CMU 17-651 Models of Software Systems Fall 2018

Homework #11: Concurrency Modeling and LTL

Dario A Lencina-Talarico Due: 7 November 2016

1. Exercises from MK06. (NOTE: For every question in which you are asked to use LTSA, you should include the relevant output of the checker and briefly explain those results.)

   (a) 5.4 (just the FSP part) The Dining Savages: A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot unless it is empty in which case he waits for the pot to be filled. If the pot is empty, the cook refills the pot with M servings. The behavior of the savages and the cook is described by:

   ```
   SAVAGE = (getserving -> SAVAGE).
   COOK   = (fillpot -> COOK).
   ```

   Model the behavior of the pot as an FSP process and then implement it as a Java monitor.

   ```
   const PotCapacity = 5
   range Int = 0..PotCapacity

   SAVAGE = (getserving -> SAVAGE).

   COOK   = (fillpot -> COOK).

   POT(I=0) = POTIMP[I],
   POTIMP[n:Int] = (when(n > 0) getserving -> POTIMP[n-1]
                    |when(n == 0) fillpot -> POTIMP[PotCapacity]
                    ).

   ||FEAST(N=5) = (forall [i:1..N] savage[i]:SAVAGE || COOK || POT(PotCapacity))
                  /{{savage[1..N]}.getserving/getserving}.
   ```

   (b) 6.2 One solution to the Dining Philosophers problem permits only four philosophers to sit down at the table at the same time. Specify a BUTLER process which, when composed with the model of section 6.2, permits a maximum of four philosophers to engage in the sitdown action before an arise action occurs. Show that this system is deadlock-free.

   ```
   const N = 5

   PHIL = (butler.requestSitdown->right.get->left.get
                  ->eat->left.put->right.put
                  -> butler.arise->PHIL).
   ```

```
FORK = (get -> put -> FORK).

BUTLER = BUTLER[0],
BUTLER[n:0..(N-1)] =(when (n<(N-1)) requestSitdown -> BUTLER[n+1]
|arise -> BUTLER[n-1]).

||DINERS =
        (forall [i:0..N-1] phil[i]:PHIL || {phil[i:0..(N - 1)]}::butler:BUTLER
        || forall [i:0..N-1] {phil[i].left,phil[((i-1)+N)%N].right}::FORK
        ).
```

(c) 6.4 It is possible for the following system to deadlock. Explain how this deadlock occurs and relate it to one of the four necessary and sufficient conditions for deadlock to occur.

```
Alice = (call.bob -> wait.chris -> Alice).
Bob   = (call.chris -> wait.alice -> Bob).
Chris = (call.alice -> wait.bob -> Chris).

||S = (Alice || Bob || Chris) /{call/wait}.
```

Given that the label rewrite maps all the calls to the wait prefixes, that means that the system can be rewriten as:

```
Alice = (bob -> chris -> Alice).
Bob   = (chris -> alice -> Bob).
Chris = (alice -> bob -> Chris).

||S = (Alice || Bob || Chris).
```

It can be seen that Alice tries to execute bob, but because Chris also defines an action called bob, then Alice gets stuck. Chris can not make any progress either as it is waiting for Bob to get to the alice action, and Bob is waiting for Alice to get to the chris action.

In my opinion, this is an example of "Wait-for-cycle": "a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire."[MK06]

The following model attempts to fix the problem by allowing Alice, Bob and Chris to time out from a call attempt. Is a deadlock still possible? If so, describe how the deadlock can occur and give an execution trace leading to the deadlock.

```
Alice = (call.bob -> wait.chris -> Alice
      |  timeout.alice -> wait.chris ->Alice).
Bob   = (call.chris -> wait.alice -> Bob
      |  timeout.bob -> wait.alice ->Bob).
Chris = (call.alice -> wait.bob -> Chris
      |  timeout.chris -> wait.bob ->Chris).

||S = (Alice || Bob || Chris) /{call/wait}.
```

Yes, it is possible to get a deadlock, using lts I was able to get the following trace:

```
Trace to DEADLOCK:
timeout.alice
timeout.bob
timeout.chris
```

using a sequence = $\langle timeout.alice, timeout.bob, timeout.chris \rangle$

(d) 7.1 What action trace violates the following safety property?

```
property PS = (a->(b->PS|a->PS)|b->a->PS).
```

$bad\_trace = \langle b, b \rangle$

(e) 7.2 A lift has a maximum capacity of ten people. In the model of the lift control system, passengers entering a lift are signaled by an enter action and passengers leaving the lift are signaled by an exit action. Specify a safety property in FSP which when composed with the lift will check that the system never allows the lift that it controls to have more than ten occupants.

```
const MAX_PASSENGERS = 100

range Passengers = 0..MAX_PASSENGERS

property RESPECT_OCCUPANCY(N=0) = RESPECT_OCCUPANCY_IMP[N],
RESPECT_OCCUPANCY_IMP[p:Passengers] = (when(p < 10) enter -> RESPECT_OCCUPANCY_IMP[p + 1]
                                      |exit -> RESPECT_OCCUPANCY_IMP[p - 1]
                                      ).
```

(f) 7.3 Specify a safety property for the car park problem of Chapter 5, which asserts that the car park does not overflow. Specify a progress property which asserts that cars eventually enter the car park. If car departure is lower priority than car arrival, does starvation occur?

```
const MAX_CAPACITY = 10

property RESPECT_MAX_CAPACITY(N=0) = MAX_CAPACITY_IMP[N],
MAX_CAPACITY_IMP[i:0..N] = (when(i < N) arrive -> MAX_CAPACITY_IMP[i + 1]
                           |when(i > 0) exit -> MAX_CAPACITY_IMP[i - 1]
                           ).

||CARPARK = (ARRIVALS||CARPARKCONTROL(MAX_CAPACITY)||DEPARTURES || RESPECT_MAX_CAPACITY(0)).
```

Progress property:

```
progress CAR_ARRIVE = {arrive}
```

After running LTSA progress check:

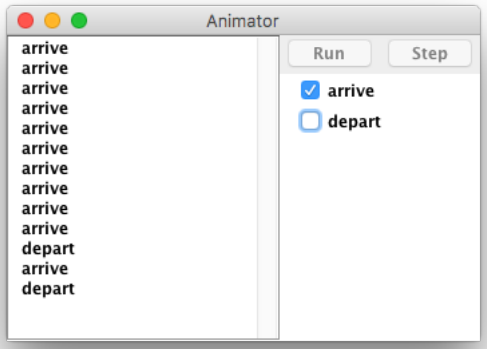
```
Progress Check...
-- States: 11 Transitions: 20 Memory used: 45837K
No progress violations detected.
Progress Check in: 0ms
```

After decreasing the priority of the depart event, I found that it is not possible for all vehicles to leave:

```
||CARPARK = (ARRIVALS||CARPARKCONTROL(MAX_CAPACITY)|| DEPARTURES
|| RESPECT_MAX_CAPACITY(0)) >> {depart}.
```

As it can be seen in the screenshot, the depart action is not available even when the parking lot is almost full.

I consider that to be a typical case of **starvation**.



2. Using the semantic rules for LTL covered in the lecture on that topic, expand the following temporal formulas to their equivalent semantic form (using the $(\sigma, i) \models P$) representation). For each, say what it means in English.

   (a) $Q \Rightarrow \Diamond\Box P$
   $(\sigma, 1) \models Q \Rightarrow \Diamond\Box P$
   if initially Q then eventually P will be true forever.

   Procedure: in lecture 19 it is explained what $\Diamond\Box$ means and how the $\Rightarrow$ operations functions as an if statement, I did not have much more procedure than that to attempt to answer this question.

   (b) $\Diamond(Q \wedge \Box P)$
   $(\sigma, 1) \models \Diamond(Q \wedge \Box P)$
   There is a state in the trace where initialy Q and always P are true

   Procedure: it is important to mention that the whole expression is wrapped by a $\Diamond$ operator, this is equivalent to an exists statement in predicate logic, then the expression inside says that initially Q and always P have to be true.

3. Translate each of the following into a formula of linear temporal logic. (NOTE: For this assignment, use only the temporal logic operators: $\Box$, $\Diamond$, $\bigcirc$, $\mathcal{U}$, $\mathcal{U}_w$.)

   (a) p is sometimes true and sometimes false.
   $\Diamond p \wedge \Diamond \neg p$

   (b) p and q are never true together.
   $\neg\Diamond(p \wedge q)$

   (c) q and r are both eventually false.
   $\neg\Diamond q \wedge \neg\Diamond r$

(d) q and r are eventually both false.
$\Diamond(\neg q \wedge \neg r)$

(e) If p is never true, then q is always true.
$\neg p \Rightarrow \Box q$

(f) Eventually, if p is true, then q is also true
$\Diamond(p \Rightarrow q)$

(g) If p is eventually true, q is never true.
$\Diamond p \Rightarrow \neg q$

(h) p is true finitely often (and at least once).
$\Diamond\Box p$

(i) If p is initially true, then it will be true infinitely often; otherwise it will never be true.
$p \Rightarrow \Box\Diamond p$

(j) p is initially true and thereafter alternates between being true and being false (i.e., if $(\sigma, i) \models p$, then $(\sigma, i+1) \models \neg p$ and if $(\sigma, i) \models \neg p$, then $(\sigma, i+1) \models p$).

$p \wedge \bigcirc\neg p$