Zeszyty Naukowe Wydziału Informatycznych Technik Zarządzania
Wyższej Szkoły Informatyki Stosowanej i Zarządzania
„Współczesne Problemy Zarządzania”
Nr 1/2014

FINITE AUTOMATA IN SOFTWARE MODELING WITH SEMAPHORES AND DEADLOCK POTENTIAL

Boguslaw Schreyer*, Krzysztof Kosinski**

*Nipissing University, North Bay,
100 College Dr., Canada

** Wyższa Szkoła Informatyki Stosowanej i Zarzadzania,
01-447 Warszawa, Newelska 6, Poland

Abstract: From among many tools for concurrent software modelling we have chosen in this paper the classic Finite Automata (FA). Their quite intuitive properties and the respective 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 intention 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 concurrent processes and two binary semaphores is investigated. Then, a graph of deadlock for any number of processes and semaphores has been introduced. In order to represent many semaphores on one graph, a compound graph has been created.

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 Schreyer and Kosinski (2010) and Schreyer and Bozic (2006). In general terms, the subject of software modeling is related to Model Checking (Beier and Kaoten, 2008), and to real time system modelling, validation and verification (Lamport, 1974). We do not, however, restrict our current discussion to the Real Time Operating Systems (RTS). The existing approaches use the tools such as Timed Automata with Discrete Data [TADD], as in Rataj, Wozna and Zbrzezny (2009), Transition Systems [TS] (Arnold, 1994), or Timed Automata (TA), as in Rataj, Wozna and Zbrzezny (2009), Beier and Kaoten (2008), or UPPAAL (no date).

In Stotts and Pugh (1994), a class of the parallel finite automata (PFA) have been described and applied. The equivalence of this class to DFA and NFA was demonstrated. A relation between this class of languages and the Petri net languages has been shown. In Dragert, Dingel and Rudie (2008), the discrete-event system (DES) has been applied. In Hirst, Yehetzkael and Mendelbaum (2003) the parallel automata conflicts are discussed. Then, in Stolz (2007), an extension to a concept of assertions, the temporal assertion, has been discussed. An example of a lock-order reversal pattern indicating a potential deadlock in a concurrent program has been given. Finally, in Monzón, Fernández and de la Puente (2011) a software architectural evaluation in the Real Time systems is considered.

In our work we apply Finite Automata, and concentrate on a classic, concurrent Operating System program with possible deadlock (Silberschatz et al., 2012; Tanenbaum and Bos, 2013; Sipser, 1997). We do not consider specifically an RTS. An example of any number of processes and semaphores has been considered with a deadlock between two processes. Keeping in mind an educational application, we tried to generate the useful graphs of the NFA and DFA.

For the majority of cases, the actual implementations are based around the counting (or general) semaphore concept. We decided 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 (n_i) 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:

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

As an example: for N=3 processes, and $n_1=n_2=n_3=3$ 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 a limited 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 multi-

processor system, when applied to the threads, when one thread is spinning on one CPU, while another thread is in a CS on another CPU (Silberschatz et al., 2012; Tanenbaum and Bos, 2013; Sipser, 1997). At a kernel level, typically, critical sections prevent process and thread migration between processors and the preemption of processes and threads by interrupts.

2. DFA application

Let us consider first a classic example of 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 one of the 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 not used in the OS texts (Silberschatz et al., 2012; Tanenbaum and Bos, 2013).

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 considered, P1 and P2

Let be given a deterministic finite automaton (DFA): $(R, \Sigma, \delta, q0, F)$, where

R is a finite set of states,
Σ is a finite alphabet,
$\delta: R \times \Sigma \rightarrow R$ is the transition function,
$q0 \in R$ is the start state, and
$F \subseteq R$ is a set of accept states.

Let us distinguish the following elements of a DFA state: $Q\epsilon\{0,1\}$, $S\epsilon\{0,1\}$, $CS\epsilon\{0,1\}$, $B\epsilon\{0,1\}$. 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 the given state R by a given alphabet member Σ. At the beginning of the P1 execution both semaphores 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), so that 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 in Figure 2 (left). A very similar graph of a process P2 is shown in Figure 2 (right). The P2 graph shows different sequences of the P and V execution: first P(Q) and then P(S).

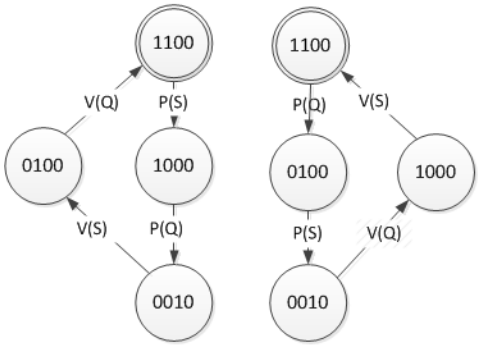

Figure 2. Process P1 (left) and P2 (right)

If the two processes run concurrently, we need to combine those two graphs. We assume that both processes start with the 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 ones of the OS. 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 in Figure 3, where no edge is originating from the state 0001.

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

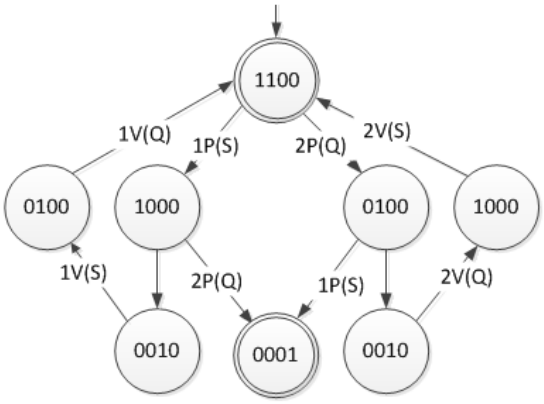

Figure 3. Possible deadlock state 0001

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 is 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 the given process executes both: V(Q) and V(S), it allows a blocked process to enter CS (state 0010). This situation is depicted in Figure 4 with appropriate edges between states 0010 and 0011.

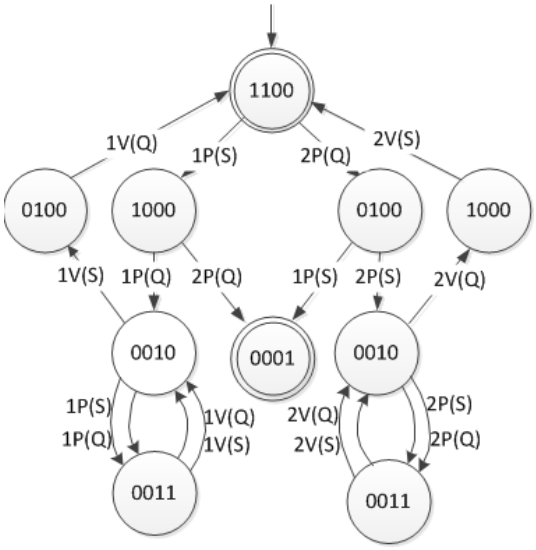

Figure 4. CS=1, the processes are blocked (state 0011)

3. NFA Application

If there were more than two processes, the waiting queues would have more than one process. Also, releasing those processes from the queue would require sev-

eral executions of V(Q) and V(S). This would modify the graph in such a way that the loops of V() and P() would appear around the 0011 state as in 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: $(R \times \Sigma) \rightarrow \mathcal{P}(R)$ is the transition function ($\mathcal{P}(R)$ is a power set of R),

$q0 \in R$ is the start state, and

$F \subseteq R$ 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) ($i,j \in \{1,2\}$). As a result, both semaphores become equal zero, and they create a deadlock. From the point of view of the above sequence, the number of the process executing does not matter. Therefore, we can ignore the process number. Also, a transfer between the two states: 0010 and 0011 does not depend on the process number. Both segments of the graph are symmetric. Again, we can ignore the process number (1 and 2). If we ignore the process number on the edges, we may simplify/generalize the graph and substitute it with a simple combined graph as in 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 the 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 state 0010 (B is equal 0). This happens through execution of a V() operation. A transfer from the state 1000 (or from the state 0100) to a deadlock state is obtained by a sequence: 2P(Q) then 1P(S) or 1P(S) and then 2P(Q).

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.

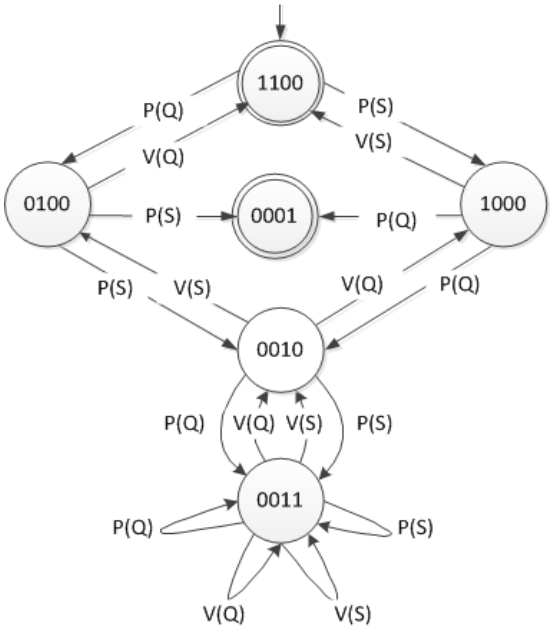

Figure 5. NFA graph combined for two processes

Σ \ 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. NDFA. Transition function δ

4. NFA graph for any number of processes and semaphores.

Similarly to a situation depicted in Figure 2, let be given a system with m processes and m semaphores. All the P and V operations are identical, except for the semaphores. A prefix k in kP() is the number of a process, in which an operation appears. A double index in a semaphore S(k,l) specifies a process Pl and a line of code k, where a semaphore appears. Two semaphores, which have different indexes (k,l), do not have to h. In most of cases, the semaphores in the same line of code k and in a different processes Pn and Pm are the same: S(k,n)=S(k,m). Altogether there are m semaphores and m processes.

In order to create a potential deadlock situation, we need the two semaphores, say $S_{m,i}$ and $S_{m-1,I}$ in one process, say Pj to change their order with respect to another process, say Pi. This is why the indexing of the two operations **iP($S_{m-1,i}$)** and **iP($S_{m,i}$)** and the indexing of the semaphores do not follow any of the above rules. The semaphore indexes: (m,i) and (m-1,i) specify only that the semaphores are equal to the semaphores in a process Pi, in the lines of code respectively m-1 and m (Figure 6).

If at a certain moment a iP(Sm-1,i) is executed in a process I, then a jP(Sm-1,i) is executed in a process j.

Both semaphores: Sm-1,i and Sm, I become equal 0 and none of those two processes can execute in CS. It is a deadlock situation.

P1	**P2**	…	**Pi**	…	**Pj**	…	**Pm**
1P($S_{1,1}$)	2P($S_{1,2}$)	…	.iP($S_{1,i}$).	…	jP($S_{1,j}$)	…	mP($S_{1,m}$).
1P($S_{2,1}$)	2P($S_{2,2}$)	…	iP($S_{2,i}$).	…	jP($S_{2,j}$)	…	..
...	…	…	…	…	....	…	..
....	…	…	...	…		…	..
		...	...	...	….	...	
1P($S_{i,1}$)	2P(Si,2)	…		…		…	..
...	….	…	…	…	....	…	..
1P($S_{j,1}$)	2P(Sj,2)	...	…	...	...	...	..
...	…	…	**iP(Sm-1,i)**	**…**	**jP(Sm,i)**	…	..
1P($S_{m,1}$)	2P(Sm,2)	…	**iP(Sm,i)**	**…**	**jP(Sm-1,i)**	…	..
------- CS	------- CS	…	-----CS	…	-----CS	…	-----CS
1V($S_{1,1}$)	2V($S_{1,2}$)	…	…	…	....	…	mV($S_{1,m}$)
1V($S_{2,1}$)	2V($S_{2,2}$)	…	…	…	....	…	..
…	…	...	iV($S_{m-1,i}$)	...	jP($S_{m,i}$)	...	..
1V($S_{m,1}$)	2V($S_{m,2}$)	…	iV($S_{m,i}$)	…	jV($S_{m-1,j}$)	…	mP($S_{m,m}$)

Figure 6. m processes and m semaphores with deadlock potential

The above situation is depicted with a graph in Figure 7.

In an initial state *11* 00 all semaphores are equal 1, CS=0 and B=0,

- an edge $iP(s_{1,i})$ represents a P operation executed in a process i, on a semaphore $S_{1,}$
- an edge $jP(s_{1,j})$ represents a P operation executed in a process j, on a semaphore $S_{1,}$
- an edge $*P(S_{k,l})$ represents a P operation in either process i or j, on any semaphore, according to a scheduler,
- similar to the above is the indexing of the V operations.

In a state *….* 00 some semaphores become equal 0, as the process i and a process j advance in execution. A semaphore $S_{k,l}$ represents any semaphore, where k,l ϵ (1,2,….,m-1,m). In a state 0…01 00 all semaphores but S_m are equal 0, while in a state 0…10 00 all semaphores but S_{m-1} are equal 0. The state 0…0 01 represents a deadlock between processes i and j. If any of those processes is executing in CS, any new process request will put this process on a waiting list. In a state 0…0 10 one process is in CS, while in a state 0…0 11 a new process is blocked. The bold symbols and thick edges show the **P** operations causing directly a deadlock of the two processes.

5. Conclusion

The FA method, useful in education, has been introduced. Our Computer Science education program, especially the Operating Systems class, has been quite successfully using this method, i.e. it helped the professors in explaining the OS topics and helped students in better understanding of the concurrent software execution with deadlocks. Application of the FA (DFA and NFA) for the concurrent processes modelling with binary semaphores is relatively simple. A simple compound graph for many semaphores has been created.

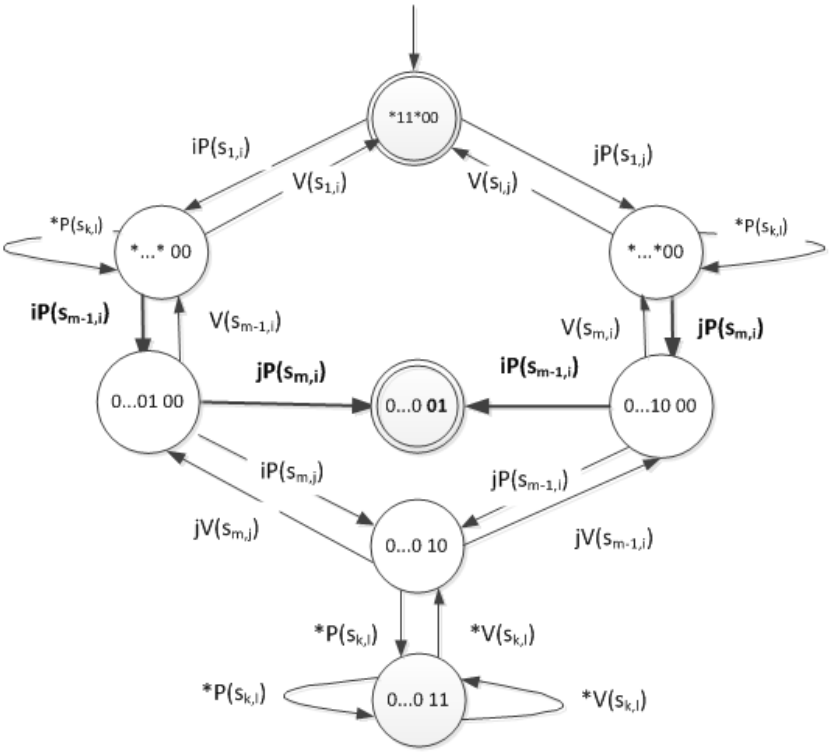

Figure 7. A compound graph for any two processes Pi and Pj and any m semaphores

References

Arnold, A. (1994) *Finite Transition Systems*. Prentice Hall.

Beier, Ch. and Kaoten, J. (2008) *Principles of Model Checking*. The MIT Press.

Dragert C., Dingel J. and Rudie K. (2008) Generation of Concurrency Control Code using Discrete-Event Systems Theory. *Proceeding, SIGSOFT '08/FSE-16 Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering.*

Hirst T., Yehezkael R.B. (formerly Haskell), Mendelbaum H.G. (2003) *Some Theoretical Results on Parallel Automata, Conflict, Complexity*. JCT Research Report.

Lamport, L. (1974) A new solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM* 17,8 (August), pp. 453-455.

Monzón A., Fernández J.L. and de la Puente J.A. (2011) Application of Deadlock Risk Evaluation of Architectural Models. *Journal of Software Practice and Experience* Article first published online: 2 SEP 2011 DOI: 10.1002/spe.1118.

Rataj, A., Wozna, B. and Zbrzezny, A. (2009) A Translator of Java Programs to TADDs. *Fundam Inform*., **93**(1-3), pp. 305-324.

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

Schreyer, B. and Kosinski, K. (2010) Critical resources software and control modeling with finite automata. *Theoretical and Applied Informatics* **22**(3), pp.155-163

Silberschatz, A., et al. (2012) *Operating System Concepts*. John Wiley & Sons Inc.

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

Stolz V. (2007) Temporal Assertions for Sequential and Concurrent Programs. Thesis, RWTH Aachen University, AIB-2007-15, August, http://aib.informatik.rwth-15.pdf.

Stotts P.D. , Pugh W. (1994) Parallel Finite Automata for Modeling Concurrent Software Systems, *Journal of Systems and Software*.

Tanenbaum, S. and Bos, H. (2013) *Modern Operating Systems*. Pearson.

UPPAAL software (http://www.uppaal.org/)