4 APPLICATIONS OF PETRI NETS

In this section, we study several simple examples that are given to introduce some basic concepts of Petri nets that are useful in modeling.

4.1 Finite state machines

Finite-state machines or their state diagrams can be equivalently represented by a subclass of Petri nets. As an example of a finite-state machine, consider a vending machine, which accepts either nickels or dimes and sells 15-cent or 20-cent candy bars. For simplicity, suppose the vending machine can hold up to 20 cents. Then, the state diagram of the machine can be represented by the Petri net shown in Fig. 23, where the five states are represented by the five places labeled with 0 cent, 5 cent, 10 cent, 15 cent, and 20 cent, and transformations from one state to another state are shown by transitions labeled with input conditions, such as "deposit 5 cent." The initial state is indicated by initially putting a token in the place p_1 with a 0 cent label in this example. Note that each transition in this net has exactly one incoming arc and exactly one outgoing arc. The subclass of Petri nets with this property is known as state machines. Any finite-state machine (or its state diagram) can be modeled with a state machine. The structure of the place p_1, having two (or more) output transitions t_1 and t_2, as shown in Fig. 23, is referred to as a conflict, decision, or choice, depending on applications. State machines allow the representation of decisions, but not the synchronization of parallel activities.

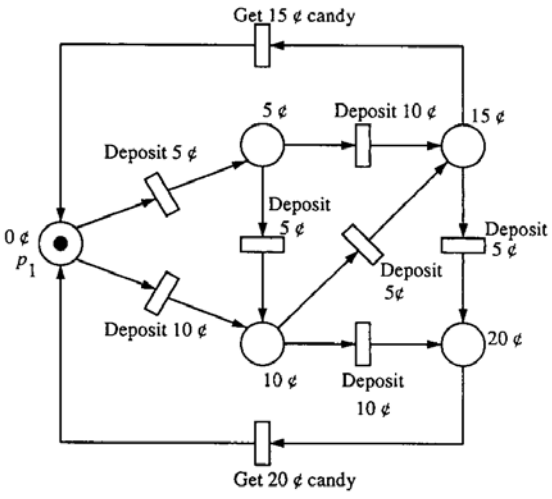

Fig. 23. A Petri net representing of the state diagram of a vending machine, where a coin return transitions are omitted.

4.2 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 spagetti. 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 top see if the right fork is available. If it is not, the philosopher put 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 falling 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 use a semaphore type variable MUTEX. Before to acquire forks, a philosopher would do a DOWN an 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.

Let us consider the method, which allows the maximum parallelism for arbitrary number of philosophers. It uses as array to keep track of whether a philosopher is eating, thinking or hungry. A philosopher may move into eating state if neither neighboris 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.

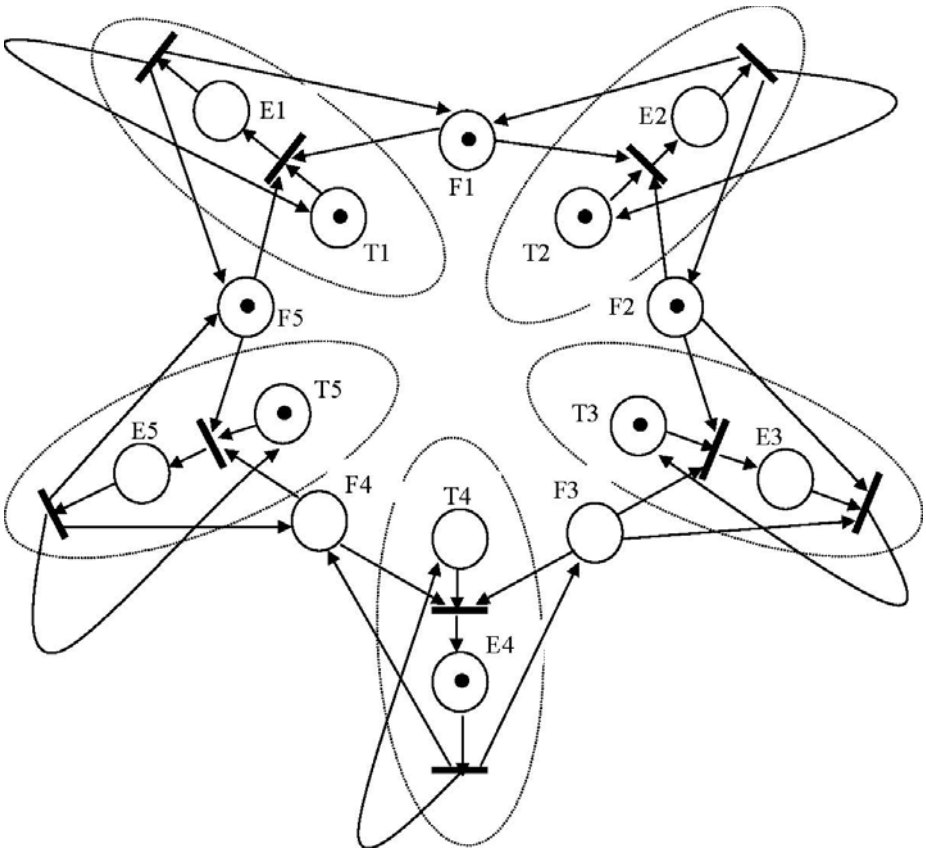

Fig. 24. Petri Net model of dining philosophers problem.

A Petri net modelling the problem is illustated in Fig. 24. In this figure places E1 through E5 correspond to EATING state of the philosophers. A token in any of these places means related 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/absense of the forks. If fork is available then corresponding place must contain a token.

4.3 Producer-Consumer Problem

Let us consider the producer-consumer problem. Two 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.

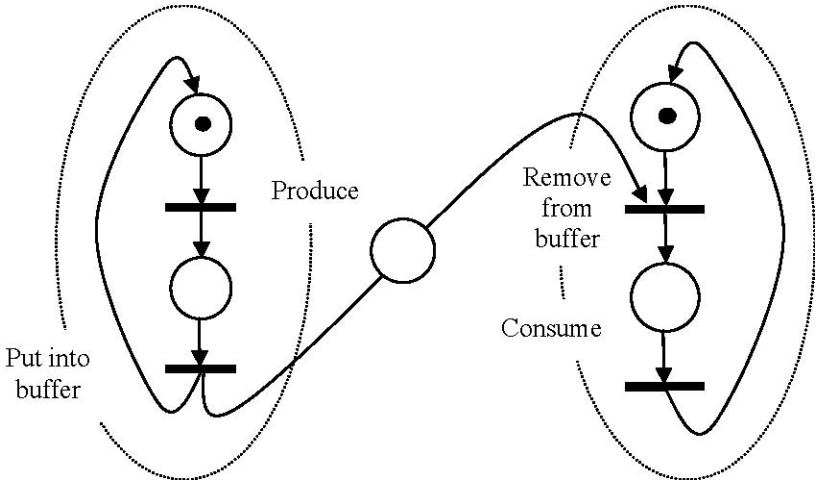

Fig. 25. The Petri Net model of producer and consumer problem with unlimited buffer.

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 thebuffer, 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 beawakened 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 putssomething 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.

This interesting problem can be modelled and analyzed in terms of Petri nets. Correspong Petri net is shown in Fig. 25.

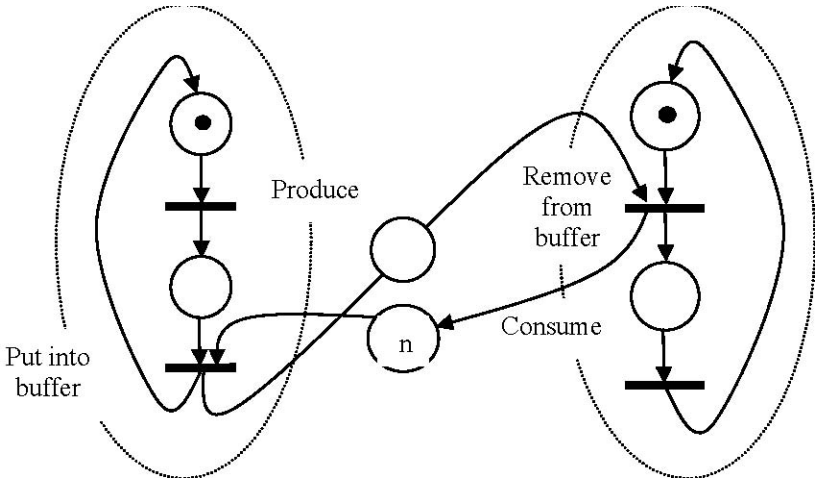

Fig. 26. The Petri Net model of producer and consumer problem with if buffer is limited.

Sometimes it is recommended to restrict the buffer size. Modified Petri net describing the prodcer-consumer problem with restricted buffer is illustrated in Fig. 26.