Algorithmic Verification of Programs
(Final Examination)

NB: There are two parts: Exercices 1 - 2, and Exercices 3 - 6. Please send two separate files, one for each part.

Exercice 1: Burns protocol – TSO

Figure 1 shows an implementation of the Burns mutual exclusion protocol. The program has two parallel processes $P_0$ and $P_1$, sharing the two boolean variables $F_0$ and $F_1$.

![Figure 1 – Burns algorithm](image)

**Question 1:** Verify that the program, when executed under the SC memory model, satisfies indeed mutual exclusion, which means that the two processes are never in the critical section in the same time.

**Question 2:** Show that under the TSO memory model this program does not satisfy mutual exclusion, which means that there are behaviors of the program where both of the processes are in the critical section.

**Question 3:** Add an irreducible set of memory fences to the program ensuring mutual exclusion. (Irreducible means that removing any fence from the set leads to a program that does not satisfy mutual exclusion, i.e., a program that have behaviors violating this property).

All your answers must be justified rigorously.
Exercice 2 : TSO – Context Bounded Analysis

Assume we are given a program running over the TSO memory model where each process is finite state (finite data domain, no procedure calls). We consider an under-approximate analysis that considers only computations having a bounded number of *phases*. A phase is defined to be a computation segment where only one process is active (can perform write and read operations), and only write operations from its store buffer can be committed to the main memory. During a phase, the non-active processes are idle and their store buffers remain unmodified (the write operations they contain cannot be committed and used by the active process). So, during a phase, the active process, say $P$, is running in isolation, and when the computation switches to another phase, another process, say $P'$, becomes active and is resumed with its store buffer as it was left after the last phase $P'$ was active. Similarly, if process $P$ becomes active in the future, it will find its store buffer unchanged. Notice however, that at each phase switch, the state of the main memory is changing. Notice also that during a phase, some of the write operations issued by the active process will be committed in the same phase, but other operations will be committed only in forthcoming phases.

A computation is $k$-bounded, for some integer $k \geq 1$ is each process can be active in at most $k$ phases in the computation. Our goal is to define a procedure for solving state reachability with $k$-bounded computations.

**Question 1 :** Assume $k = 1$. In this case each process will be active only once, i.e., after being interrupted, the process will not be resumed. This means that the write operations that will not be committed during its phase of activity will not be committed later in the computation. But these writes are still important since they might be used by the process for its own read operations.

Define what is the information needed for performing the state reachability analysis with 1-bounded computations, and provide an algorithm for that.

**Question 2 :** Same question for $k = 2$.

Hints : In this case, writes issued by a process in its first phase can either be committed in that same phase, or in the next phase, or never. Answer the following questions :

- how to execute the read operations in the second phase? when is it possible to read writes issued in the first phase and committed only in the second phase?
- can writes issued in the second phase be committed? if yes, in which case and in which phase?

**Question 3 :** Generalize the solution to the case of any $k \geq 1$. 
Figure 2 – Histories for Exercise 3

**Exercice 3 : Consistency criteria**

1. Figure 2(a)-(c) picture three histories of a concurrent *register* with methods *write(v)* writing the value v to the register and *read()* returning the current value of the register. Also, Figure 2(d)-(e) picture two histories of a concurrent stack. For each of these histories, give the correctness criteria they satisfy among quiescent consistency, sequential consistency, linearizability, eventual consistency, and causal consistency. We consider that the specification is the standard sequential data type in each case (for eventual/causal consistency, we assume that the replicated data type defines the return value of a *read()* or *pop()* in function of the operations in the context ordered by arbitration order). Provide a short justification for your answer.

2. Give an example of a history of a concurrent register that is linearizable but not sequentially consistent, and another that is sequentially consistent but not causally consistent. Provide a short justification for your answer.

**Exercice 4 : Stack object**

The following code presents an implementation of a concurrent stack (each statement is executed in a single atomic step) :
void push(int x) {
    i = range++;
    items[i] = x;
}

int pop() {
    t = range - 1;
    for i = t downto 0 {
        x = swap(items[i],null);
        if ( x != null )
            return x
    }
    return empty;
}

This stack stores the elements into an infinite array items, a shared variable range keeping the index of the first unused position in items (initially, range is 0). The push method stores the input value in the array while also incrementing range. The pop method first reads range and then traverses the array backwards starting from this position, until it finds a position storing a non-null element (array cells can be nullified by concurrent pop invocations). It atomically reads this element and stores null in its place. If the pop reaches the bottom of the array without finding non-null cells, then it returns that the stack is empty.

Does this implementation admit fixed linearization points? Justify your answer.

Is this implementation linearizable? Justify your answer.

Exercice 5 : Counter objects

1. The following code presents a possible implementation of a concurrent counter (each statement is executed in a single indivisible step):

   // counter is an integer array of size 2

   void inc() {
       counter[0]++;
   }

   void dec() {
       counter[1]--;
   }

   int read() {
       c1[0] = counter[0];
       c1[1] = counter[1];
       return c1[0] + c1[1];
   }

The value of the counter is represented using an array counter of size 2, more precisely, by the sum of the elements in counter (see the return value of read). The method inc increments the counter by incrementing the first element of the array counter. Similarly, dec decrements the counter by decrementing the second element of the array counter. The method read returns the sum of the array elements.

Is this implementation linearizable w.r.t. the standard specification of a counter (i.e., read returns a value which equals the number of increments inc minus the number of decrements dec, that precede it)? Justify your answer.

2. What about the following revised version, where the method read repeatedly reads the two elements of the array counter until two successive reads have the same sum. Is this version linearizable? Justify your answer.
void inc() {
    counter[0]++;
}  
void dec() {
    counter[1]--;
}  

int read() {
    c1[0] = counter[0];
    c1[1] = counter[1];
    repeat
        counter[0]++;
        c2 = c1;
        c1[0] = counter[0];
        c1[1] = counter[1];
        until c1[0] + c1[1] == c2[0] + c2[1]
    return c1[0] + c1[1];
}  

Exercice 6 : Testing concurrent stack histories
Consider the problem of deciding if a differentiated concurrent stack history is linearizable. Show that this problem is polynomial-time (Indication : Using the “bad pattern” characterization presented in the lecture is helpful).

1. A value is pushed at most once.