CS 6110, Spring 2022, Assignment 4

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

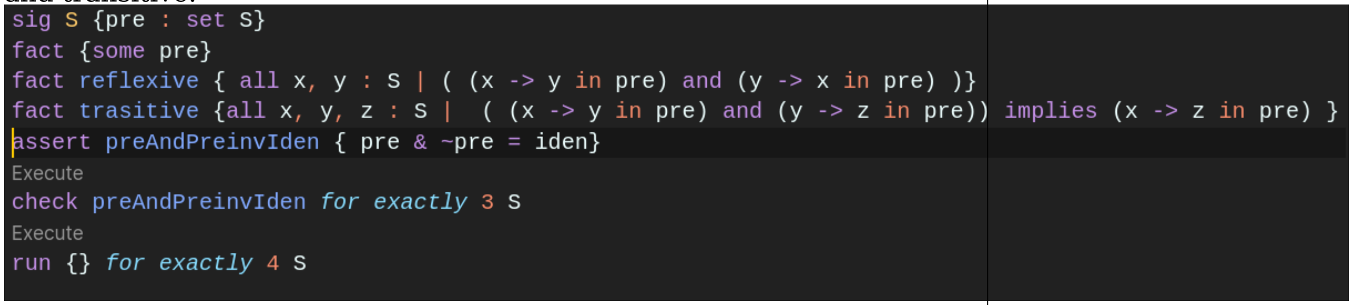
NAME: Tripti Agarwal **UNID: u1319433**

1. **(50 points - 25 for pre and 25 for partial - Alloy)**

(a) sig S is a signature where pre is a binary relation over S.

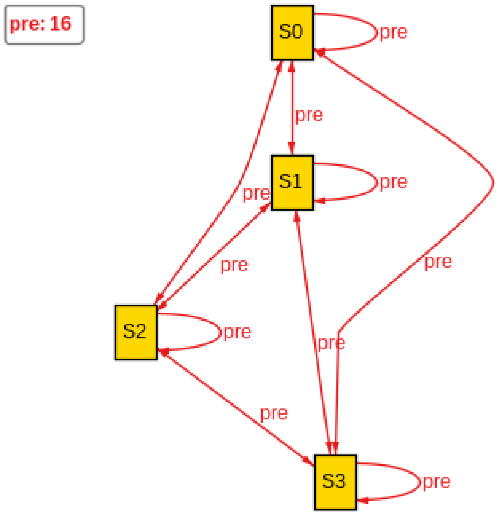
(b) Different models are generated at this point which have one or more relations with S.

(c) **Preorder facts and output** Pre order is the order such that it is reflexive and transitive.

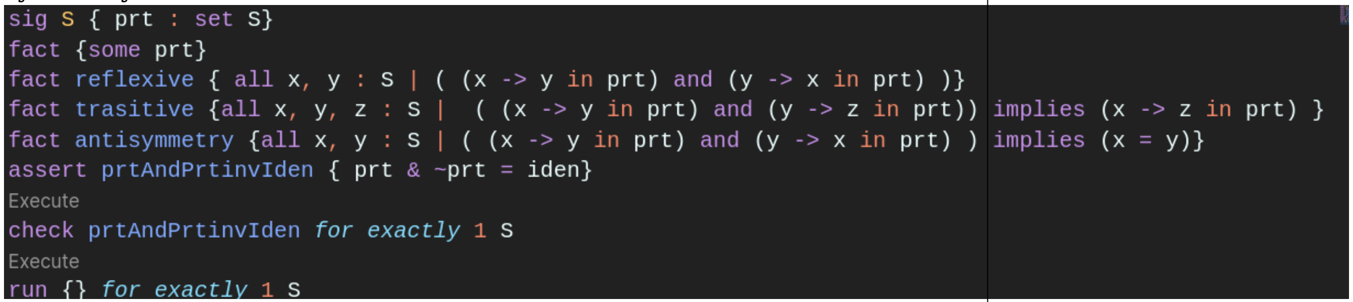


```
sig S {pre : set S}
fact {some pre}
fact reflexive { all x, y : S | ( (x -> y in pre) and (y -> x in pre) )}
fact trasitive {all x, y, z : S |  ( (x -> y in pre) and (y -> z in pre)) implies (x -> z in pre) }
assert preAndPreinvIden { pre & ~pre = iden}
Execute
check preAndPreinvIden for exactly 3 S
Execute
run {} for exactly 4 S
```

(d) The check did not pass as iden has more relation than pre and pre.

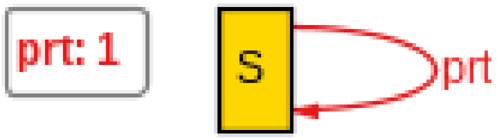


(e) **Partial order facts and output** Partial order is a preorder with anti-symmetry



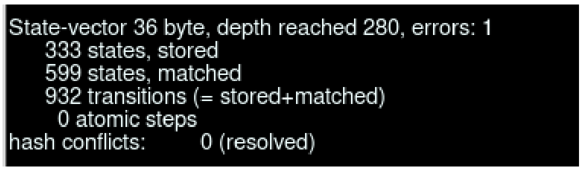
```
sig S { prt : set S}
fact {some prt}
fact reflexive { all x, y : S | ( (x -> y in prt) and (y -> x in prt) )}
fact trasitive {all x, y, z : S |  ( (x -> y in prt) and (y -> z in prt)) implies (x -> z in prt) }
fact antisymmetry {all x, y : S | ( (x -> y in prt) and (y -> x in prt) ) implies (x = y)}
assert prtAndPrtinvIden { prt & ~prt = iden}
Execute
check prtAndPrtinvIden for exactly 1 S
Execute
run {} for exactly 1 S
```

(f) The check did not pass as iden has more relation than part and part.



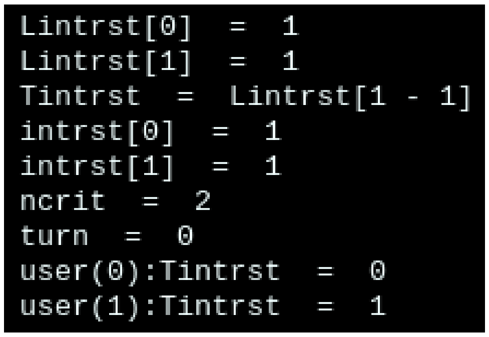
2. **Peterson TSO**

- Memory ordering describes the order of accesses to computer memory by a CPU. Most programming languages have some notion of a thread of execution which executes statements in a defined order.
- Processors use write buffers to hold committed stores until the memory system can process them. A store enters the write buffer when the store commits, and a store exits the write buffer when the block to be written is in the cache in a read–write coherence state.
- For a single-core processor a write buffer can be made invisible by ensuring that a load returns the value of the most recent store even if one or more stores to are in the write buffer. When building a multicore processor, it seems natural to use multiple cores, each with its own bypassing write buffer.
- **TSO** In-order memory operations: Read-to-Read, Read-to-Write, Write-to-Write. Out-of-order memory operations: Write-to-Read (later reads can bypass earlier writes)
  - Forwarding of pending writes in the store buffer to successive reads to the same location. Writes become visible to writing processor first
  - Store buffer is FIFO. Breaks Peterson's algorithm for mutual exclusion
- **Peterson TSO error** We see that the given code has an error.



```
State-vector 36 byte, depth reached 280, errors: 1
     333 states, stored
     599 states, matched
     932 transitions (= stored+matched)
       0 atomic steps
hash conflicts:        0 (resolved)
```

- The error trace shows that the ncrit value is 2 where as we asserted the value of ncrit as 1. Hence the mutual exclusion of peterson's algorithm is violated.



```
Lintrst[0]  =  1
Lintrst[1]  =  1
Tintrst  =  Lintrst[1 - 1]
intrst[0]  =  1
intrst[1]  =  1
ncrit  =  2
turn  =  0
user(0):Tintrst  =  0
user(1):Tintrst  =  1
```

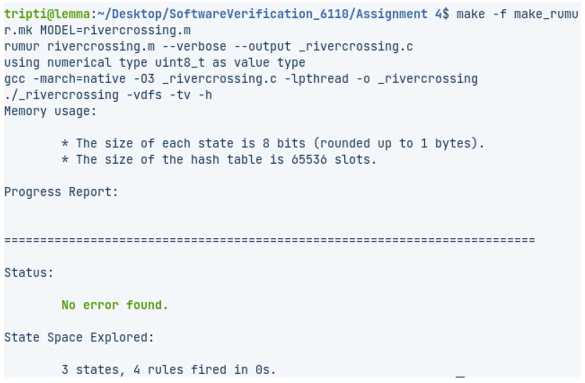
3. (10 points, The Java example with volatiles) Read-up on Java volatiles. Run the example `VBad.java`. Insert volatiles selectively (just for req or just for ack). Does that correct the apparent hang? (I don't know the answer but thought you'd like to try.) To get the apparent hang, first you must leave 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.)

- We get the hang when ack is not set to volatile.
- Volatile in java are like synchronized, atomic wrapper of making class thread-safe. Thread-safe means that a method or class instance can be used by multiple threads at the same time without any problem.
- If both ack and request are not set to volatile, then the value of req and ack keeps changing from true and false and both tid 0 and 1 gets stuck at first and second while loop, respectively.
- This means that the request is never getting acknowledged. The code is available on github link

[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)

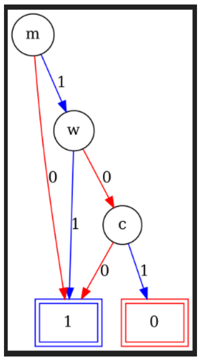
- The psuedo code for BFS is available in the following pdf
- Yes, the model checker does not infinitely loop and terminate when there is no next state. This is because the terminate variable becomes true and the loop breaks at that point. This is clear from figure 3 and 4 in the paper
- We recreate the mwgc example. The example in the class results in an error(code available here). The error trace is available here error trace.
- The new example have some extra safe state producing no errors (code available here), as shown in the figure below:



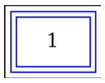
```
tripti@lemma:~/Desktop/SoftwareVerification_6110/Assignment 4$ make -f make_rumu
r.mk MODEL=rivercrossing.m
rumur rivercrossing.m --verbose --output _rivercrossing.c
using numerical type uint8_t as value type
gcc -march=native -O3 _rivercrossing.c -lpthread -o _rivercrossing
./_rivercrossing -vdfs -tv -h
Memory usage:

        * The size of each state is 8 bits (rounded up to 1 bytes).
        * The size of the hash table is 65536 slots.

Progress Report:


===========================================================================

Status:

        No error found.

State Space Explored:

        3 states, 4 rules fired in 0s.
```

- The BDD example is satisfiable but not equivalent (code available here), as shown below:

With given → test



With test → Given



- Let say we add another noun 'lion' and change the safe state to "lion can eat wolf" and try to solve the problem, we end up getting a deadlock. This is because none of them can cross the river, without at least one of them eating the other away. Hence this leads to a deadlock. The code is available here and the error trace is available here.

5. (20 points, DCL) **The "Double-Checked Locking is Broken" Declaration**

- C++ codes are dependent on memory model of the processor, the re-orderings performed by the compiler and the interaction between the compiler and the synchronization library. None of these are specified in C++. In java explicit memory barriers can be used.

- **Non synchoronized code** In multithreaded context, there can be multiple objects that can be allocated and hence we need to synchronize the method.

- The above doesn't work, because write can done or percieved out of order. Even if the compiler does not reorder those writes, the memory system reorder those writes and hence can run the thread running in another processor.

- Another fix is synchronizing inside an inner synchronized block. The rules of synchronization works that anything inside the synchronized block will work once the lock is released but anything outside can still work, and hence this idea also fails.

- Using full bidirectional memory barrier. This idea is inefficient and will not work in java. Also the thread which sees a non-null value for the method field also needs to perform memory barriers. This is because processors have their own locally cached copies of memory.

- **Making it work for static singleton** This guarantees that the field will not be intialized until the field is referenced and that any thread which accesses the field will see all of the writes resulting from initializing that field. This works with 32-bit primitive values but not with 64-bits. This is because unsynchronized reads/writes of 64-bit primitives are not guaranteed to be atomic.

- **Using thread local storage**. In this each thread keeps a local flag to determine to determine whether that thread has done the required synchronization. The performanceis dependent on JDK implementation used.

- **Using volatile** In JDK5 we can use volatile, which ensures that the system will not allow write of a volatile to be reordered wrt any previous read or write and a read of a volatile cannot be reordered with respect to any following read or write.

- **Immutatble objects** Will work without using volatile as well.