How to establish whether a concurrent code satisfies certain properties

The aim of this document to present some systematic ways to show whether a concurrent code satisfies certain properties like mutual exclusion, being deadlock-free and starvation-free.

The content of this document is supplementary and is not part of the examinable material. The treatment of the concurrency problems below go beyond what is expected from students. The hope is that students will find this formal approach enlightening.

1 Incorrect codes

In general, it is easier to show if there is a flaw in the code than showing correctness. So we will start with some incorrect "solutions" to the critical section problem.

Consider the code (called 1st Attempt in the slides) for two processes, P_0 and P_1, in Figure 1.

The labels L_i are not part of the code; we will use them to refer to *locations*, i.e., to specify where a process is in its execution. Thus if process P_0 is at location L_i, then the next instruction it will execute is L_i. The critical section is divided into three lines ($L_4 - L_6$) emphasizing that it is usually longer than a single instruction, so while one process is in its critical section, a process switch can occur.

A *state* will be an ordered tuple (L_i, L_j, x, y) indicating that P_0 is at location L_i, P_1 is at location L_j and that the current values of `flag[0]` and `flag[1]` are x and y, respectively. We will abbreviate `false` by f and `true` by t, respectively. Both processes start at L_1 and the initial values of the flags are false, i.e., the *initial state* is (L_1, L_1, f, f), and the process which is chosen by the scheduler can make a *transition*, i.e., execute one instruction. For instance, if P_0 runs first, then it executes the statement

```
    int flag[2] = {false, false};

    void first-attempt (int i)
    {
L1  while(true){
L2    while (flag[1-i]);  //loop
L3    flag[i] = true;
L4    /* start-critical-section */;
L5    /* critical-section */;
L6    /* end-critical-section */;
L7    flag[i] = false;
L8    /* noncritical-section */;
    }
    }
```

Figure 1: 1st Attempt

`while(true)` and its new location will be L_2. We can denote this step by $(L_1, L_1, f, f) \overset{0}{\mapsto} (L_2, L_1, f, f)$.

To show that the code from Figure 1 does not achieve mutual exclusion we have to find a *run* during which both processes can enter their critical sections. That is, we have to find an (interleaved) execution of the two processes both ending up at location L_5. The following run will do:

$$(L_1, L_1, f, f) \overset{0}{\mapsto} (L_2, L_1, f, f) \overset{0}{\mapsto}$$
$$(L_3, L_1, f, f) \overset{1}{\mapsto} (L_3, L_2, f, f) \overset{1}{\mapsto} (L_3, L_3, f, f) \overset{1}{\mapsto} (L_3, L_4, f, t) \overset{1}{\mapsto}$$
$$(L_3, L_5, f, t) \overset{0}{\mapsto} (L_4, L_5, t, t) \overset{0}{\mapsto} (L_5, L_5, t, t)$$

In this run, first P_0 gets the CPU and executes the statements L_1, L_2 and L_3. At this point there is a process switch, so P_1 runs and executes the statements L_1, L_2 (observe that `flag[0] = false` at this point), L_3 and L_4. After another process switch, P_0 executes L_3 (note that P_0 already checked `flag[1]` when executed L_2) and L_4.

Exercise. In Figure 2 we have the modification of the code from Figure 1

(called 2nd Attempt in the slides). Find a run showing that the code from Figure 2 does not avoid deadlock[1].

```
    int flag[2] = {false, false};

    void second-attempt (int i)
    {
L1  while(true){
L2    flag[i] = true;
L3    while (flag[1-i]);  //loop
L4    /* start-critical-section */;
L5    /* critical-section */;
L6    /* end-critical-section */;
L7    flag[i] = false;
L8    /* noncritical-section */;
    }
    }
```

Figure 2: 2nd Attempt

2 Correct codes

When we want to prove correctness we usually look for some *invariance properties*, IP for short, and use them to show the correctness of the code. IPs are properties of the code that are maintained during every execution of the code. Usually there are two steps in proving that an IP remains true:

1. show that IP is true initially
2. show that IP remains true whenever a process makes a transition (i.e., executes one instruction).

Then we can conclude that IP is true during all runs.

[1]Strictly speaking we should talk about livelock, but we will ignore the difference between a process being blocked or executing the same statement indefinitely.

2.1 Critical section with semaphores

Figure 3 shows the typical scheme for solutions to critical section problems using semaphores. We assume that there are a finite number of processes: $P_1, \ldots, P_n$.

```
    binary semaphore s = 1;

    void cs-with-semaphore (int i)
    {
L1  while(true){
L2    wait(s);
L3    /* start-critical-section */;
L4    /* critical-section */;
L5    /* end-critical-section */;
L6    signal(s);
L7    /* noncritical-section */;
    }
    }
```

Figure 3: Critcal Section with Semaphore

We want to prove mutual exclusion, i.e., that at most one process can be in the critical section (at locations $L_3 - L_6$). Sometimes it is easier to show a stronger statement, like the following.

IP1 At any point during the execution of the processes precisely one of the following statements hold:

- P_1 is in the critical section,
- P_2 is in the critical section,
- ...
- P_{n-1} is in the critical section,
- P_n is in the critical section,
- $s = 1$.

In case of $n = 2$ we can write this as the formula

$$(A_1 \wedge \neg A_2 \wedge (s = 0)) \vee (\neg A_1 \wedge A_2 \wedge (s = 0)) \vee (\neg A_1 \wedge \neg A_2 \wedge (s = 1))$$

where A_i stand for the statement 'P_i is in the critical section', $\neg$ is negation, $\wedge$ is conjunction and $\vee$ is disjunction (taking into account that s is either 0 or 1).

We prove that IP1 is invariant.

1. The initial state is when all the processes are at location L_1, and IP1 is clearly true here, since none of the processes is in the critical section and the initial value of s is 1.

2. Next we have to show that IP1 remains true during all the possible transitions.

   If a process makes the transition $L_1 \mapsto L_2$, then the value of s does not change and the process does not enter the critical section, so IP1 remains true.

   Next assume that one of the processes, say P_i, tries to make the transition $L_2 \mapsto L_3$. There are two cases according to the current value of s. First assume that $s = 1$. Since IP1 is true before the transition, there is no process in the critical section. In this case the transition is enabled, i.e., P_i can move to L_4 and, according to the wait operation on semaphores, the new value of s is 0. Thus P_i is in the critical section and $s = 0$, so IP1 remains true. The second case is when $s = 0$. Since we assume that IP1 is true before the transition, there must be one process already in the critical section. Furthermore, according to the wait operation on semaphores, P_i is blocked and cannot move into the critical section. Thus IP1 remains true in this case as well.

   During transitions $L_3 \mapsto L_4, L_4 \mapsto L_5, L_5 \mapsto L_6$ the process remains in the critical section and s is unchanged, so IP2 remains true.

   During transition $L_6 \mapsto L_7$ a process (the only process which has been in) leaves the critical section by performing a signal on s. There are two cases according to whether there are processes blocked on s. If the semaphore queue is empty, then the value of s is increased to 1 and there will be no process in the critical section. If there are processes waiting on s, then the value of s remains 0 and one of the processes is unblocked so that it can resume execution from location L_3 (i.e., inside the critical section). So IP1 remains true in this case too.

   Finally, transition $L_7 \mapsto L_1$ does not change the truth value of IP1.

We are ready to show mutual exclusion. Assume that P_i is in the critical section and another process P_j tries to enter the critical section. According to IP1 the current value of s is 0. So P_j becomes blocked when it tries to make the transition $L_2 \mapsto L_3$ (i.e., tries to perform wait on the empty semaphore).

Next we show that deadlock cannot occur. Deadlock would occur if none of the processes were able to enter the critical section. This means that $s = 0$ (otherwise one of the processes can perform the wait on s). By IP1 there is already a process in the critical section, so there is no deadlock.

2.2 Peterson's solution

Recall Peterson's solution for two processes (again identified by 0 and 1, respectively), see Figure 4.

```
     int turn;
     int interested[2] = false;

     void peterson (int i)
     {
L1   while(true){
L2     interested[i] = true;
L3     turn = i;
L4     while (turn == i && interested[1-i] == true);  //loop
L5     /* start-critical-section */;
L6     /* critical-section */;
L7     /* end-critical-section */;
L8     interested[i] = false;
L9     /* noncritical-section */;
     }
     }
```

Figure 4: Peterson's solution

The code avoids deadlock, since

```
turn == i && interested[1-i] == true
```

cannot be true for both processes at the same time (`turn` is either 0 or 1), whence one of the processes can make the transition $L_4 \mapsto L_5$.

Showing mutual exclusion is a bit harder. First observe the following obvious invariance property:

IP1 If process P_i is in the critical section then `interested[i] == true`.

Assume that one of the processes, say P_0, is in the critical section and the other process P_1 wants to enter the critical section. We have to show that P_1 cannot enter the critical section. We use case distinction according to the location of P_1 when P_0 entered the critical section, i.e., made the transition $L_4 \mapsto L_5$.

Locations L_1, L_2, L_3: In these cases P_1 did not execute the statement `turn = i` when P_0 entered the critical section. Hence P_1 has to execute L_4, i.e., `turn == 1` when P_1 is trying to make the transition $L_4 \mapsto L_5$. Also we have `interested[0] == true` by IP1. Thus

```
turn == i && interested[1-i] == true
```

is true for P_1, i.e., P_1 cannot enter the critical section.

Location L_4: If P_1 was at location L_4 when P_0 entered the critical section, then it already executed L_3, i.e., `turn = i`. Thus we had

```
turn == 1 && interested[0] == true
```

when P_0 made the transition $L_4 \mapsto L_5$. Hence P_1 cannot make the transition $L_4 \mapsto L_5$.

Locations L_5, L_6, L_7, L_8: According to this (hypothetical) scenario P_1 was in the critical section when P_0 entered the critical section. By the previous case, we have that P_0 was not at L_4 when P_0 entered the critical section (see the case Location L_4 above with the roles of P_0 and P_1 interchanged). Thus P_0 had to execute L_3 before entering the critical section. This means that when it tried to make the transition $L_4 \mapsto L_5$ we had `turn == 0` and `interested[1] == true` (since P_1 was already in the critical section). Thus P_0 in fact was not able to make this transition, showing that this case is impossible.

Location L_9: The argument is the same as for the case Locations L_1, L_2, L_3.

2.3 Critical section with exchange

Recall the code for critical section for N processes using the `exchange` instruction from Figure 5.

```
     int bolt = 0;
     int key[N] = 1;

     void cs-with-exchange (int i)
     {
L1   while(true){
L2     while (key(i) == 1) {exchange (key(i), bolt)};
L3     /* start-critical-section */;
L4     /* critical-section */;
L5     /* end-critical-section */;
L6     exchange (key(i), bolt);
L7     /* noncritical-section */;
     }
     }
```

Figure 5: Critcal Section with Exchange

We want to show that

- mutual exclusion is achieved and
- deadlock is avoided.

The main idea of the code is that the following two properties remain true during every execution of the processes.

IP1 There is a processes in the critical section (i.e., at locations $L_3 - L_6$) if and only if `bolt` $= 1$.

IP2 A process P_i is in its critical section (i.e., at locations $L_3 - L_6$) if and only if `key(i)` $= 0$.

Let us assume for the moment that the above conditions are indeed true to show correctness of the code.

For mutual exclusion assume that process P_i is in the critical section. Then by IP1 we know that `bolt` $= 1$. Now assume that another process P_j tries to enter the critical section. Since P_j is not in the critical section yet, IP2 tells us that `key(j)` $= 1$. If P_j tries to make the transition $L_2 \mapsto L_3$, this attempt will fail (`key(i)` remains 1 after exchanging it with `bolt`). Hence P_j cannot enter the critical section while P_i is in the critical section.

Showing that deadlock is avoided is easy too. Deadlock would occur if none of the processes would be able to enter the critical section, i.e., to make the transition $L_2 \mapsto L_3$. Since none of the processes is in the critical section, IP1 tells us that `bolt` $= 0$. Then one of the processes (the first one to be scheduled), say P_i, can make the transition, since `key(i)` will be 0 after exchanging it with `bolt`.

It remains to show that IP1 and IP2 are invariant. Although they are rather intuitive, proving them is not straightforward. Let us start with the 'only if' direction of IP2.

IP2' When process P_i is in its critical section (i.e., at locations $L_3 - L_6$) then `key(i)` $= 0$.

The transition $L_2 \mapsto L_3$ can be made only when `key(i)` $= 0$ (maybe after exchanging `key(i)` and `bolt`), otherwise the process would execute the statement L_2 again. Thus `key(i)` $= 0$ when P_i is at L_3. Other processes do not have access to `key(i)`, hence it remains 0 until P_i leaves the critical section (by executing L_6).

The following additional property will be useful.

IP0 The sum of `bolt` and the `key(i)` variables is always N:

$$\texttt{bolt} + \sum_{1}^{N} \texttt{key(i)} = N$$

Let us check that IP0 is indeed maintained during every execution. The initial values of these variables (`bolt` $= 0$ and `key(i)` $= 1$ for each process P_i) make IP0 true at the initial state. The "dangerous" transitions are when the values of `bolt` and `key(i)` are updated. These are the transitions $L_2 \mapsto L_3$ and $L_6 \mapsto L_7$, i.e., when a process executes the statements L_2 and L_6, respectively. In both cases the sum `bolt` $+ \sum_1^N$ `key(i)` remains unchanged, since the values of `bolt` and `key(i)` are swapped. Thus we can conclude that IP0 is indeed invariant.

Using IP0 we can show IP1 as follows. Initially IP1 is true, since `bolt` $=$ 0 and none of the processes is in the critical section. Whichever process

makes the transition $L_1 \mapsto L_2$ IP1 remains true (the value of `bolt` does not change and the process remains outside of the critical section). The "dangerous" transitions are $L_2 \mapsto L_3$ and $L_6 \mapsto L_7$. First, consider the case when a process P_i enters its critical section by making the transition $L_2 \mapsto L_3$. We saw that this transition can be made only when `key(i)` $= 0$ and it stays 0 while P_i is in the critical section. By IP0, we have that `bolt` must be 1 after this transition. Hence IP1 remains true during this transition. IP1 remains true whenever a process makes any of the transitions $L_3 \mapsto L_4$, $L_4 \mapsto L_5$ or $L_5 \mapsto L_6$. Next look at transition $L_6 \mapsto L_7$ made by process P_i. P_i is in its critical section at L_6, so we know by IP2' that `key(i)` $= 0$. Since IP1 is true at L_6, it must be that `bolt` $= 1$. After executing L_6 process P_i is outside its critical section and `bolt` $= 0$, as desired. The remaining transitions $L_6 \mapsto L_7$ and $L_7 \mapsto L_1$. do not change the truth value of IP1.

Finally, we can prove the 'if' direction of IP2:

When processes P_i is not in its critical section then `key(i)` $= 1$.

Obviously, this property is true initially and the transition $L_1 \mapsto L_2$ does not change the value of `key(i)`. At $L_3 - L_6$, P_i is in the critical section, so we do not have to check the value of `key(i)`. When P_i leaves the critical section, it changes the value of `key(i)`. By IP1, `bolt` $= 1$ when P_i is at L_6. Thus after executing L_6, we have `key(i)` $= 1$, and it stays unchanged until P_i tries to enter the critical section again.