

Finite Automata In Software Modeling With Semaphores And Deadlock Potential

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

Krzysztof Kosinski (†) Wyższa
Szkoła Informatyki Stosowanej i
Zarządzania, Warszawa, Poland

ABSTRACT

Finite automata with their graphs are an important tool in our Computer Science education. We have been using them 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 dead-locks 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. A classic case of two (or more) concurrent processes and two binary semaphores is investigated. Processes with a shared critical section (CS) are considered. We anticipate that this method will be used in the Computer Science education program.

General Terms

Computer Education, Operating Systems, Concurrent Systems, Finite Automata, Model Checking, Process Synchronization, Process Scheduling, Deadlocks.

Keywords

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. WCCCE'14, May 2-3, 2014, Richmond, BC, Canada. Copyright 2014 ACM 978-1-4503-2899-9/14/05 ...\$15.00. <http://dx.doi.org/10.1145/2597959.2597961>

However, we decided to apply here 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. (Obviously, if the CS was not shared, use of the process synchronization tools would not be necessary). 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:

$$M = \frac{(\sum_{i=1}^N n_i)!}{\prod_{i=1}^N (n_i!)} , n_i = n_1, n_2, \dots, n_N$$

Example: for N=3 processes and n_i=3 threads in each, M=9!/(6³)=1680. In order to concentrate on a general concept, rather than on quantitative issues, we have limited the numbers of processes and interleaving threads.

The excessive process delays that can result when a process that holds a lock is preempted can be controlled either by making the lock unavailable to a process (that is, by freezing the lock) during the last phase of the process time slice (when the process would not have time to complete its critical section before preemption), or by allowing a lock-holding process to be preempted but then immediately re-scheduled (that is, recovering the process) [1].

As the semaphores have no concept of an owner, any of those two processes can lock/unlock each semaphore. How critical sections are implemented varies among operating systems. Most of the modern real-time operating systems support the semaphore.

A traditional approach to graphical representation/modeling is a system resource – allocation graph (SRAG). On those graphs the semaphores are considered as resources. In our case of using the FA, we have therefore the two resources/semaphores: Q and S. An instance of a resource allocation graph may look as simple as the one the one on **Figure 1** below. However, before the semaphores Q and S have been allocated to the processes P1 and P2, those processes may first claim the resources (one more graph needed), and then they may request those resources (another graph required).

The SRAG looks very simple, but there is a cost for this simplicity. The cost is that we need to draw more graphs to represent thoroughly a sequence of states, potentially leading to a deadlock. Also, on the SRAG there is no clear visual representation of the busy waiting (BW) or blocked state of a process.

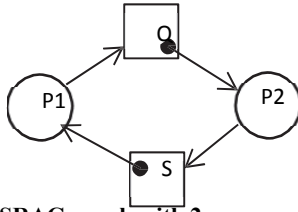


Figure 1. SRAG graph with 2 processes and 2 semaphores

The BW state is very common and important. The spinlock semaphores may represent a problem in a multiprogramming 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 between 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.

2. 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. 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 an await() operation, and V is a signal() operation. The modifications to the semaphore value in wait() and signal() operations must be 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)

Let be given a deterministic finite automaton (DFA): $(R, \Sigma, \delta, q_0, F)$, where
 R is a finite set of states,
 Σ is a finite alphabet,
 $\delta: R \times \Sigma \rightarrow R$ is the transition function, $q_0 \in R$ is the start state, and
 $F \subseteq R$ is a set of accept states.

In case of one process say, P1 let us to distinguish the following values as the elements of a DFA state:
 $Q = \{0,1\}$, $S = \{0,1\}$, $CS = \{0,1\}$, $B = \{0,1\}$. The (binary) semaphores Q and S may be equal either 0 or 1; $CS=0$ if the CS is not being executed, otherwise $CS=1$; $B=0$ if 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 \emptyset means no transition from given state R by a given alphabet member Σ .

At the beginning of the P1 execution both semaphore values 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.

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

$\Sigma = \{P(Q), P(S), V(Q), V(S)\}$

δ is given in the Table 1 below:

R\S	P(Q)	P(S)	V(Q)	V(S)
1100	\emptyset	1000	\emptyset	\emptyset
1000	0010	\emptyset	\emptyset	\emptyset
0010	\emptyset	\emptyset	\emptyset	0100
0100	\emptyset	\emptyset	1100	\emptyset

Table 1. A transition function δ

$q_0 = (1100)$, and

$F = (1100)$

A graph corresponding to the above automata is shown below on Figure 2 (left). A very similar graph depicts a Figure 2 (right). process P2 with different order of the P and V execution: first P(Q) and then P(S). The P2 process is shown on Figure 2 (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: $q_0=1100$.

At the same time, we need to distinguish the P and V operations running in those two processes. Now we have 1P(),1V() operations 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.

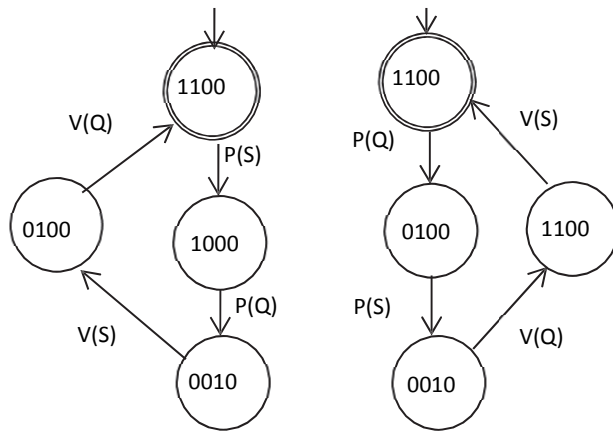


Figure 2. Process P1 (left) and P2 (right)

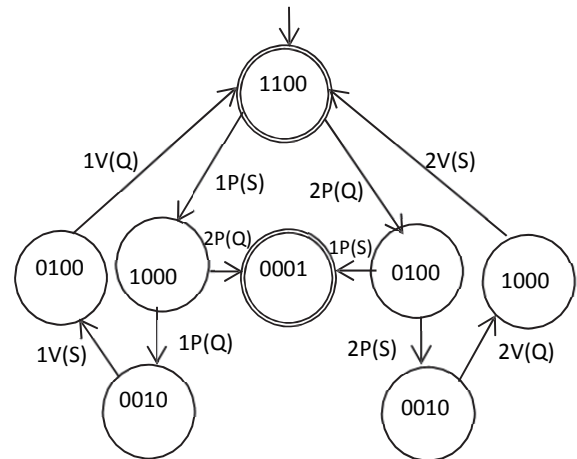


Figure 3. Possible deadlock state 0001

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 $S=Q=0$, $CS=1$. If another process wants 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 V() are executed. The graph neglects this short moment of time and its 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 allows a blocked process to enter CS (state 0010). This situation is depicted on Figure 4 with edges between states 0010 and 0011.

3. 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,

$\delta: (R \times \Sigma) \rightarrow \mathcal{P}(R)$ is the transition function ($\mathcal{P}(R)$ is a power set of R),

$q_0 \in R$ is the start state, and

$F \subseteq R$ is a set of accept states

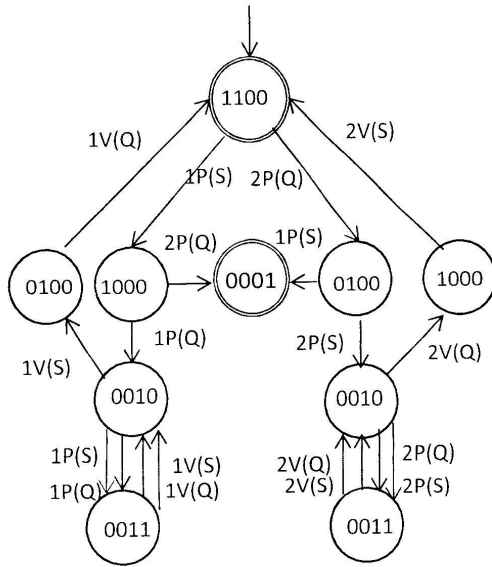


Figure 4. The processes can be also blocked (state 0011)

If we ignore a process/graph 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 (Figure 5). For this reason, an NFA has been applied. Only the last process in a waiting queue (blocked) allowed to a CS will change the state from 0011 (where B is equal 1) to a new state 0010 (B is equal 0). All P() operations around the 0011 state. A transfer from a state 1000 (or from a state 0100) to a deadlock state is obtained by a sequence of P(Q) and P(S) run by the two processes: 1P(S), then 2P(Q) or: 2P(Q) then 1P(S). 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.

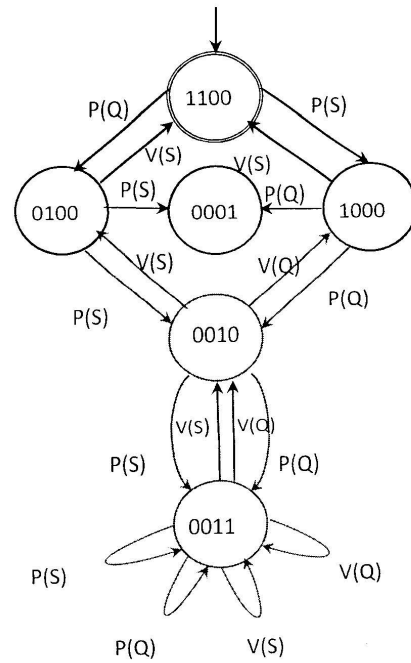


Figure 5. NFA graph combined for two process

Visual representation of the concurrent system with many processes helps students significantly in understanding complex synchronization of the processes and the phenomenon of deadlocks. The popular saying that "one picture is worth a thousand words" is very true in this case.

This teaching method significantly increases student understanding of the timing of the processes execution, sequence of the execution and the waiting queues. Furthermore, it is much easier for students to memorize the overall principle of synchronization and scheduling and understand the concept of semaphores.

A corresponding transition function δ of the above NFA graph is defined in a Table 2.

$\Sigma \backslash R$	P(Q)	P(S)	V(Q)	V(S)
1100	0100	1000	ϕ	ϕ
1000	{0001,0010}	ϕ	ϕ	ϕ
0001 (deadlock)	ϕ	ϕ	ϕ	ϕ
0100	ϕ	{0001,0010}	1100	ϕ
0010	0011	0011	1000	0100
0011	0011	0011	{0011,0010}	{0011,0010}

Table 2. NFA transition function δ

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 Translator 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 Concurrent Programming Problem*. Communications of the ACM 17,8 (August 1974), pp. 453-455
- [10]. Arnold,A. *Finite Transition Systems*, Prentice Hall, 1994.