**DINING-PHILOSPHERS pROLEM AND MULTI-SEMAPHORE SYSTEM MODELING USING Finite Automata**

**Boguslaw Schreyer,** Nipissing University, North Bay, Canada Tel. 705 474 3450 bjs@nipissingu.ca

ABSTRACT (15

From among many tools for concurrent software modelling we have chosen in this paper a classic Finite Automata (FA). Their quite intuitive properties and the graphs appeared to be a very useful tool in Computer Science education. We have been using the FA for years in our Operating Systems I class, mainly in modeling the process synchronization and Critical Section (CS) problems. Now, we want to extend this method for deadlocks and process scheduling simulations. The intent of this paper is to develop an application of the Finite Automata (FA): DFA (Deterministic Finite Automata) and NFA (Nondeterministic Finite Automata) for software modeling in which the binary semaphores are used and a deadlock may occur. First, a classic case of two (or more) concurrent processes and two(or more) binary semaphores is investigated. A deadlock situation is considered. Next a Dining-Philosophers problem is simulated using the FA graphs.

Keywords

Model Checking, Concurrent Systems ,Deadlocks, , Finite Automata, Semaphores.

1. INTRODUCTION

We have introduced the FA for CS problem modelling in our classes as described in [3], [6]. In general terms the subject of software modeling is related to Model Checking [5], and to real time system modelling, validation and verification [9]. We do not, however, restrict our current discussion to the Real Time Operating Systems (RTOS). The existing approaches use the tools such as Timed Automata with Discrete Data [TADD], as in [4], Transition Systems [TS] [10], or Timed Automata (TA), as in [4], [5], [8].

For the majority, the actual implementations are based around the counting (or general) semaphore concept.

We decided however to apply an original Dijkstra’s binary semaphore which is more convenient for graphical representation. We assume that the processes share a very short critical section (CS) code; therefore, we can concentrate on the non-preemptive shared CS. Due to the thread (ni) interleaving, the number M of scheduling sequences for a given number N of processes consisting of atomic sections is “exploding”, according to the following equation:

M=, ni=n1,n2,…,nN

Example: for N=3 processes, and n1=n2=n3= 3 atomic sections in each, M=9!/(6\*6\*6)=1680. In order to concentrate on a general concept, rather than on quantitative issues, we consider limited the number of processes and interleaving threads. However, the number of all cooperating processes in a system, potentially involved in a deadlock, is not limited. We select any two of them for deadlock analysis.

As the semaphores have no concept of an owner, any process can lock/unlock each semaphore. The way the critical sections are implemented varies among operating systems. The busy waiting (BW) state is very common and important. The spinlock semaphores may represent a problem in a multi programming system, where a single CPU is shared among many processes. They have an advantage in that no context switch is required. The spinlocks, if short, may be useful in a multiprocessor system, when applied to the threads, when one thread is spinning on one CPU, while another thread is in a CS on another CPU [1], [2], [7]. At a kernel level, typically, critical sections prevent process and thread migration be- tween processors and the preemption of processes and threads by interrupts. Clear graphic representation of those states, offered in FA application method, is advantageous in system design as well as in programmers communication and in Computer Science education.

1. DFA APPLICATION

Let’s consider a classic example of the two processes (P1 and P2) and two semaphores (Q and S) where a deadlock may occur (Figure 1). Each of the two processes shown below is executed in a loop, basically sequentially, although a sequential execution can be interrupted by context switching and the execution is passed to the other of two processes. Here P is a wait() operation, and V is a signal() operation. The modifications to the semaphore value in wait() and signal() operations are executed indivisibly (atomically).

The idea of using the DFA and NFA for software modeling is not new, but rarely or at all used in the OS texts [1], [2].

\*\*P1\*\* \*\*P2\*\*

P(S) P(Q)

P(Q) P(S)

Critical section CS Critical section CS

V(S) V(Q)

V(Q) V(S)

Figure 1. The two processes P1 and P2

Let be given a deterministic finite automaton (DFA): (R, Σ, δ, q0, F), where

R is a finite set of states,

Σ is a finite alphabet,

δ: R x Σ → R is the transition function,

q0 ϵ R is the start state, and

F ⊆ R is a set of accept states.

Let us to distinguish the following elements of a DFA state: Qϵ{0,1},Sϵ{0,1},CSϵ{0,1}, Bϵ{0,1}. The (binary) semaphores Q and S may be equal either 0 or 1. CS=0 if the CS is not being executed, otherwise CS=1. B=0 of there is no busy waiting (or blocked) process, otherwise B=1. As the changes of the above values can be done only by the semaphore operations, P and V become the members of the alphabet - the edges of the graph. For any state transition, both the current state and the alphabet members result in a new state. Analysis of the sequential execution of a process in a loop defines the transition function δ. This function is defined in the Table 1 below. In Table 1 a Ø means no transition from given state R by a given alphabet member Σ. At the beginning of the P1 execution both semaphore are equal 1: Q=1, S=1, also a CS is not being executed, therefore CS=0, and there is no process busy waiting (or blocked), then B=0. As the process is executed in an infinite loop, there is no finite state. Or, we may assume it is in an initial state, since a whole loop has been executed successfully.

R = {(Q, S,CS,B)}={1100, 1000, 0010, 0100},

Σ = { P(Q), P(S),V(Q),V(S)}

δ is given in the Table 1 below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Σ**  R | P(Q) | P(S) | V(Q) | V(S) |
| 1100 | Ø | 1000 | Ø | Ø |
| 1000 | 0010 | Ø | Ø | Ø |
| 0010 | Ø | Ø | Ø | 0100 |
| 0100 | Ø | Ø | 1100 | Ø |

Table 1. A transition function δ

q0= (1100), and

F = (1100)

A graph of a process P1 from Figure 1 is shown below on Figure 2 (left). A very similar graph of a process P2 is shown on Figure 2 (right). The P2 graph shows different sequence of the P and V execution: first P(Q) and then P(S).

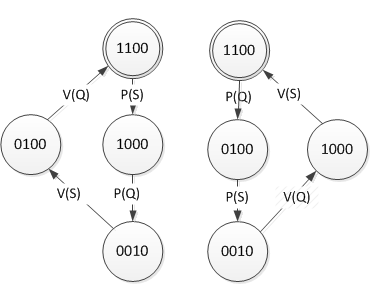


Figure 2. Process P1 (left) and P2 (right)

If the two processes run concurrently, we need to combine those two graphs. We assume that both processes start with initial value of Q=1 and S=1. CS is not being executed therefore CS=0, and there is no process in busy waiting state, i.e. BW=0. Initial state is therefore the same for both processes: q0=1100.

At the same time, we need to distinguish the P and V operations running in those two processes. Now we have 1P(), 1V() operations executing in a process P1, and 2P(), 2V() operations in a process P2. Those are the user’s processes not the OS’ ones. The semaphores are not hardware supported. When a context switch between those two processes takes place, one process is switched to the other’s process code execution; therefore, some of the edges in one graph must now have a number of the other graph.

Assume a process P1 starts the execution and an interrupt happens after P1 has executed its 1P(S). If a P2 executes now 2P(Q), both semaphores Q and S become equal 0 and both processes are prevented from further execution. A deadlock takes place. Similarly, if the sequence of the execution is: 2P(Q) first, then 1P(S). A state 0001 reflects the deadlock. This situation is shown on Figure 6., where no edge is originating in state 0001.

In a state 0010 a process (P1 or P2) is executing in CS , therefore S=Q=0, CS=1.

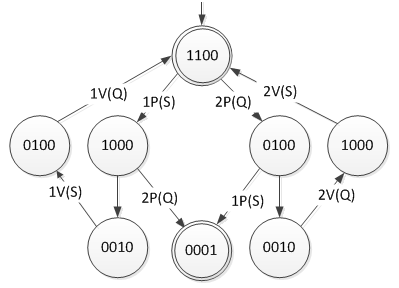


Figure 3. Possible deadlock state 0001

If another process is ready to enter the CS area, it will be blocked and put into a waiting state by (P(S) in P1 or P(Q)in P2). The waiting process will be allowed to CS only if a second V() operation is being executed ( V(Q) in P1 or V(S) in P2 ). Before a process reaches CS, both semaphores are set again to 0.

Therefore, a state becomes again 0010, which is true for both processes. There may be a very short period of time when a process is blocked (P=0 and Q=0), even if another process has already left a CS area (CS=0). The blocked process will be allowed CS after the both Vs are executed. The graph neglects this short moment of time and it’s very short transitory process state 0001

When a process is in a state 0010 (CS=1), another process attempting to execute its code will be blocked (state 0011). After a given process executes both: V(Q) and V(S), it al- lows a blocked process to enter CS (state 0010). This situation is depicted on Figure 4 with edges between states 0010 and 0011.

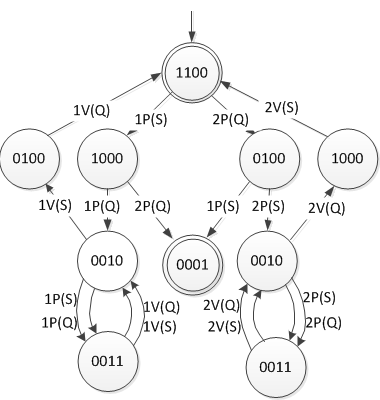


Figure 4. CS=1, the processes are blocked (state 0011)

1. NFA APPLICATION

If there were more than two processes, the waiting queues had more than one process. Also, releasing those processes from the queue would require several executions of V(Q) and V(S). It would modify the graph the way that the loops of V() and P() would appear around the 0011 state as on Figure 5. This modification of a graph introduces a new type of FA,:namely, a nondeterministic finite automaton (NFA):

R is a finite set of states,

Σ is a finite alphabet,

δ: (R x Σ) → Ƥ(R) is the transition function (Ƥ (R) is a power set of R),

q0 ϵ R is the start state, and

F ⊆ R is a set of accept states

For the deadlock to occur, we need a sequence: iP(Q) then jP(S) or: jP(S) then iP(Q) (i,jϵ{1,2} ). As a result both semaphores become equal zero, and creating a deadlock. From the point of view of the above sequence, a number of a process executing does not matter. Therefore we can ignore a process number . Also, a transfer between the two states: 0010 and 0011 does not depend on a process number. Both segments of the graph are symmetric. Again, we can ignore the process number (1 and 2). If we ignore a process number on the edges, we may simplify/generalize the graph and substitute it with a simple combined graph as on **Figure 5**: Here, on an NFA graph the state 0011 and an edge P() or V() may result either in a state 0010 or again in a state 0011 . For this reason, an NFA has been applied. All P() operations around the 0011 state increase a size of the waiting queues, while the V() operations decrease it. Only the last process in a waiting queue allowed to a CS will change the state from 0011 (where B is equal 1) to a new stet 0010 (B is equal 0). It happens by executing a V() operation. A transfer from a state 1000 (or from a state 0100) to a deadlock state is obtained by a sequence: 2P(Q) then 1P(S) or 1P(S) and then 2P(Q).

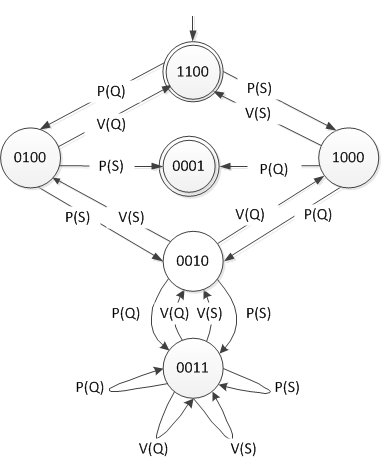


Figure 5. NFA graph combined for two processes.

The simplified graph does not make this distinction and identifies the two process numbers. Similarly, this identification could be extended further to any number of processes, while a graph would remain the same. The above method also applies to the codes with shared CS.

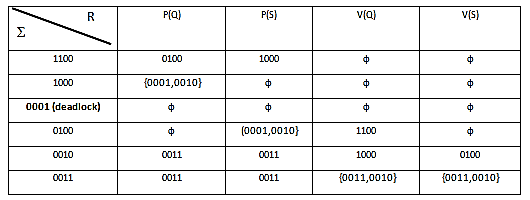


Table 2. NDFA. Transition function δ

1. NFA GRAPH FOR ANY NUMBER OF PROCESSES AND SEMAPHORES.

Similarly to a situation depicted on a Figure 2, let be given a system with m processes and m semaphores. If at certain moment a jP(Sl) is executed, then a iP(Sk), the execution of iP(Sl) cannot be performed. Similarly, if the sequence is opposite: iP(Sk) first, then jP(Sl). Both, sk and sl become zero, and the next P operation ( iP(sl) or jP(sk) respectively) is blocked. Such a situation result from a position of the above operations and semaphores sk and sl.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| \*\*P1\*\* | \*\*P2\*\* | …. | \*\*Pi\*\* | …. | \*\*Pj\*\* | …. | \*\*Pm\*\* |
|  |  |  |  |  |  |  |  |
| 1P(S1) | 2P(S1) | …. | .iP(S1). | …. | jP(S1) | …. | mP(S1). |
| 1P(S2) | 2P(S2) | …. | iP(S2). | …. | jP(S2) | …. | .. |
| ……. | ……. | …. | .. | …. | .. | …. | .. |
| ……. | ……. | …. | iP(Sk). | …. | jP(Sl) | …. | .. |
| 1P(Si) | 2P(Si) | …. | iP(Sl).. | …. | jP(Sk) | …. | .. |
| ……… | ……. | …. | .. | …. | .. | …. | .. |
| 1P(Sj) | 2P(Sj) | …. | .. | …. | . | …. | .. |
| ……. | …….. | …. | .. | …. | .. | …. | .. |
| 1P(Sm) | 2P(Sm) | …. | .. | …. | .. | …. | .. |
| ---- CS | ---- CS | …. | --CS | …. | --CS | …. | --CS |
| 1V(S1) | 2V(S1) | …. | .. | …. | .. | …. | .. |
| 1V(S2) | 2V(S2) | …. | .. | …. | .. | …. | .. |
| …… | …….. | …. | .. | …. | .. | …. | .. |
| 1V(Sm) | 2V(Sm) | …. | iV(Sm) | …. | jV(Sm) | …. | mP(Sm) |

The above

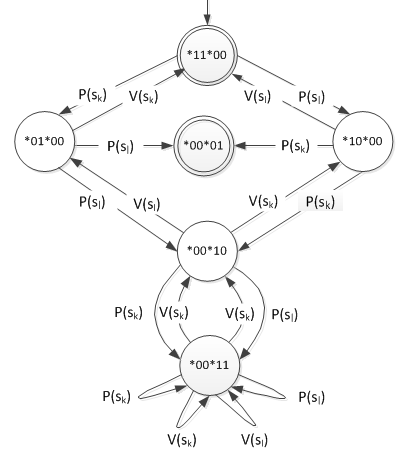


Figure 6. A compound graph for any two processes Pi and Pj, and for any two semaphores sk and sl

4. CONCLUSION

In order to apply the FA in education, a theoretical basis of the method had to be built, and a reference to the Automata Theory had to be specified. Our CSc education program, especially the Operating Systems class has been successfully using this method, i.e. it helped the professors in explaining the OS topics and helped students in better understanding those quite complex phenomena. Application of the FA (DFA and NFA) for the concurrent processes modelling with binary semaphores is relatively simple and makes more detailed models than the SRAGs, especially with respect to sequence of execution. The method is based on generic

finite state automata introduced to the students in the early stages of study, and the method is quite intuitive.

5. REFERENCES

[1]. Silberschatz,A at all, 2012. Operating System Concepts, John Wiley & Sons Inc

[2]. Tanenbaum,S. and Bos,H. 2013. Modern Operating Systems, Pearson.

[3] Schreyer,B. and Kosinski,K.. 2010. Critical resources software and control modeling with finite automata, Theoretical and Applied Informatics, Vol.22(2010), no.3 pp.155-163

[4]. Rataj,A., Wozna,B., and Zbrzezny, A. 2009. A Transla-tor of Java Programs to TADDs, Fundam. Inform.

93(1-3): 305-324 (2009)

[5]. Beier,Ch., and Kaoten,J.2008. Principles of Model Checking, The MIT Press

[6]. Schreyer,B. and Bozic.V. 2006. Deterministic finite automata for critical section modelling. ACM SIGCSE 11th, Bologna, Italy, June 26-28, 2006

[7]. Sipser,M.1997. Introduction to the theory of computation. PWS Publishing Company (1997)

[8]. UPPAAL software ( http://www.uppaal.org/ )

[9]. Lamport,L. 1974. A new solution of Dijkstra’s Con- current Programming Problem. Communications of the ACM 17,8 (August 1974), pp. 453-455

[10]. Arnold,A. FiniteTransition Systems, Prentice Hall,

1994.