Homework 3 Commentary
COMP3151 T2 2021

Vincent Jackson
Raphael Douglas Giles
Johannes Åman Pohjola

August 3, 2022

Question 1: Manna-Pnueli Algorithm

This question asked you to break up the atomic if statement of the Manna-Pnueli algorithm, and find a trace showing that it no longer has mutual exclusion.

$wantq, wantq \coloneqq 0, 0$			
p_1:	*non-critical section*	q_1:	*non-critical section*
p_2:	**if** $wantq = -1$	q_2:	**if** $wantp = -1$
p_{2t}:	**then** $wantp \coloneqq -1$	q_{2t}:	**then** $wantq \coloneqq 1$
p_{2f}:	**else** $wantp \coloneqq 1$	q_{2f}:	**else** $wantq \coloneqq -1$
p_3:	**await** $wantq \neq wantp$	q_3:	**await** $wantq \neq -wantp$
p_4:	*critical section*	q_4:	*critical section*
p_5:	$wantp \coloneqq 0$	q_5:	$wantq \coloneqq 0$

Such a trace is (where q_i occuring in the trace means that that statement gets run)

$$\{wantq = 0 \land wantq = 0\}$$
$$p_1$$
$$p_2 \qquad (wantp \neq -1)$$
$$q_1$$
$$q_2 \qquad (wantq \neq -1)$$
$$q_{2f}$$
$$\{wantq = 0 \land wantq = -1\}$$
$$q_3 \qquad (wantq \neq -wantp)$$
$$p_{2f}$$
$$\{wantq = 1 \land wantq = -1\}$$
$$p_3 \qquad (wantq \neq wantp)$$

Question 2: Szymanski's Algorithm

In this question, you were asked to edit a Promela version of Szymanski's algorithm to determine if the ordering of the await tests was relevant to the correctness of the algorithm.

The correct result for this question is that reordering any of

```
flag[0] < 3;
flag[1] < 3;
flag[2] < 3;
```

in any way, regardless of the (internal) ordering of the other awaits, causes mutual exclusion to fail. Internally reordering any of the other awaits does not break the algorithm.

Question 3: Composing Solutions

(a) **If either A or B satisfies mutual exclusion, then C satisfies mutual exclusion:**

This is true. To see why, note that if B satisfies mutual exclusion, then no two processes can be in p_4 at the same time, and if A satisfies mutual exclusion, then no two processes can be in p_3, p_4 or p_5 at the same time (e.g. you could not have one process in p_3 and another in p_5). Hence, both cases guarantee no two processes can be in p_3 at the same time, so mutual exclusion is satisfied for C.

(b) **If A has no unnecessary delay and B satisfies mutual exclusion then C has no unnecessary delay.**

This is in general not true. To see why, recall what it means for a critical section solution to satisfy absence of unecessary delay; if a process P is the only process which wants to enter the critical section (meaning, here, that it has started A's pre-protocol/is at p_2) then it will eventually be in its critical section. Here, this only guarantees we will reach p_3 without unnecessary delay since $p_3p_4p_5$ is the critical section from A's perspective. Hence, we may well be unnecessarily delayed upon reaching p_3 since mutual exclusion guarantees nothing about unnecessary delay.

(c) **If A satifies mutual exclusion and B has no unnecessary delay then C has no unnecessary delay.**

This is also not true in general. The reason why is very similar to the last question; here we can experience an unnecessary delay in p_2.

(d) **If A is deadlock free and B guarantees eventual entry then C guarantees eventual entry.**

This is in general not true since we can get stuck in a livelock in A's pre-protocol.

```
#include "critical2.h"

byte flag[3] = {0,0,0};

active[3] proctype P() {
  bit b = 0;
  do
  ::
L0: non_critical_section();
wap: flag[_pid] = 1;
L1:  /* Await FORALLj. flag[j] < 3 */
     flag[0] < 3;
     flag[1] < 3;
     flag[2] < 3;
L2:  flag[_pid] = 3;
L3:  /* if EXISTS j. flag[2] = 1 */
     b = 0;
     b = b || (flag[0] == 1);
     b = b || (flag[1] == 1);
     b = b || (flag[2] == 1);
     if
     :: b ->
L40:    flag[_pid] = 2;
        /*  await EXISTS j. flag[j] = 4 */
L41:    do
        :: b = 0;
           b = b || (flag[0] == 4);
           b = b || (flag[1] == 4);
           b = b || (flag[2] == 4);
           if
           :: b -> break;
           :: else -> skip;
           fi;
        od;
     :: else -> skip;
     fi;
L5:  flag[_pid] = 4;
     /* await FORALL j<i. flag[j] < 2 */
L6:  _pid == 0 || flag[0] < 2;
     _pid <= 1 || flag[1] < 2;
csp: critical_section();
     /* await FORALL j>i. flag[j] < 2 or flag[j] > 3 */
L7:  _pid >= 1 || flag[1] < 2 || flag[1] > 3;
     _pid == 2 || flag[2] < 2 || flag[2] > 3;
L8:  flag[_pid] = 0;
  od
}
```


Figure 1: Szymanski's Algorithm for three processes