Algorithmic Verification of Programs (Final Examination)

The subject of the exam contains two parts : Exo 1-2 and Exo 3-6. Write solutions for each part on a different exam sheet. The exam duration is 3 hours.

Exercice 1 : Correctness under Weak Memory Models

Figure 1 shows a program implementing the Peterson mutual exclusion protocol. The program has two parallel processes P_1 and P_2, sharing the two boolean variables b_1 and b_2, and the variable x ranging over the set $\{1, 2\}$.

Process P1

```
loop forever
    noncritical ;
    b1 := true ;
    x := 2 ;
    wait until   x = 1 or (not b2)
    do
        critical
    od
    b1 := false
end loop
```

Process P2

```
loop forever
    noncritical ;
    b2 := true ;
    x := 1 ;
    wait until   x = 2 or (not b1)
    do
        critical
    od
    b2 := false
end loop
```

FIGURE 1 – Peterson algorithm

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).

Exercice 2 : Decidability

We consider the problem of parametrized verification of concurrent programs having an arbitrary number of thread, which means verifying that the program is correct for any number of threads. We assume that threads do not have procedure calls, and therefore they can be modeled using state machine with a finite number of control states. Moreover, we consider that the threads use a finite number of shared variables according to the PSO weak memory model. The data domain of the variables is finite, and therefore there is a finite number of possible write and read operations.

We recall that the PSO model is the relaxation of TSO where writes on different variables issued by a same thread can be reordered. This can be modeled by considering that each thread uses a store buffer for each variable (contrary to TSO where each thread has only one store buffer use to store all the write operations issued by that thread). We recall that the reachability problem for a fixed number of threads running under PSO is decidable. The decidability is proven by reduction to the reachability problem in well-structured systems : roughly, PSO store buffers can be considered as lossy. In this exercice, we want to extend this result to the case of an arbitrary number of threads.

For that, we recall some known fact about well-quasi orderings :

1. If S is a finite set, then $(S, =)$ is a well-quasi ordered set.
2. If $(S_1, \leqslant_1)$ and $(S_2, \leqslant_2)$ are two well-quasi ordered sets, then $(S_1 \times S_2, \leqslant_{1\times 2})$ is also a well-quasi ordered set, where $S_1 \times S_2$ is the cartesian product of S_1 and S_2, and $\leqslant_{1\times 2}$ is the product relation of $\leqslant_1$ and $\leqslant_2$, i.e., $(u_1, v_1) \leqslant_{1\times 2} (u_2, v_2)$ if and only if $u_1 \leqslant_1 v_1$ and $u_2 \leqslant_2 v_2$.
3. If $(S, \leqslant)$ is a well-quasi ordered set, then $(S^*, \leqslant^*)$ is a well-quasi ordered set, where S^* is the set of all words over the alphabet S, and $\leqslant^*$ is the subword relation over S^*, modulo the relation $\leqslant$ on the individual

elements of S, i.e., given two words $\alpha = u_1 \cdots u_n$ and $\beta = v_1 \cdots v_m$, $\alpha \leqslant^* \beta$ if and only if $n \leqslant m$ and there exists $v_{i_1} v_{i_2} \cdots v_{i_n}$ a subword of β such that $u_j \leqslant v_{i_j}$ for every $i \in \{1, \cdots, n\}$.

Question : Show that the parametrized reachability problem under PSO is decidable.

Hints :

- Define a formal model of programs with an unbounded number of threads running over PSO : how to represent the configuration of one thread, how to represent configurations of programs with an arbitrary number of threads, what is a transition is such a system.
- Show that the defined model is a well-structured system : there is a well-quasi ordering on the set of configurations of the system, and the transition relation of the system is monotonic w.r.t. that well-quasi ordering.

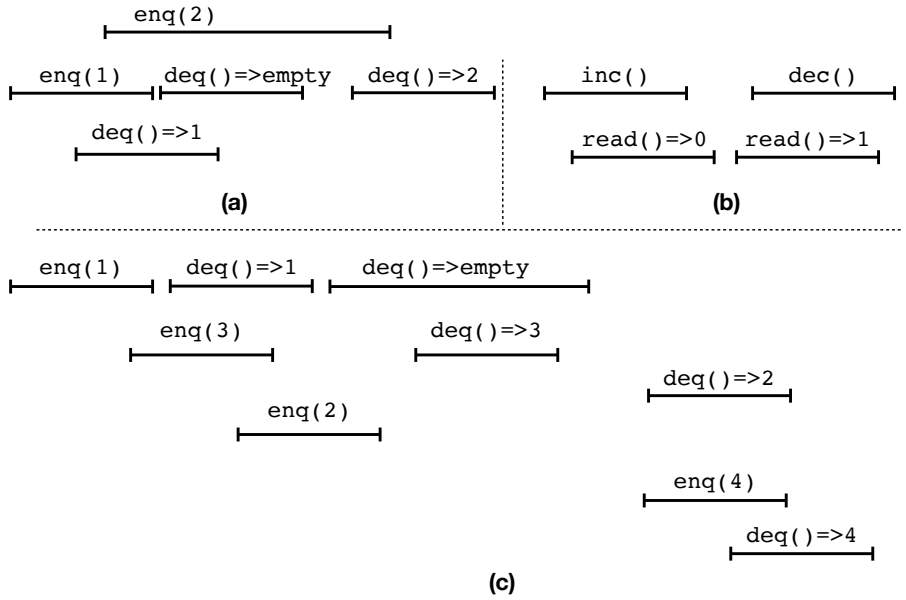

FIGURE 2 – Histories for Exercice 3

Exercice 3 : Consistency criteria

1. Figure 2 shows two histories of a concurrent queue and one history of a concurrent counter. For each of the these histories, say whether it is linearizable and provide a short justification for your answer (e.g., showing a possible assignment of linearization points when the history is linearizable).

2. Give an example of a history of a *concurrent stack* that is *linearizable but not sequentially consistent*, *sequentially consistent but not quiescent consistent*, and *quiescent consistent but not sequentially consistent*. Write a short justification in each case.

Exercice 4 : Concurrent queue

The following code presents an implementation of FIFO queue :

```
void enqueue(int x) {
  int slot;
  do {
    slot = tail;
  while (! CAS(tail, slot, slot+1))
  items[slot] = x;
}
```

```
int dequeue() {
  int value, slot;
  do {
    slot = head;
    value = items[slot]
    if (value == null)
      return "empty";
   while (! CAS(head, slot, slot+1))
  }
  return value;
}
```

This queue stores the elements into an infinite array `items`, and it has two shared variables : `tail` represents the index of the next slot in which to place an item, and `head` represents the index of the next slot from which to remove an item. These two variables are initially set to 0.

1. Is this implementation linearizable ? Justify your answer.

Exercice 5 : Concurrent set

```
class Node {
  int val;
  bool marked;
  Node next;
}

// Global Vars
Node Root;

bool contains(int key) {
  Node pred, curr;
  int k;
  pred = Root;
  curr = Root->next;
  k = curr->val;
  // locate
  while (k < key) {
    pred = curr;
    curr = curr->next;
    k = curr->val;
  }
  return (k == key);
}
```

```
bool add(int key) {
  Node pred, curr, entry;
  int k;
  while (true) {
    pred = Root;
    curr = Root->next;
    k = curr->val;
    // locate
    while (k < key) {
      pred = curr;
      curr = curr->next;
      k = curr->val;
    }
    if (k == key) return false;
    entry = new Node(key,false,curr);
    atomic {
      if (pred->next == curr
          && !pred->marked) {
        pred->next = entry;
        return true;
      }
    }
  }
}
```

```
bool remove(int key) {
  Node pred, curr, next;
  int k;
  while (true) {
    pred = Root;
    curr = Root->next;
    k = curr->val;
    // locate
    while (k < key) {
      pred = curr;
      curr = curr->next;
      k = curr->val;
    }
    if (k > key) return false;
    atomic {
        if (pred->next == curr
            && !pred->marked) {
        next = curr->next;
        curr->marked = true,
        pred->next = next;
        return true;
      }
    }
  }
}
```

FIGURE 3 – A concurrent set.

We consider the implementation of a concurrent set given in figure 3. This object provides three methods : `contains`, `add` and `remove`. Notice that the state of the object is organized as a sorted linked list, where the field `marked` indicates whether an item should be considered in the list or not (see `remove`). Also, the first node of the list (`Root`) is a sentinel node whose value is irrelevant. The `atomic` keyword marks code that is executed in a single indivisible step.

1. Consider only the methods `add` and `remove`. Is this subset of the object linearizable ? If so indicate the linearization points in these methods.

2. Consider now all the methods. Is the object linearizable ? Can we provide linearization points for each of the methods ? Does the implementation satisfy static linearizability ? Justify your answer.

Exercice 6 : Reductions to linearizability

Consider the problem of checking whether a word w over an alphabet Σ is included in a regular language L. Is this problem reducible to checking linearizability of *completed* histories, i.e., where every invocation of a method returns ? More precisely, does there exist a library *Lib* and a sequential specification *Spec* such that *Lib* admits a completed history that is not linearizable w.r.t. *Spec* if and only if $w \in L$? (*Suggestion* : You can try to use the ideas and the synchronization mechanisms we used to show undecidability, EXPTIME-completeness, for linearizability).