CS 6110, Spring 2022, Assignment 4

Given 2/11/22 – Due 2/18/22 by 11:59 pm via your Github

NAME: Jack Wilburn UNID: u0999308

CHANGES: Please look for lines beginning with underlined words when they are made. none yet.

Answering, Submission: Have these on your private Github: a folder Asg4/ containing your submission, which in detail comprises:

- A clear README.md describing your files.
- Files that you ran + documentation (can be integrated in one place).
- A high level summary of your cool findings + insights + learning – briefly reported in a nicely bulletted fashion in your PDF submission.

Start Early, Ask Often! Orientation videos and further help will be available (drop a note anytime on Piazza for help). *I encourage students constructing answers jointly!*

1. (50 points - 25 for pre and 25 for partial - Alloy) To learn Alloy, you can get a PDF copy of the **older** edition of the book by Daniel Jackson called "Software Abstractions." I've found a PDF by searching for the above. Since this is a very old edition, this is probably OK. Other tutorials are `http://alloy4fun.inesctec.pt/` `https://haslab.github.io/formal-software-design/overview/index.html` `https://www.cs.montana.edu/courses/se422/currentLectures/AlloyIntro.pdf` and `https://alloytools.org/tutorials/online/`. Start reading through this book and also Roger Costello's slides on the class Github.

   We have to understand mathematical relations properly before we can use Alloy. Read my chapter in CEATL on relations (the chapter featured in the tutorial I recorded). This is CS 2100 material—so, leaving it for your self-study.

   Here are some experiments I ran to study preorders and partial orders.

   TL;DR *The intersection of a preorder and its inverse is not an identity relation.*

   TL;DR *The intersection of a partial order and its inverse is an identity relation.*

   These are the conclusions to be drawn in the experiments to follow.

   Your task is to fill out the ellipsed portions (where I provide an English phrase for you to fill as `<text>`) and answer the questions below (questions begin with "Q:" and comments by "C:"):

```
-- C: We are defining "some old" relation 'pre'
-- C: nd slowly endowing it with the properties that make it a preorder
--
sig S { pre  :  set S } -- (1) C: This defines 'pre' as a binary relation over S
fact  { some pre }      -- (2) Q: Can you explain what this means?
fact  { <state here that pre is reflexive>  } -- (4) Q: answer the question below
fact  { <state here that pre is transitive> } -- (5) Q: answer the question below
assert preAndPreinvIden
 { <FALSELY Assert that the intersection of pre and its inverse is identity.> }
  -- (6) Q: answer the question below
check preAndPreinvIden for exactly 3 S
```

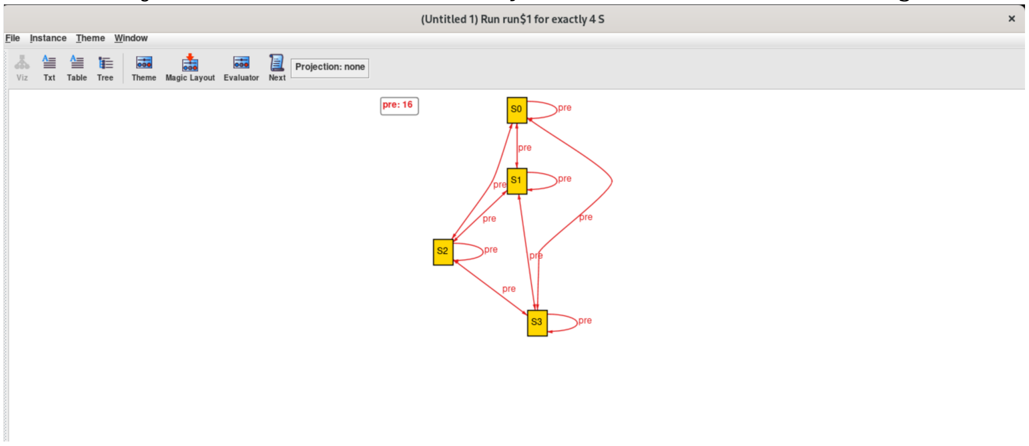
```
  -- (7) Q: Report on the result of this check
run {} for exactly 4 S
  -- (8) Q: If the check in (7) fails, comment (7) and run this to diagnose why
```

(1) `pre` is introduced as a "plain old" relation

(2) Explain what this assertion does for `pre`

- This asserts that there are 1 or more pre relations on S.

(3) At this juncture, "run" for "`exactly 4 S`" and show the models generated.



(4) How did you endow 'pre' with the property that it is reflexive? Explain.

I used the iden in pre trick to make sure the relationship has an identity for every element of S.

(5) Explain how you endowed 'pre' to be transitive

To make pre transitive, I set up a chain of s -> t -> u and said that implies s -> u. This was for all s in S.

(6) How did you falsely assert that the intersection of pre and its inverse is identity?

Using the suggestion from the lecture, I used pre pre = iden.

(7) Did this check pass? You can use a pull-down menu and run this check. If the check in (7) failed, look at the counterexample and explain why it failed.

The check failed. It failed because it created a fully connected graph of 3 nodes and when you flip all the connections (pre), you get the same fully connected graph.

(8) Also run this "run" statement and see the instance generated. Does the instance generated provide another explanation as to why the above `check` failed?

The problem is that if you have a bi-directional edge, that will show up in pre pre. For the assertion to hold, that would need to be explicitly removed from the relationship.

(9) Now, define a *partial order* along the same lines as below. Instead of the assertion `preAndPreinvIden`, define `partAndPartinvIden`, where we changed "pre" to "part" (partial order). Did this check pass? Justify the answer.

This check did pass. Since the only issue above were bi-directional edges, enforcing antisymmetry removes them and there are no counter examples where pre and not pre is not the identity.

My code:

```
sig S { pre  :  set S } -- (1) C: This defines 'pre' as a binary relation over S
```

```
fact  { some pre }      -- (2) Q: Can you explain what this means?
fact  { S <: iden in pre } -- (4) Q: answer the question below
fact  { all s,t,u: S | s->t in pre and t->u in pre implies s->u in pre } -- (5) Q: answ
assert preAndPreinvIden
 { pre & ~pre in iden}
  -- (6) Q: answer the question below
check preAndPreinvIden for exactly 3 S
  -- (7) Q: Report on the result of this check
run {} for exactly 4 S
  -- (8) Q: If the check in (7) fails, comment (7) and run this to diagnose why


  -- For partial order, I kept the name pre, but added antisymmetry
  fact {all x,y: S |x->y in pre and y->x in pre implies x=y} --antisymmetry
assert preAndPreinvIden
 { (pre & ~pre) in iden}

check preAndPreinvIden for exactly 3 S
```

2. (10 points, Store Buffer) Run the store-buffer example created in Promela. Argue that it simulates the situation of the writes being buffered before it goes to memory (as in TSO which is described at `https://en.wikipedia.org/wiki/Memory_ordering` and elsewhere). Observe the assert failure and explain the interleaving (via an error trace) causing the bug. (The file in question is `Peterson_tso.prm`.)

The variable definitions up top, give us 2 arrays of 2 bools for interest. These arrays allow the code to simulate writing to a buffer, where the buffer would store the interest for each process. There's access time involved in these lookups and the resulting context switches of this program simulate the different scheduling and time it takes to access the store.
The promela code asserts that only one thread gets into the critical section, but this is not a guarantee from the model since local interest may be out of sync with the global interest.

```
257: proc  1 (user:1) peterson_tso.pml:18 (state 3) [turn = _pid]
258: proc  1 (user:1) peterson_tso.pml:20 (state 4) [printf('Lintrst[%d] = 1\\n',_pid)]
258: proc  1 (user:1) peterson_tso.pml:21 (state 5) [printf('turn = %d\\n',_pid)]
259: proc  0 (user:1) peterson_tso.pml:33 (state 12) [(((0==Tintrst)||(turn==(1-_pid)))
260: proc  1 (user:1) peterson_tso.pml:26 (state 8) [Tintrst = Lintrst[(1-_pid)]]
Whee, PID 0 reached CS
261: proc  0 (user:1) peterson_tso.pml:35 (state 13) [printf('Whee, PID %d reached CS\\
262: proc  0 (user:1) peterson_tso.pml:37 (state 14) [ncrit = (ncrit+1)]
263: proc  0 (user:1) peterson_tso.pml:38 (state 15) [assert((ncrit==1))]
264: proc  0 (user:1) peterson_tso.pml:39 (state 16) [ncrit = (ncrit-1)]
265: proc  0 (user:1) peterson_tso.pml:41 (state 17) [Lintrst[_pid] = 0]
266: proc  2 (copier:1) peterson_tso.pml:51 (state 1) [intrst[0] = Lintrst[0]]
267: proc  0 (user:1) peterson_tso.pml:17 (state 2) [Lintrst[_pid] = 1]
268: proc  2 (copier:1) peterson_tso.pml:51 (state 1) [intrst[0] = Lintrst[0]]
269: proc  0 (user:1) peterson_tso.pml:18 (state 3) [turn = _pid]
270: proc  0 (user:1) peterson_tso.pml:20 (state 4) [printf('Lintrst[%d] = 1\\n',_pid)]
270: proc  0 (user:1) peterson_tso.pml:21 (state 5) [printf('turn = %d\\n',_pid)]
271: proc  0 (user:1) peterson_tso.pml:24 (state 6) [Tintrst = intrst[(1-_pid)]]
272: proc  0 (user:1) peterson_tso.pml:25 (state 7) [printf('Tintrst = intrst[1 - %d]\\
273: proc  2 (copier:1) peterson_tso.pml:52 (state 2) [intrst[1] = Lintrst[1]]
274: proc  0 (user:1) peterson_tso.pml:33 (state 12) [(((0==Tintrst)||(turn==(1-_pid)))
275: proc  1 (user:1) peterson_tso.pml:27 (state 9) [printf('Tintrst = Lintrst[1 - %d]\
Whee, PID 0 reached CS
276: proc  0 (user:1) peterson_tso.pml:35 (state 13) [printf('Whee, PID %d reached CS\\
277: proc  1 (user:1) peterson_tso.pml:33 (state 12) [(((0==Tintrst)||(turn==(1-_pid)))
Whee, PID 1 reached CS
278: proc  1 (user:1) peterson_tso.pml:35 (state 13) [printf('Whee, PID %d reached CS\\
279: proc  1 (user:1) peterson_tso.pml:37 (state 14) [ncrit = (ncrit+1)]
280: proc  0 (user:1) peterson_tso.pml:37 (state 14) [ncrit = (ncrit+1)]
spin: peterson_tso.pml:38, Error: assertion violated
spin: text of failed assertion: assert((ncrit==1))
```

3. (10 points, The Java example with volatiles) Read-up on Java volatiles. Run the example `VBad.java`. Insert volatiles selectively (just for req or just for ack). Does that correct the apparent hang? (I don't know the answer but thought you'd like to try.) To get the apparent hang, first you must leave out the volatile totally and get the hangs on your machine. Then *explain the reason for this hang.* (Why might it be happening? What reordering in the protocol can cause it to change. Assume only store/load reorderings.[1]) *Then* try to add one volatile and see if you get a hang. Explain your observations. (Bound your empirical testing to say an hour.)

   Using N = 50000, I was able to see it hang with both volatiles missing, or with either req or ack missing a volatile.
   Without the volatiles, the program hangs because of a data race and reordering of instructions. One thread is able to skip by one of the handoffs without waiting for the other thread and the instructions execute in such a way that the program is stuck in while loops with no way to proceed.

[1]All attempts to read the generated code failed. We assume there is no advantage gained by the compiler reordering such a short program's instructions. Thus it must be the hardware store-buffer and/or cache of the processor.

4. (10 points, the Man-Wolf-Goat-Cabbage or mwgc game) Write a pseudo-code for a DFS model-checker by modifying the BFS model-checker's pseudo-code here `https://www.cs.utah.edu/~kirby/Publications/Kirby-33.pdf`. Do you now understand why a model-checker does not "infinitely loop?" For definiteness, assume that the model-checker has an example as follows and follow a BFS strategy to draw out its state-space, and then a DFS strategy. Assume `p1` is run before `p2`. (Just to reduce ambiguity, I'll tell you the obvious: in the DFS "left-to-right" order; thus "depth exploration" with respect to `p1` will finish it before touching anything in `p2`.)

```
bit x;
active proctype p1()
{do
 :: x++ ; x++
 od
}
active proctype p2()
{do
 :: x-- ; x--
 od
}
```

   (a) In mwgc, **change the definition of safe** to something else (obtaine from a completely different perspective). Show that these expressions (the one I wrote and the one you wrote) are equivalent, using a BDD. **This is SIMPLE but CRUCIAL**: once we had to chase down a bug in an *expert*'s code—and after a month we found it was due to a predicate in a switch statement! *Such bugs are best prevented than detected later or let it kill someone or destroy something*—if we can prevent it. So make it a practice to use BDD/SAT tools to verify conditions you write in production code.

   Once shown equivalent (or you argue that your definition of *safe* is better), proceed with the following steps.

   (b) Now, run the murphi model I wrote today for `mwgc` (you should be able to recreate it from memory or the recording) and run it in DFS.

Does the error-trace (winning sequence) get longer? Can you go after longer error traces (by not stopping at the first error)? Try to produce a longer error trace than with DFS. Describe that "winnning sequence" (error trace for the negated invariant).

I used rumur to run the models, instead of cmurphi.
To replace your safe check, I used (((m = g) | (g != c)) ((m = w) | (w != g))). This states that the man must be with the goat or the goat must not be with the cabbage and that the man must be with the wolf or the wolf must be not with the goat.
Checking with the BDD.py file from the earlier weeks, mine -> give gives just a one node. That means that they're equisat and satisfying my equation also satisfies the given equation.
I can't find the options in rumur to run the model in dfs vs bfs, so I'll speculate what I expect to happen. I think you'll be able to find stack traces of any length if you allow it to keep searching andn skipping the first error message it sees. However, if it doesn't allow cycles, it would be impossible to find steps that are unique after a certain point so there would be a maximum length.
The winning trace for my most recent rumur run is below. In summary it's:
Move m + g
Move m
Move m + w
Move m + g
Move m + c
Move m
Move m + g

```
Startstate 1 fired.
m:false
w:false
g:false
c:false
----------

Rule "M and G move safely" fired.
m:true
g:true
----------

Rule "M moves safely" fired.
m:false
----------

Rule "M and W move safely" fired.
m:true
w:true
----------

Rule "M and G move safely" fired.
m:false
g:false
```

```
----------

Rule "M and C move safely" fired.
m:true
c:true
----------

Rule "M moves safely" fired.
m:false
----------

Rule "M and G move safely" fired.
m:true
g:true
----------
```

5. (20 points, DCL) Read Pugh's analysis of the Java Memory Model through the URL `https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html` and describe what weak memory ordering issues are discussed there. In neat bullets, list all the memory ordering issues that cause the double-checked locking idiom to fail. (This is a classic website that sowed the whole area of understanding weak memory models; Java was supposed to be a "safe" language, but with weak memory, you could export an object reference before initializing the object—thus leaking a secret that you did not wipe out.)

• The webpage describes double checked locking and the ways it fails in Java, even with additional modifications to try fix the issue.
• The first issue is that the instantiation of the Helper object happens before the write to the variable tracking it. Thus the var is null, even though the object has been created. The check passes and two helpers are made
• The next suggested fix is to add another synchronization requirement around the if statements, but that fails for the same reason as above. The assignment to helper is not guarded thoroughly enough.
• There's a suggestion to use bidirecitonal memory barriers, but that is inefficient and fails. It doesn't work because some processors store local, stale caches that might contain an old value for the variable of interest.
• The solution is to make the method syncronized or to use a different representation of the variable
• The same issue arises with floats and longs, but not 32 bit integers, because the number of instructions mean reordering is possible.
• You can also get around the issue by using volatile.