done.

One of the suggestions is that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. But it is not a good solution, since all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, etc., forever. A situation like this is called starvation. But if the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that everything would continue in lockstep for even an hour is very small.

One way to avoid deadlock and starvation is by using a semaphore type variable MUTEX. Before to acquire forks, a philosopher would do a DOWN on MUTEX. After replacing the forks, she would do an UP on MUTEX. From a practical one, it has a performance bug: only one philosopher can be
eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

![Petri Net model of dining philosophers problem.](image)

**Fig. 4.1** The Petri Net model of dining philosophers problem.

Let us consider the method, which allows the maximum parallelism for arbitrary number of philosophers. It uses an array to keep track of whether a philosopher is eating, thinking or hungry. A philosopher may move into eating
state if neither neighbors eating. Note that each philosopher has two neighbors: left and right. The program uses an array of philosophers, one per philosopher, so hungry philosophers can be blocked if the needed forks are busy.

A Petri net describing the problem is illustrated in Fig.4.1. In this figure places E1, ..., E5 correspond to EATING state of the philosophers. A token in any of these places means relevant philosopher is at EATING state. Similarly, a token in any of the states T1 through T5 indicates that corresponding philosopher is at THINKING state. Places F1 through F5 are used to keep track of presence/absence of the forks. If fork is available then corresponding place must contain a token.

CPN model described above is fully adequate to the nature of Dining Philosophers Problem. The solutions of the problem can be easily obtained by firing relevant transitions of the Petri net.

4.2 Petri net model of Producer-Consumer Problem

Let us consider the producer-consumer problem to see how these two primitives are used. The processes share a common buffer. One of them called a producer writes information into the buffer, and the other one, called consumer, reads
and deletes it out. It is clear that, nothing can be written by the producer if the buffer is full. Similarly, nothing can be read and deleted by consumer is buffer is empty. In the first case the producer must sleep until consumer reads and information from buffer. Respectively, in the second case the consumer must sleep until the producer writes and information into the buffer. In both cases the active process awakes the process slept.

There may be a situation when a wake signal is sent to the process that is not slept yet. If so, the signal will be lost. In such case both processes will sleep forever. E.W.Dijkstra in 1965 suggested using an integer variable to count the number of wake-ups stored for future use. The idea is based on using a new variable type called semaphore. A semaphore could have the value 0, indicating that no wake-ups were saved, or some positive value if one or more wake-ups were sending.

The main points of the problem are summarized below:

- Two processes share a common, fixed-size buffer.
- One of them, the Producer, puts information into the buffer, and the other, the Consumer, takes information out.
- Trouble arises when the Producer wants to put a new item in the buffer, but it is already full.
- The solution is for the Producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the Consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the Producer puts something in the buffer and wakes it up.

Dijkstra proposed having two operations DOWN and UP (generalizations of SLEEP and WAKEUP). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value and just continues. If
the value of 0, the process is put to sleep. The UP operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier DOWN operation, one of them is chosen by the system, and is allowed to complete its DOWN operation.

A Petri net model of the problem when the buffer is unlimited is shown in Fig. 4.2.
4.3 Petri net model of mutual exclusion

In multitasking operating system multiple processes oftenly need to get access to a shared resource such as commonly used file, variable, printer, scanner, etc. If so, a task manager decides whos turn it is to be in critical section (a fragment of the program in which a process requests shared resource). Letting one process to be critical section and forbidding that for another process is called mutual exclusion. Put in other words, mutual exclusion is the case when if one process is using a shared variable, or file, the other processes will be executed from doing the same thing. If we could arrange matters such that no two processes were ever in their critical sections at the same time, we could avoid race conditions.

4.4 Petri net modeling of pipeline computers

Petri nets can be used to model and analyze the functionality of pipeline computers. Simple pipeline computer consists of linearly arranged stages, in which each stage has an input register and an output register. A
stage gets turn to be active and to perform a task if its input register contains data (input register is full) and output register contains no data (output register is empty). A stage sends the results to its output register, if it is empty, then to input register of the next stage, if it is also empty. Finally the next stage gets turn to be active and to perform useful job if its input register contains data. In pipeline computers multiple stages can be simultaneously active and perform tasks each according own logic.

A Petri net in Fig. 4.5 fully adequate to the nature of pipeline computer and illustrates the way it works. At the initial step, stage k-1 starts job if input register k-1 is full (there is a token in related place) and if output
register $k-1$ is empty (if there is a token in related place) As soon as stage $k-1$ completes the job the results of calculations will be sent to output register $k-1$ (a token will be added to corresponding place), then to input register $k$ (a token will be added to corresponding place), then stage $k$ will become active and start performing task, etc.