CS 6110, Spring 2022, Assignment 2

Given 1/25/22 – Due 2/1/22 by 11:59 pm via your Github

NAME: Tripti Agarwal **UNID: u1319433**

1. **Justice and Compassion**

We start the x at 0 or 1. Then we set the init state to true. This means that the init state is set non-deterministically.

- **p1.pml** We now want to flip the next states of x. The LTL formula says eventually x implies henceforth x. There is a point at which x is true, but this not imply that x is always from from that point. Hence this gives an error shown:

```
State-vector 28 byte, depth reached 6, errors: 1
        4 states, stored
        0 states, matched
        4 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

- **p2.pml** Here the LTL formula is for henceforth x implies eventually x. This means at some point i, x is true and remains true. This does imply that x is eventually true at point i. Hence we end up getting no error.

```
State-vector 28 byte, depth reached 7, errors: 0
        5 states, stored
        4 states, matched
        9 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

- **p3.pml Justice:** (<>[]x -> []<>y) i.e. eventually-henceforth x then infinitely-often y. **Compassion:** ([]<>x -> []<>y) i.e. infinitely-often x then infinitely-often y In p3, it is **compassion implies justice**. As compassion is strong fairness and it make sense that if we infinitely often perform an action then infinitely often other action will happen. This does definitely imply that at some point an action is true then we are performing infinitely often other action.

```
State-vector 28 byte, depth reached 14, errors: 0
       93 states, stored
      237 states, matched
      330 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

- **p4.pml** In p4, the LTL formula is for **justice imply compassion**. As we can see justice means weak fairness, and it is eventually henceforth an action then infinitely often other action will happen. This does not imply, that if we infinitely often keep doing an action then infinitely often other action will also happen. Hence we end up getting an error.

```
State-vector 28 byte, depth reached 25, errors: 1
       24 states, stored
       45 states, matched
       69 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

- **Justice imply compassion?** From example p4.pml it is clear that justice does not imply compassion. **Compassion imply justice?** From example p3.pml it is clear that compassion imply justice.

- **Pnueli's paper's title true?**

- (a) The solicitor is walking around in round robin fashion and knocking at each door. Since the home-owners are found to be infinitely-often opening and closing, therefore this is **strong fairness** or **compassionate**. Here τ (solicitor knocking at the door) is infinitely often enabled and the home owners are bound to open and then close the door infinitely many times.

  (b) Since the solicitor walks at one door and keep on pressing the bell till it melts, i.e this transition is continuously enabled and then the home owner infinitely-often opens the door, therefore in this case the home owners have **weak fairness** or **justice**.

  (c) If the solicitor is a network packet and the home owner doors are ports, then in first case (a), the network packet reaches every port and ask whether the port is open and repeats this for all the ports and hence this is called compassion. Whereas in case (b), the packet comes at one port and waits there till the time the port is open and this is called justice.

2. **LTL for sa and sb kripke structure**

(a) After changing the never automata with the corresponding LTL formula we can see that Sb gives no erro with both the LTLs. This is because Sb is true for both and Sa works only with second LTL. Proofs for the above are provided in the snipet shown below.

```
ltl p1 {( !(<>([](a && b))) -> ((<>([]a)) || (<> (!  a && !  b))))}

ltl p2 {( !(<>([](a && b))) -> ( (<>([]a)) || (<> (b)) ) )}
```

```
State-vector 36 byte, depth reached 21, errors: 1
       11 states, stored
        1 states, matched
       12 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

(a) Sa with LTL p1

```
State-vector 28 byte, depth reached 0, errors: 0
        1 states, stored
        0 states, matched
        1 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

(b) Sa with LTL p2

```
State-vector 36 byte, depth reached 20, errors: 0
       19 states, stored (29 visited)
       20 states, matched
       49 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

(c) Sb with LTL p1

```
State-vector 28 byte, depth reached 0, errors: 0
        1 states, stored
        0 states, matched
        1 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

(d) Sb with LTL p2

(b) i. **Does sa implies sb?** No sa do not imply sb. This can be observed from the error traces obtained from LTL formulas. We can see the LTL formula p1 can result in an error for sa and not for sb. Similiary for p2 both sa and sb satisfies. This means, the kripke structure of sa cannot give full information of sb. Hence LTL of sa do no imply sb.

ii. **Does LTL of sb implies sa?** Yes, sb implies sa. This can be observed from LTL p1 and p2. We see p2 satisfies both sa and sb, but p1 only satisfies sb. This means the the LTL formula for sb can imply LTL formula for sa.

3. **Bubble sort**

- **Fixed broken bubble sort** The bubble sort code was broken at line 31 and 42 of the code. The value of t was assigned to a[1] whereas it should be assigned as 1. Also, the code is further enhanced, as the bubble sort is sorting the maximum value first and putting it at the end. The code provided in the github repo, is an optimized version where the loop is decreased by 1 every time the maximum element is pushed to end.

```
byte k;
k=aMaxIndx;
do /* Subsequent ''repeat'' iterations */
:: k!=1 ->
   t=a[aMinIndx]; j=aMinIndx+1;
   k--
   do
   :: (j > (k)) -> break /*-- For-loop exits --*/
   :: (j<=(k)) ->
     if
     :: (a[j-1] > a[j]) -> t=a[j-1]; a[j-1]=a[j]; a[j]=t
:: (a[j-1] <= a[j])
     fi;
     j++ /*-- for-index increments --*/
   od  /*-- end of for-loop --*/
:: k==1 -> break
```

- **Evidence that the code has been checked exhaustively**
  - *Ran the code for depth 10, no error obtained*

```
State-vector 20 byte, depth reached 9, errors: 0
      118 states, stored
       65 states, matched
      183 transitions (= stored+matched)
        0 atomic steps
```

  - *Ran the code for depth 20, no error obtained*

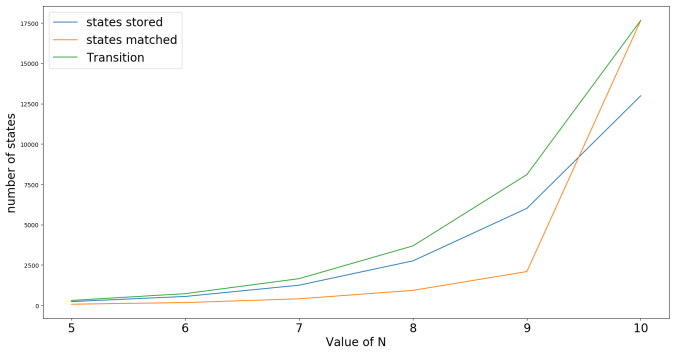
```
State-vector 20 byte, depth reached 19, errors: 0
      201 states, stored
       65 states, matched
      266 transitions (= stored+matched)
        0 atomic steps
```

  - *Ran the code for depth 30, no error obtained*

The maximum depth was till 27 and still we did not get any error. Hence we have ran the code exhaustively to check that the code is now correct.

```
State-vector 20 byte, depth reached 27, errors: 0
      233 states, stored
       70 states, matched
      303 transitions (= stored+matched)
        0 atomic steps
```

- **Increasing the size of array**



4. **Dinning Philosopher's code**

The code is ran exhaustively to obtain following traces. The code works fine without any error.

```
State-vector 160 byte, depth reached 9, errors: 0
        5 states, stored
        4 states, matched
        9 transitions (= stored+matched)
       10 atomic steps
hash conflicts:         0 (resolved)
```

(a) Error trace for depth 10

```
State-vector 160 byte, depth reached 19, errors: 0
       13 states, stored
        8 states, matched
       21 transitions (= stored+matched)
       10 atomic steps
hash conflicts:         0 (resolved)
```

(b) Error trace for depth 20

```
State-vector 160 byte, depth reached 29, errors: 0
       19 states, stored
        9 states, matched
       28 transitions (= stored+matched)
       10 atomic steps
hash conflicts:         0 (resolved)
```

(c) Error trace for depth 30

```
State-vector 160 byte, depth reached 34, errors: 0
       21 states, stored
        9 states, matched
       30 transitions (= stored+matched)
       10 atomic steps
hash conflicts:         0 (resolved)
```

(d) Error trace for depth 40

5. **DT.pml**

The code with the changes are available on Github. Here is the change in the code for the work assignment from any node to upstream root node. The code did not had the case when we pick the root node. The code is also changed for the random initialization given in the question.

```
:: ns[myid] == A ->
            if
            :: ns[myid] = P //--R03
            :: do
               :: (pick == 0  )   -> pick // pick place is root
               :: (pick > 0   )   -> pick--
               :: (pick < Ns-1)   -> pick++
               :: (pick != myid) -> break  // pick place to assign work
               od;
            workqArray[pick]!WORK ;
            if
          :: pick > myid -> nc=B //--R04, upstream
          :: pick == 0 -> nc=B  //upstream to root
          :: pick < myid         //--R05, downstream
          fi;
            ns[myid] = P //--Common to R04 and R05
            :: tokIn?HasT   //--R06
            fi
```

The code is run exhaustively for different depth. For depth of 50000, we get 43215 with no error as shown in the figure below.

```
State-vector 132 byte, depth reached 43215, errors: 0
   324984 states, stored
   824795 states, matched
  1149779 transitions (= stored+matched)
       28 atomic steps
hash conflicts:      4301 (resolved)
```

Plot for different values of N for states stored, matched and no of transition states.