CMU 17-651 Models of Software Systems Fall 2016

Homework #12: LTL and FSP Concurrency

Garlan Due: 14 November 2016

1. For each of the following pairs, either argue (informally) why they are equivalent, or provide a counterexample trace that shows they are not equivalent.

   NOTE regarding counterexamples: To show that, for example, $\Box(p \wedge q)$ and $\Box(p \vee q)$ are *not* equivalent we provide the counterexample trace $\langle(p, \neg q), (p, \neg q), \ldots\rangle$. This trace is read as follows: "in state 1 p is true and q is false, in state 2 p is true and q is false, and so on for the entire trace." $\Box(p \vee q)$ is true for the given trace since p is true in every state of the trace (and hence so is $p \vee q$), but $\Box(p \wedge q)$ is not true since that would require both p and q to be true in every state of the trace.

   (a) $\Diamond p \wedge \Diamond q$ $\Diamond(p \wedge \Diamond q) \vee \Diamond(q \wedge \Diamond p)$
   equivalent:
   In the original assignement I classified it as not equivalent because I failed to understand the meaning of the outer $\Diamond$ operator in $\Diamond(q \wedge \Diamond p)$. I said that if q was false in the first state then the whole thing would become false which is not true. if q is true in any state and p is true in the same state or afterwards, then that expression would be true.

   (b) $\Diamond p \wedge \Diamond q$ $\Diamond(p \wedge q)$
   not equivalent.
   sample trace = $\langle(\neg p, q), (p, \neg q)\rangle$ is true for the expression on the left but not for the one on the right as p and q are not true at the same time in any state.

   (c) $\Box(p \vee q)$ $\Box p \vee \Box q$
   not equivalent
   sample trace = $\langle(\neg p, q), (p, \neg q)\rangle$
   is true for the expression on the left side but not for the one of the right as neither p or q are true forever.

   (d) $(p \wedge q)\,\mathcal{U}\, r$ $(p\,\mathcal{U}\, r) \wedge (q\,\mathcal{U}\, r)$
   equivalent.

2. Assuming that the following are true of σ:

   - $\Box((p \Rightarrow q) \vee s)$
   - $(\sigma, 3) \models \Box p$
   - $(\sigma, 3) \models \bigcirc(q \wedge \bigcirc\Box r)$
   - $(\sigma, 4) \models \Box(r \Rightarrow \neg q)$

   which of the following are true, which are false, and which could be either?

   (a) $(\sigma, 5) \models q$
   $(\sigma, 3) \models \bigcirc(q \wedge \bigcirc\Box r)$ tells us that when $\sigma = 5$ then $\Box r$ will be true,
   then because $(\sigma, 4) \models \Box(r \Rightarrow \neg q)$ then q has to be false.

(b) $(\sigma, 4) \models s$
s could be true or false because $p \Rightarrow q$ is true which satisfies $\Box((p \Rightarrow q) \vee s)$

(c) $(\sigma, 5) \models s$
s has to be true because q is false, so s needs to satisfy $\Box((p \Rightarrow q) \vee s)$

(d) $(\sigma, 3) \models q \vee s$
true because $(\sigma, 3) \models \Box p$ and $\Box((p \Rightarrow q) \vee s)$

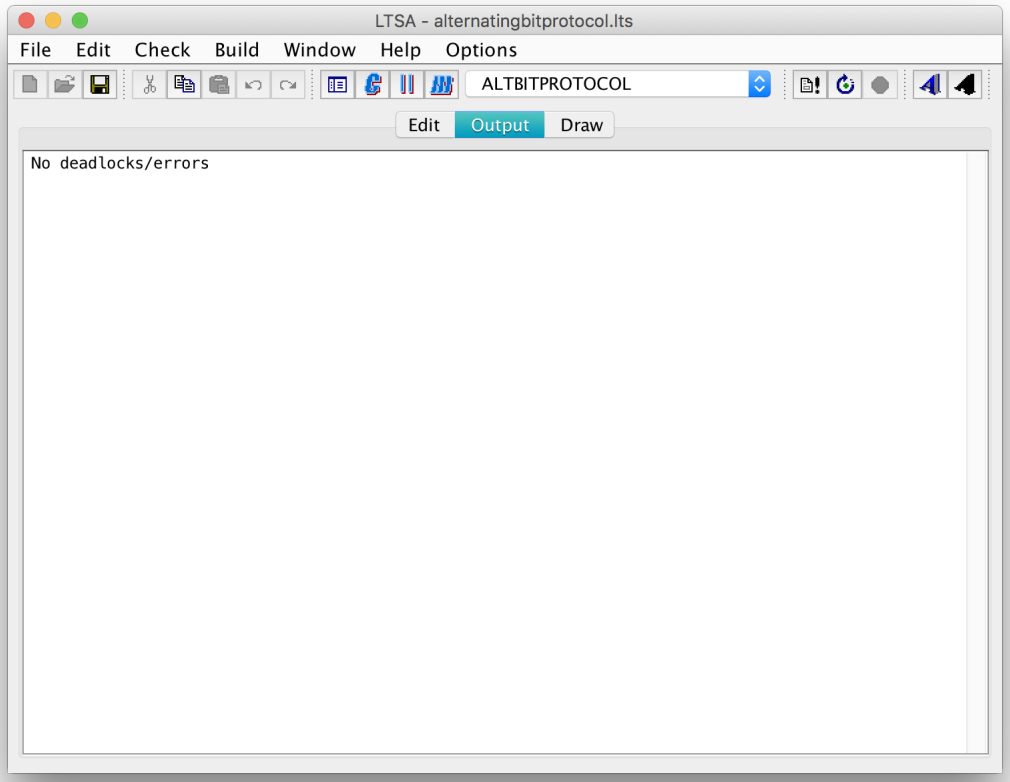
(e) $(\sigma, 4) \models r$
r has to be false because q is true due to $(\sigma, 3) \models \bigcirc(q \wedge \bigcirc\Box r)$

3. The following is an FSP model of the alternating-bit communication protocol over an unreliable link:

```
const Max = 1 range Msg = 0..Max

SENDER = SENDER[0], SENDER[i:Msg] = (
    send_msg[i] -> (
        msg_received[i] -> (
            // proceed to send the next message
            ack_received[i] -> SENDER[(i + 1) % (Max + 1)]
            |
            // retransmit
            ack_timeout[i] -> SENDER[i]
        )
        |
        // retransmit
        msg_timeout[i] -> SENDER[i]
    )
).

RECEIVER = RECEIVER[0][1], RECEIVER[i:Msg][j:Msg] = (
    msg_received[i] -> send_ack[i] -> ( // process the currently expected message
        ack_received[i] -> RECEIVER[(i + 1) % (Max + 1)][i]
        |
        ack_timeout[i] -> RECEIVER[(i + 1) % (Max + 1)][i]
    )
    |
    msg_received[j] -> send_ack[j] -> ( // process the previously expected message
        ack_received[j] -> RECEIVER[(j + 1) % (Max + 1)][j]
        |
        ack_timeout[j] -> RECEIVER[(j + 1) % (Max + 1)][j]
    )
).

||ALTBITPROTOCOL = (SENDER || RECEIVER).
```

As shown by the model, this protocol follows the "stop-and-wait" style. That is, a new message is not transmitted from the sender to the receiver unless (1) the receiver has sent back an acknowledgment and (2) the sender has received that acknowledgement. Since, the link is unreliable, both messages and acknowledgments may be lost at any time. Also, notice that the link is "half-duplex"—meaning that transmissions go over one direction at a time.

(a) Use LTSA to check if this protocol is deadlock free. Briefly explain why the protocol is deadlock free or why it is not.



According to the checker, the protocol is deadlock free as eventually both messages will be received and acknowledged correctly even when it takes many attempts due to the msg and ack timeouts.

(b) Define fluents `MSG_SENT` and `ACK_SENT` and an FLTL formula which uses those fluents and states that every message transmitted by the sender is eventually retrieved by the receiver.

```
fluent MSG_SENT[i:Msg] = <send_msg[i], msg_received[i]>

fluent ACK_SENT[i:Msg] = <send_ack[i], ack_received[i]>

assert ALL_MESSAGES_ARE_RECEIVED = forall[i:Msg] [] (MSG_SENT[i] -> <>ACK_SENT[i])
```

(c) Use LTSA to check if the protocol satisfies your LTL property. Briefly explain why the property is satisfied or why it is not.

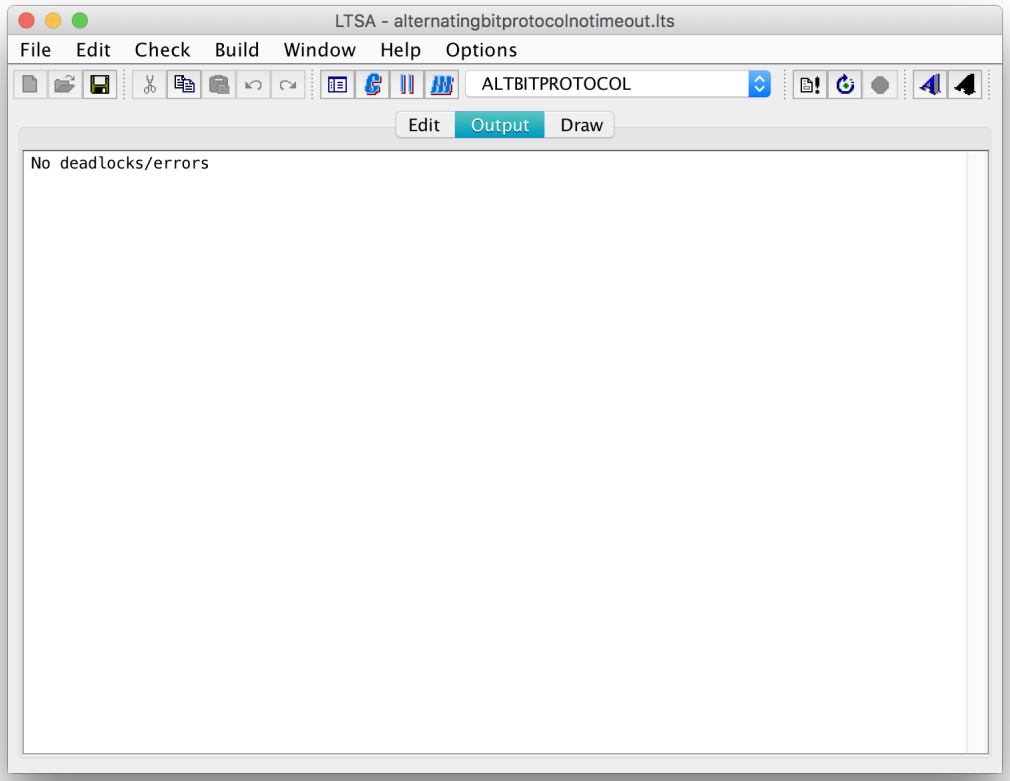
```
Composition:
ALTBITPROTOCOL = SENDER || RECEIVER || ALL_MESSAGES_ARE_RECEIVED
State Space:
 6 * 10 * 12 = 2 ** 11
LTL Property Check...
```

```
-- States: 59 Transitions: 108 Memory used: 240200K
No LTL Property violations detected.
LTL Property Check in: 0ms
```

According to LTL, the property is satisfied, I think that the reason is that even when timeouts occur, because of the $\diamond$ that I used, eventually the message will make it to the receiver and the ack will be received by the sender.

(d) Modify the model by removing the `*_timeout` transition choices and rerun LTSA to check if the modified protocol is deadlock free. Briefly explain why the modified protocol is deadlock free or why it is not.

```
const Max = 1 range Msg = 0..Max

SENDER = SENDER[0], SENDER[i:Msg] = (
    send_msg[i] -> (
        msg_received[i] -> (
            // proceed to send the next message
            ack_received[i] -> SENDER[(i + 1) % (Max + 1)]
        )
    )
).

RECEIVER = RECEIVER[0][1], RECEIVER[i:Msg][j:Msg] = (
    msg_received[i] -> send_ack[i] -> ( // process the currently expected message
        ack_received[i] -> RECEIVER[(i + 1) % (Max + 1)][i]
    )
    |
    msg_received[j] -> send_ack[j] -> ( // process the previously expected message
        ack_received[j] -> RECEIVER[(j + 1) % (Max + 1)][j]
    )
).

||ALTBITPROTOCOL = (SENDER || RECEIVER).

fluent MSG_SENT[i:Msg] = <send_msg[i], msg_received[i]>

fluent ACK_SENT[i:Msg] = <send_ack[i], ack_received[i]>

assert ALL_MESSAGES_ARE_RECEIVED = forall[i:Msg] [] (MSG_SENT[i] -> <>ACK_SENT[i])
```



According to the ltsa checker, the system is deadlock free as there is no way for the receiver to block the sender or the other way around, the second of the system imo is less desirable as timeouts are a useful and realistic error scenario that makes the model easier to implement.

NOTE: For every question in which you are asked to use the LTSA LTL property checker, you need to include the actual resulting output of the checker.