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) $\Diamond p \wedge \Diamond q$ $\qquad$ $\Diamond(p \wedge \Diamond q) \vee \Diamond(q \wedge \Diamond p)$

   (b) $\Diamond p \wedge \Diamond q$ $\qquad$ $\Diamond(p \wedge q)$

   (c) $\Box(p \vee q)$ $\qquad$ $\Box p \vee \Box q$

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

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) $(\sigma, 5) \models q$

   (b) $(\sigma, 4) \models s$

   (c) $(\sigma, 5) \models s$

   (d) $(\sigma, 3) \models q \vee s$

   (e) $(\sigma, 4) \models r$

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)]
            |
```

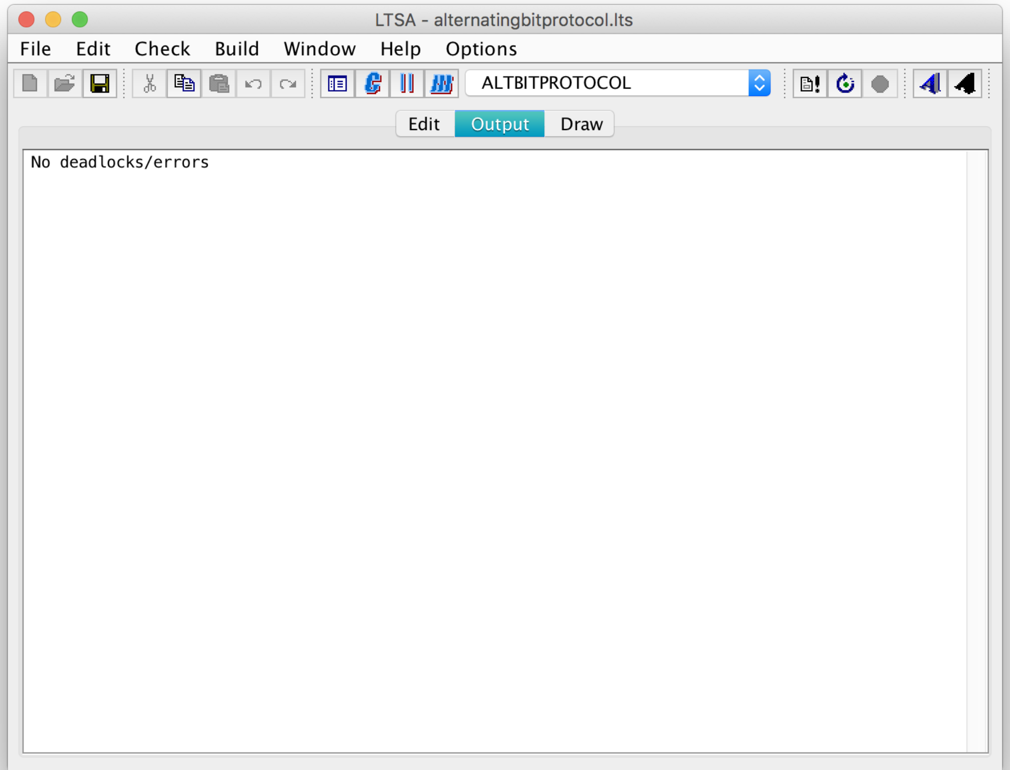
```
            // retransmit
            ack_timeout[i] -> SENDER[i]
        )
        |
        // retransmit
        msg_timeout[i] -> SENDER[i]
    )
).

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.



(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.

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

(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.

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.