Mateusz Machalica April 28, 2014 r.

Analysis of Szymanski's algorithm in Spin

Evaluated properties

During the course of this brief analysis of Szymanski's mutual exclusion algorithm we will repeatedly refer to the following properties. All quantifiers are over the set of process identifiers and constructs of the form `P[i]@label` mean that process i is about to execute instruction labelled with `label`. Names of the labels should be understood intuitively, they also match names of the labels present in provided models. Note that pairs of operators $\forall$ and $\Box$ as well as $\exists$ and $\Diamond$ are commutative, therefore all properties have an expected invariant form $\Box(\ldots)$.

Mutual exclusion

$$\Box\, \forall_i \forall_j (i = j \vee \neg \texttt{P[i]@critical\_section} \vee \neg \texttt{P[j]@critical\_section}) \tag{1}$$

Inevitable waiting room entry

$$\neg(\exists_i \Diamond (\texttt{P[i]@started\_protocol}\, \mathcal{U}\, (\neg(\texttt{we[i] \&\& !chce[i]})\, \mathcal{U}\, \texttt{P[i]@critical\_section}))) \tag{2}$$

Waiting room exit

$$\Box\, \forall_i (\exists_j (\texttt{we[i] \&\& !chce[i]} \implies i \neq j \wedge \Diamond\, \texttt{wy[j]})) \tag{3}$$

Liveness

$$\Box\, \forall_i (\texttt{P[i]@started\_protocol} \implies \Diamond\, \texttt{P[i]@critical\_section}) \tag{4}$$

Linear wait

The algorithm is symmetric with respect to all pairs of processes of the form $i < j$, that means if for one such pair number of times one of the processes can overtake the other is not bounded by a constant, then for all such pairs the bound does not hold and we have no linear waiting. Therefore linear waiting condition is equivalent (for this particular algorithm) to bounded overtaking condition. The latter is possible to express in LTL if we assume appropriate constant that bounds the number of overtakings that can happen between any two processes. For our purposes we will use two formulas that approximate desired condition – existence of such constant.

If property (5) holds then we know that overtaking is bounded by two.

$$\begin{aligned} \Box\, (\forall_i \forall_j ((i \neq j \wedge\ & \texttt{P[i]@started\_protocol}) \implies \\ & \neg\texttt{P[j]@critical\_section}\, \mathcal{U}\, (\texttt{P[j]@critical\_section}\, \mathcal{U}\, (\\ & \neg\texttt{P[j]@critical\_section}\, \mathcal{U}\, (\texttt{P[j]@critical\_section}\, \mathcal{U}\, (\\ & \neg\texttt{P[j]@critical\_section}\, \mathcal{U}\, \texttt{P[i]@critical\_section})))))) \end{aligned} \tag{5}$$

On the other hand if there exists a counterexample to property (6), then we have found a computation such that process i starts protocol but never enters critical section, while process j enters and leaves critical section infinitely many times, which means that waiting time cannot be bounded by linear function of the number of processes.

$$\begin{aligned} \Box\, (\forall_i \forall_j ((i \neq j \wedge\ & \texttt{P[i]@started\_protocol}) \implies \neg(\\ & \Box\, \neg\texttt{P[i]@critical\_section} \\ & \wedge \Box\, \Diamond\, \texttt{P[j]@critical\_section} \\ & \wedge \Box\, \Diamond\, \neg\texttt{P[j]@critical\_section}))) \end{aligned} \tag{6}$$

Waiting room exit – improved definition

Note that property (3) no longer reflects our intentions when we start permuting assignment statements in the protocol's epilogue as described in the next section. For this reason we have proposed more accurate (based on process' instruction pointer) definition of the state described as a *waiting room* in Szymanski's original paper [1]. Improved definition is being used in the LTL formulation of property (7).

$$\Box \forall_i (\exists_j (\texttt{P[i]@in\_anteroom} \implies i \neq j \wedge \Diamond \texttt{wy[j]})) \tag{7}$$

All experiments were conducted with *weak fairness* enabled. Even though for some properties this assumption can be weakened, it is crucial for proving liveness since in the real-world algorithm we implement each wait statement as a loop, that means all processes are executable at all times.

Summary of the results

properties	failure susceptible algorithm									
	no failures model								possible restarts at any moment	
	never blocking		blocking in local section						never blocking	blocking in local section
	321	312	321	312	231	213	132	123	312	312
(1)	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
(2)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
(3)	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
(4)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗
(5)	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
(6)	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
(7)	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗

properties	failure resistant algorithm							
	no failures model				possible restarts at any moment			
	never blocking		blocking in local section		never blocking		blocking in local section	
	312	123	312	123	312	123	312	123
(1)	✓	✗	✓	✗	✓	✗	✓	✗
(2)	✗	✗	✗	✗	✗	✗	✗	✗
(3)	✗	✗	✗	✗	✗	✗	✗	✗
(4)	✓	✗	✓	✗	✗	✗	✗	✗
(5)	✓	✗	✓	✗	✗	✗	✗	✗
(6)	✓	✗	✓	✗	✗	✗	✗	✗
(7)	✓	✓	✓	✓	✗	✗	✗	✗

I strongly believe that most of the labels in above summary are self-explanatory. The only model's feature which might be a bit cryptic is the permutation associated with it. If we associate the following labels with assignment statements present in protocol's epilogue:

1. `chce[i] = false;`
2. `we[i] = false;`
3. `wy[i] = false;`

, then the permutation denotes the order of execution of above statements.

Labels „no failures model" and „possible restarts at any moment" refer to the possibility of nondeterministic restart of a process. Even though the restart is truly nondeterministic we assume that process atomically resets

all of its flags to default values and execute a special *watchdog procedure* before continuing. The watchdog procedure involves waiting until $\forall_k$`(!we[k] || wy[k])`.

Labels „never blocking" and „blocking in local section" refer to the ability of each process to block infinitely in its local section.

Results discussion

Failure susceptible algorithm

It should be now clear why property (7) is more on par with our intentions – every permutation of epilogue's assignments that falsifies `chce[i]` before `we[i]`, *tricks us* into thinking that process i is in the waiting room, while it is actually leaving critical section. This is also an explanation why (3) does not hold for assignments permutation 312 in „never blocking" model, while (7) does.

There exists as very simple counterexample which explains why properties (3), (4) and (7) do not hold for „blocking in local section" model with assignments permutation 321. If process α proceeds through prologue, critical section and halts between assignment 2 and 1, then the other process β can enter the waiting room and wait for any (other) process k to set `wy[k]` to true, which might never happen, since process α can get stuck in local section after setting `chce[i]` to false in the very last instruction of the epilogue.

Even simpler counterexample exists for property (2) which states that every process must visit waiting room before entering critical section in every turn of the main loop. It is easy to verify that if processes enter and leave critical section one after another, without contention, then none of them enters waiting room.

It is highly disturbing that unmodified algorithm does not ensure mutual exclusion in presence of nondeterministic restarts. Szymanski's paper [1] states that the only purpose of the modifications is to ensure liveness and linear wait properties. Additional experiments have shown that unmodified algorithm ensures mutual exclusion for two processes, but not for three. Minimal found counterexample (not necessarily minimal at all) after simplifying the model (by turning off weak fairness and possibility of process getting stuck in local section) has a total of almost 10000 steps which makes it intractable for analysis. However, after careful analysis of the algorithm before modification we have determined the scenario, which proves lack of mutual exclusion in presence of restarts. The scenario involves three processes and this is the minimal number since we have verified that mutual exclusion for two processes holds even in presence of restarts (which is easy to prove by hand). Consider the interlace as follows.

- process $P(3)$ starts the protocol and halts just before checking a condition to enter the waiting room
- process $P(2)$ executes `chce[i] = true;`
- process $P(3)$ enters the waiting room and hangs on the condition that does not hold at the moment
- process $P(2)$ executes prologue up to the instruction `wy[i] = true;`
- process $P(3)$ passes the waiting condition but stops before `chce[i] = true;`
- process $P(2)$ restarts
- process $P(1)$ executes `chce[i] = true;` and proceeds through the condition, halts before `we[i] = true;`
- process $P(3)$ executes `chce[i] = true;` and *enters critical section*
- process $P(1)$ executes rest of the prologue (skipping waiting room) and *enters critical section*

Failure resistant algorithm

Szymanski's paper [1] states that correct order of epilogue's assignments is described by permutation 123, while our analysis shows that not only it does not work for failure resistant algorithm, but also for failure susceptible one. Also note that liveness and linear wait properties do not hold when we allow processes to restart, which contradicts thesis of the paper.

Technical (implementation) details

There are just a few caveats in the models implementations, we would like to cover in more details.

First, to reduce complexity of the models, we have made all quantifiers over processes atomic and got rid of busy waiting. This way all negative results (e.g. property A does not hold) are rock-solid, positive ones would need more sophisticated argumentation.

Second, since we model process failures in such a way that resetting process forgets its internal state (local variables and instruction pointer), there is no difference between failing at instruction α or instruction β,

as long as there is no write to global variable between them. This leads to another simplification: we can nondeterministically fail after each global write, yet still cover all possible executions of real-world algorithm, which can fail at any moment.

Each model comes with `./verify` script, which allows for much broader customisation of the algorithm and environment model than the one described in this report, details and examples can be found in attached `README.md` file.

Final remarks

Failure susceptible version of the algorithm is not precisely formulated in [1]. As a result of our analysis, we have determined which permutation of epilogue's assignment statements leads to the algorithm that ensures mutual exclusion, liveness and linear wait time, even in the model where processes can block in their local sections.

According to our analysis, the modified algorithm ensures mutual exclusion in presence of restarts of processes. Neither liveness nor linear wait hold in this case.

References

[1] B. K. Szymanski
„A simple solution to Lamport's concurrent programming problem with linear wait"