CS 6110, Spring 2022, Assignment 2

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

NAME: Jack Wilburn UNID: u0999308

CHANGES: Please look for lines beginning with underlined words.

Answering, Submission: Please provide a PDF with these questions answered in the spaces indicated (if you don't retain the frameboxes, at least please begin each answer on a new page with the question numbers/parts indicated). Also have your work ready on your Github for me to pull/test. Note that Asg2 will be gone over in class and even partially worked out; you'll be finishing the unfinished parts and submitting the full solution. Orientation videos and further help will be available (drop a note anytime on Piazza for help)—**start early**.

1. (20 points) Read about LTL from CEATL's Chapter 22. (You may also look at Ben-Ari's Chapter 5.) By way of practice, in the directory `Lec5` within the class Git `https://github.com/ganeshutah/cs6110s22.git`, you have been given four Promela files `p1.pml` through `p4.pml`. Run these for your own understanding and repeat the results at the end of these files (if any). In particular, in `p4.pml`, you are checking whether *Justice* implies *Compassion*. These terms are defined in the paper "All you need is Compassion" `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.124.3239` (more properly `https://dl.acm.org/doi/10.5555/1787526.1787547`). Does Justice imply Compassion? Does Compassion imply Justice? Based on your observations (please provide terminal sessions showing the error trace(s) that substantiate your conclusions), do you agree that Pnueli's paper's title is true? Also answer these:

   (a) A solicitor walks around the neighborhood knocking on one door, then the other, taking a "round-robin" walk, and the home-owners are found to be infinitely-often opening and closing their doors in response. Are the home-owners being just or compassionate? Why? (justice is also known as *weak fairness*, while compassionate is also known as *strong fairness*).

   (b) A solicitor walks to one door, leans on the bell and rings the bell continuously (till it almost melts). The home-owner infinitely-often opens the door. Now are the home-owners being just or compassionate? Why?

   (c) Repeat the above answers assuming that the solicitor is a network packet and home-owner doors are ports of a switch.

   This is an invitation for you to think about such a router—or if you have seen a more realistic situation like this, plz write in your answer.

p1.pml shows an error. It shows that this model doesn't satisfy [](initstate -> (<>x -> []x)), thus when the initalstate is true, eventually x doesn't imply always x. This is clear from the promela since the value of x can be flipped back

p2.pml doesn't show an error, but it stops after 7 steps. I presume it stops because it sees it's checking for always x implies eventually x. Since x is flipping back and forth, it's not always x.

p3.pml doesn't show an error either. (Always eventually x implies always eventually y) implies (eventually always x implies always eventually y). Compassion implies justice. Since compassion implies justice, I agree with the title of the paper.

p4.pml shows an error. Flipping the outer implies clauses from pm3 does not hold. Justice does not imply compassion.

p4.pml error trace after block.

a) The home owners are being compassionate. They are answering the door infinitely for infite rings. The rings are spaced out, and not happening as if the person was leaning on the bell.

b) The home owners are being just. The solicitor is doing <>[]p. Since that's the antecedant, it must be justice.

c) A router is being compassionate by answering each packet that comes in with delays, whereas it's just if it's answering packets that are constantly hammering the port.

```
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)

<<<<<START OF CYCLE>>>>>
MSC: ~G 21
15: proc  0 (ltl0) p4.pml:21 (state 39) [(!(y))]
16: proc  1 (p) p4.pml:14 (state 12) [x = x]
MSC: ~G 80
17: proc  0 (ltl0) p4.pml:80 (state 132) [((!(x)&&!(y)))]
18: proc  1 (p) p4.pml:14 (state 12) [x = x]
MSC: ~G 54
19: proc  0 (ltl0) p4.pml:54 (state 90) [(!(y))]
20: proc  1 (p) p4.pml:14 (state 12) [x = !(x)]
21: proc  0 (ltl0) p4.pml:54 (state 90) [((!(y)&&x))]
22: proc  1 (p) p4.pml:14 (state 12) [x = x]
MSC: ~G 28
23: proc  0 (ltl0) p4.pml:28 (state 50) [(!(y))]
24: proc  1 (p) p4.pml:14 (state 12) [x = !(x)]
25: proc  0 (ltl0) p4.pml:28 (state 50) [((!(x)&&!(y)))]
26: proc  1 (p) p4.pml:14 (state 12) [x = x]
spin: trail ends after 26 steps
#processes 2:
26: proc 0 (ltl0)  p4.pml:21 (state 39) (invalid end state)
((!(x)&&!(y)))
((!(y)&&x))
((!(y)&&x))
(!(y))
26: proc 1 (p)  p4.pml:14 (state 12) (invalid end state)
```

```
x = x
y = y
x = !(x)
y = !(y)
global vars:
bit    x: 0
bit    y: 0
bit    ready: 1
```

2. (20 points) Now turn to Page 419 of CEATL and look at questions 22.3 and 22.5. In 22.5, a Promela model and `never` automata that distinguish 22.3(a) and 22.3(b) are given.

   (a) Remove the `never` and replace it with an equivalent LTL. For instance, in the code we have `never !(foo)` and here, the LTL being checked is `foo`. Make sure that my claimed checks work (the first formula is true of `sb` but not `sa`, while the second formula is true of both structures; notice that each diagram that warrants being a Kripke structure has been given the name `sa` through `sh`).

   (b) In the earlier question, a method to check the validity of LTL formulas using the most general Kripke Structure was introduced. Using that method, answer these questions, with evidence furnished.

   We will call this "LTL of sa":

   ```
   !(<>([](a && b))) -> ((<>([]a)) || (<> (!a && !b)))
   ```

   This can be read as "[Not eventually-henceforth (a and b)] IMPLIES [either (eventually-henceforth a) or eventually (!a and !b)]"

   We call this "LTL of sa" because on Page 420, we have this (in a never automation, the LTL is given one more negation):

   ```
   /* sb satisfies this, and not sa */
   /*Type 'spin -f "formula"' and cut&paste the resulting never aut. */
   /*--------------------------------------------------------------*/
   never {/* !( !(<>([](a && b))) -> ((<>([]a)) || (<> (!a && !b)))) */
   ```

   We will call this "LTL of sb":

   ```
   !(<>([](a && b))) -> ( (<>([]a)) || (<> (b)) )
   ```

   This can be read as "[Not eventually-henceforth (a and b)] IMPLIES [either (eventually-henceforth a) or eventually b]"

   This is because on Page 421, we have this (in a never automation, the LTL is given one more negation):

   ```
   /*--- in contrast, both sa and sb satisfy this --->
    never {  /* !( !(<>([](a && b))) -> ( (<>([]a)) || (<> (b)) ) ) * /}
    <---*/
   ```

   i. Does the LTL of `sa` implies that of `sb`? Provide reasons after observing the trace from your experiment.

      This is a validity-checking situation, so you must invoke the most general Kripke-structure here.

   ii. Does the LTL of `sb` implies that of `sa`? Provide reasons after observing the trace from your experiment.

      This is a validity-checking situation, so you must invoke the most general Kripke-structure here.

For part a), see q2.pml in my repo.
The "LTL of sa" does not imply that of sb, because the LTL of sa is not satisfied by sb.
Conversely, the "LTL of sb" does imply that of sa since sa satisfies the LTL.

3. (20 points) Fix the broken Bubble-sort, either through a mild repair or a wholesale rewrite. Verify (upto the limits of finite-state model checking) that it works. Submit evidence that you ran exhaustively (possible for this simple a model). For fun, increase the array size from its current value in steps of 2 (or 1) and do a plot of the number of states generated, revisited, and matched. This tells you how many states are typically generated and how these grow. In symbolic (SAT/SMT-based model-checking), the states are not in a hash-table and the representation grows differently. In fact, a BDD-based hash-table has a size of 1 when it is empty as well as when it is full! These are K-layer DFAs basically. See `https://spinroot.com/gerard/pdf/sttt98.pdf`.

While horrendously inefficient, using n squared time (guaranteed), I fixed the algorithm by iterating through the whole array n times. That means that the correct values bubble up for sure. In the pseudo-code, this looks like replacing repeat with a for loop from i = 1 to N. By being so inefficient, we are guaranteed that every value that needs to will bubble up.
In promela, it looks like replacing the check for :: t==a[1] -> break with :: (i > aMaxIndx) -> break, incrementing i on each loop round.
I tested that this worked with 1000 iterations of 100,000 steps. There were no assertion errors. Try "grep 'assertion violated' q3.out "
See q3.pml for my modified promela code.

4. (20 points) In the Philosophers example, the example suffered from “livelock” (all of them take the left, get nacked for the right, and repeat). One way to avoid such livelocks is to break symmetry (make one philosopher reach for the right fork first). Implement this solution. Now show that the system is lock-free but not wait-free (see `https://en.wikipedia.org/wiki/Non-blockingalgorithm` for these definitions). Lock-free is *communal progress* and wait-free is *individual progress*.
   - old instructions:
     - Use LTL assertions (not `never` automata).
     - Submit your work accompanied by traces and helpful observations.
   - <u>new instructions:</u>
     - Given the fix I posed on piazza (see `dp.pml` pushed in)
     - Understand the fix (the LTL used). Try it to make sure it works. Make any suggestions on improvement

I can’t find the inital starting code for this question. It seems like the right solution is implemented in dp$_c ontrarian.pml, wherethereare3philosphersandonetriestograbright, while2trytogrableft.Thisshowsthe$
To show that it’s lock-free, but not wait free, we can consider a few examples.
In the example of symmetry, all philosphers could grab left and then be dead locked. No philospher can grab right, and if it’s not coded well, none will release. This freezes the program. This is not lock free and is not wait free, because no progress is made.
In the case of asymmetry with releasing the forks, the philosphers will try to grab whichever side they prefer and then they will release if they can’t complete the set of forks they need (just the 2 from each side). This blocks their individual process (i.e. it’s not wait-free), but by releasing they allow communal progress (i.e. it’s lock-free) by letting another philosopher try to complete their set of forks.

5. (20 points) Follow-along and finish the design of Dijkstra's distributed termination algorithm. You'll be asked to type these with me in class live, and finish. `http://people.cs.aau.dk/~adavid/teaching/MVP-10/17-Termination-lect14.pdf` and `https://www.cs.rochester.edu/u/sree/courses/csc-258/spring-2018/slides/22-td.pdf` do a good job of providing slides that I'll go thru in class. The original is `https://www.cs.utexas.edu/~EWD/ewd08xx/EWD840.PDF` from Dijkstra's collection `https://www.cs.utexas.edu/~EWD/indexBibTeX.html`. Look for files in the directory `Lec5`. Come Thu to type this fully and check it!

   ADDENDUM:

   (a) `DT.pml` is the initial template we began with

   (b) `DT.pml` is the latest version we ended up with

   (c) Your task is:

      i. Begin with `DT.pml` or your latest good solution

      ii. Attempt a random initialization of the A/P status (see code below). While the original Dijkstra implementation started with the root alone in "A," with the fix suggested below, all initializations are OK—which is a nice generalization!

      iii. **Fixed problem:** If you did not get any errors, then no fix needed! But, please plot the state-space growth for N=4 (if it finishes) and N=5 (if it finishes). Just give it about 15 min max and see if it finishes. That is all (no need to burn up your time and electricity).

      To tell you the back-story, I ran into a fix, then I thought the "upstream logic" was wrong. But now I can't reproduce the error. So perhaps this is after all a fixed protocol.

      iv. Fix enough about the protocol's implementation (see Piazza discussion). The hint is to think about how "upstream" must be implemented. **As if you want more hints, but as a private Piazza post!**

      v. Then run the protocol verification!

   Detail of random initialization:

```
init {
byte i = Ns-1;
        atomic {
        do
        :: i > 0 ->
           run node(tokqArray[i], tokqArray[i-1], workqArray[i], i);
   if //-- nondet initialization of initial state
   :: ns[i] = A
   :: ns[i] = P
   fi;
           i-- ;

        :: i == 0 ->
   run node(tokqArray[0], tokqArray[Ns-1], workqArray[i], i);
   if //-- nondet initialization of initial state
   :: ns[i] = A
   :: ns[i] = P
   fi;
   break
        od
        }
}
```

i.) Finished solution in q5-1.pml
ii.) Randomization is in q5-2.pml
iii.) For state growth, I used the static assignment, just the root node as the assigned member. I got 122 states for 3, 296 for 4, and 706 for 5. This appears pretty close to quadratic time state space size increases.
iv.) I've attempted to implement this in q5-2.pml, but I seem to be having an issue with the white black coloring after adding your init code.
v.) I've been running this a ton in spin, is this what you meant by verification?