Software Quality and Dependability 2020/21

Assignment 1 – Verifying safety and liveness properties with SPIN

Abstract

This assignment consists of two model checking problems aiming to verify safety and liveness properties using SPIN. Students shall learn how to model systems that have a challenging level of complexity, how to specify correctness properties and how to apply model checking in practice using SPIN. The assignment shall be carried out in groups of two students (groups of three students are accepted in exceptional circumstances). The first problem focuses on verifying a mutual exclusion protocol and the second problem consists in designing, modeling and verifying a solution to the dining philosophers problem.

1 Critical section problem

The *critical section problem* is very well known in the context of concurrent programming. We consider a system composed of two processes repeatedly accessing a critical section and a non-critical section. The general formulation of the problem considers $N \geq 2$ processes and Peterson's solution for exactly 2 processes is particularly interesting:

```
bool turn, flag[2];

active [2] proctype P()
{
non_cs:
  flag[_pid] = 1;                       /* wants to enter critical section     */
  turn = 1 - _pid;                      /* politely give turn to the other one */
  (!flag[1 - _pid] || turn == _pid);    /* block until the other doesn't want  */
                                        /* OR it is this one's turn            */
cs:
  skip;                                 /* critical section                    */

exit:
  flag[_pid] = 0;                       /* leaves the critical section         */
  goto non_cs
}
```

As shown above, Peterson's solution only uses one *boolean* for each of the two processes to signal the will to enter the critical section and a binary variable to represent which process has the turn to enter the critical section.

Question 1. Specify the following property: at most one process at a time can be in the critical section. This important correctness property is usually called *mutual exclusion*. Use SPIN to verify if this property holds for the model shown above.

Question 2. One must consider that processes may stop, possibly for an unlimited period, outside the critical section. Students must add this behavior to the model. The goal is to model situations in which processes stop requiring access to the critical section.

Question 3. Use SPIN to verify that Peterson's algorithm is free from deadlocks. Report your findings and provide a rigorous justification.

2 Dining philosophers problem

The well known *dining philosophers problem* states that N silent philosophers eat bowls of spaghetti at a round table. Between each pair of philosophers there is a single fork. Each philosopher only eats when holding both the left and the right forks at the same time. Each philosopher alternates between *thinking* and *eating*, and each fork may only be used by one philosopher at a time. This problem is often used to illustrate synchronization issues in concurrent programs.

The PROMELA code below provides a simple structure for modeling the dining philosophers problem. Each philosopher is modeled as a PROMELA process with an infinite `do` loop. As shown in the code, philosophers simply alternate between *thinking* and *eating*.

```
#define N 5

active [N] proctype Phil() {
  do
  :: printf("philosopher %d thinks...\n", _pid);
     /* pick up left and right forks if available */
     printf("philosopher %d eats...\n", _pid);
     /* put the two forks down */
  od
}
```

By saving this model in a file named `philosophers.pml` one can simulate it using SPIN: directly by running the command `spin -u100 philosophers.pml` (limits the simulation to 100 steps) or by running the simulation in your preferred IDE (jSpin or ispin). The output will be similar to the following lines:

```
          philosopher 1 thinks...
          philosopher 1 eats...
      philosopher 0 thinks...
      philosopher 0 eats...
              philosopher 2 thinks...
                      philosopher 4 thinks...
                      philosopher 4 eats...
              philosopher 2 eats...
      philosopher 0 thinks...
          philosopher 1 thinks...
              philosopher 2 thinks...
                      philosopher 4 thinks...
              philosopher 2 eats...
      philosopher 0 eats...
```

As we can see in the produced output, the philosophers already alternate between thinking and eating. However, they do so without using any forks at all. In fact, Phil$_1$ could be eating at the same time as Phil$_0$ and Phil$_2$. We therefore need to add to our model the ability to identify which philosopher is *holding* each fork. As a suggestion, consider that processes may have three locations: thinking, waiting and eating.

Question 4. Construct a model of the dining philosophers such that *each philosopher may only eat when holding both forks at the same time*. A philosopher *must pick up one fork at a time,* eat for some time and then *put the two forks down one by one* (and return to *thinking* for some time). Between picking up the first fork and the second fork, other philosophers may execute some actions.

Your model shall be consistent with the fact that every fork is a *shared resource*. The output produced by simulating your model shall be similar to the following:

```
philosopher 0 thinks...
        philosopher 2 thinks...
            philosopher 3 thinks...
                philosopher 4 thinks...
    philosopher 1 thinks...
            philosopher 3 eats using forks 3 and 4
            philosopher 3 thinks...
        philosopher 2 eats using forks 2 and 3
        philosopher 2 thinks...
    philosopher 1 eats using forks 1 and 2
            philosopher 3 eats using forks 3 and 4
            philosopher 3 thinks...
                philosopher 4 eats using forks 4 and 0
        philosopher 2 eats using forks 2 and 3
    philosopher 1 thinks...
philosopher 0 eats using forks 0 and 1
                philosopher 4 thinks...
```

Once you model the behavior as intended, inspect the simulation output that is produced (run a few random simulations). Although the output may appear to be correct, this is clearly insufficient to *prove* any interesting properties. For this reason we must specify and verify some elementary properties.

Question 5. Consider the following correctness property: there may never be any single fork being held by more than one philosopher simultaneously. Formally specify this property and use SPIN to verify if it holds.

Question 6. The following correctness property should also hold: every philosopher only eats when holding both the left and the right fork. Formally specify this property and use SPIN to verify if it holds.

The two previous elementary properties should be specified using *assertions*. Most likely, the model constructed to answer the above questions fails to fulfil both properties. Carefully examine the counterexamples found by SPIN and understand why this is so.

Question 7. Write a new version of the PROMELA model to guarantee that there may never be a fork being held by more than one philosopher simultaneously. Atomic sequences may be used to pick up each individual fork, but not both.

Once you obtain a model that fulfils this property about forks, run several random simulations or, alternatively, run a single but very long simulation. Focus on whether you observe any `timeout` occurrences.

The above property about forks is an instance of *mutual exclusion*, in the sense that each philosopher should have exclusive access to a shared resource until releasing it. More generally, it is a *safety property* because it specifies unintended behavior (bad things never happen). We now turn to specifying *liveness properties* (good things eventually happen).

> **Question 8.** Consider the following liveness property: some philosopher waiting to eat will eventually do so. This property is usually called *deadlock freedom*. Use SPIN to show if it holds and interpret SPIN's counterexamples (which are likely to exist at this step).

Deadlock freedom is one of the main challenges in the dining philosophers problem. There are numerous solutions that avoid deadlock and SPIN's counterexamples should be helpful to find one.

> **Question 9.** Propose and model a solution that avoids deadlocks in the dining philosophers problem. The model must be consistent with the restriction that philosophers are silent (*i.e.*, shared variables and message passing are not permitted).

Notice that deadlock freedom only specifies that *some* philosopher must make progress and eat. In fact, *some other* philosopher may well be stuck thinking forever. Students must add this blocking behavior to the model: philosophers may halt forever not wanting to eat. This can be achieved similarly to Question 2.

At this point you should have a model of the dining philosophers problem that solves *mutual exclusion* and *deadlock freedom*. SPIN should be able to verify the proposed solution exhaustively up to $N = 5$ or higher if possible.

> **Question 10.** Verify the final version of your solution with regards to the correctness properties. Examine and describe the necessary assumptions regarding *fairness* to guarantee those properties.

3 Recommended approach and reporting

The final report shall consist of a text file (acceptable formats include plain `.txt` files or `.md` at most) and all `.pml` files written to answer all questions. The written report shall clearly describe the modeling and verification steps taken to answer the questions. It should also specify the necessary switches passed to all the components: SPIN, C compiler and `pan` verifier.

Informally describe the algorithm to pick up and put down the forks and explain the attempted solutions to make the philosophers eat in an orderly manner. Describe how these solutions are modeled in PROMELA and how correctness properties are formalized from the natural language specification. Arguments must be rigorous even if informal when claiming that the models are coherent with the actual system.

When writing your code bare in mind that it should be readable and understandable by others, just like any programming language. The translation from the idealized system to a language such as PROMELA should be natural and intuitive. The same applies to the specification of correctness properties. Furthermore, aim for writing the smallest sufficient model and avoid unnecessary complexity.