Assignment 1
Do a random simulation of the following PROMELA Programm. What are the sources of nondeterminism in it? Do an interactive simulation and become a “Winner”.

active proctype P() {
    int a = 0;
    do
    :: a < 21 -> a = a + 2
    :: a < 21 -> a = a + 5
    :: a > 21 -> printf("Loser.\n"); break
    :: a == 21 -> printf("Winner.\n"); break
    od
}

Assignment 2
Two processes need to share an access-restricted resource. The relevant part accessing the resource is confined to a critical section. A flag indicates that the process has entered the critical section.

The following piece of code models this situation in PROMELA.

#define mutual_exclusion 1 /* to do */

bool flag[2]; // initialized to 0
int critical; // # of threads in CS, initialized to 0

active[2] proctype P() {  // int _pid; is an implicit variable holding the process' id
    // wait for other process to not be in CS
    do
    :: flag[1 - _pid] == 0 -> break;
    od;

    // since the other process is now not in CS, I can set my flag.
    flag[_pid] = 1;

    // enter CS
    critical ++;
    printf("%d is now in critical section. (%d)\n", _pid, critical);
    critical --;
    flag[_pid] = 0;
    // leave CS
}
ltl { 1 /* to do */ }

(a) Why is this problematic?
(b) Add an according LTL specification whose verification fails and thus exposes the problem.
(c) Adjust the promela code using an atomic block. Prove that mutual exclusion for the critical section is thus ensured. What assumptions about the system did you make by changing the model?
(d) Resolve the problem without the atomic block by implementing Peterson's algorithm. Prove that mutual exclusion for the critical section is ensured.

Assignment 3

This sorting algorithm, developed for use on parallel processors, compares all odd-indexed list elements with their immediate successors in the list and, if a pair is in the wrong order (i.e., if the first is larger than the second) swaps the elements. The next step repeats this for even-indexed list elements (and their successors). The algorithm iterates between these two steps until the list is sorted.

On \( \frac{n}{2} \) parallel processors that have random access to the array of \( n \) elements to be sorted, the processors all concurrently do a compareexchange operation with their neighbours, alternating between odd-even and even-odd pairings in each step. The algorithm has linear runtime as comparisons can be performed in parallel. The skeleton of a PROMELA implementation that uses shared memory for synchronisation is presented in the following. The driver code spawns \( \frac{n}{2} \) processes.

```plaintext
#define N 5
#define M 5

byte array[N];
bit state[N];

init {
    // fill the array with random numbers between 0 and M
    // start the processes
    i = 1;
    do
    #: i < N -> run sort(i); i = i + 2;
    #: i == N -> break;
    od;
}

proctype sort(byte id) {
    byte i = 0;
    byte tmp;

    // while i < N do
    //   if i is even and state[id-1] is 0 then
    //     sort id-1 and id in array
    //     state[id-1] = 1
    //   if i is odd and state[id+1] is 1 then
    //     sort id and id+1 in array
    //     state[id+1] = 0
    //   i++
```
// Local sortedness
assert(array[id-1] <= array[id] && array[id] < array[id+1]);
}

(a) Implement the sorting algorithm following this skeleton code.
(b) Verify that the result of the algorithm is a sorted array. Can the proposed assertions be used for this purpose?
(c) Challenge: Verify that the result of this algorithm is a permutation of the original array.
(d) Tweak your model to obtain higher numbers for $N$ and $M$. 