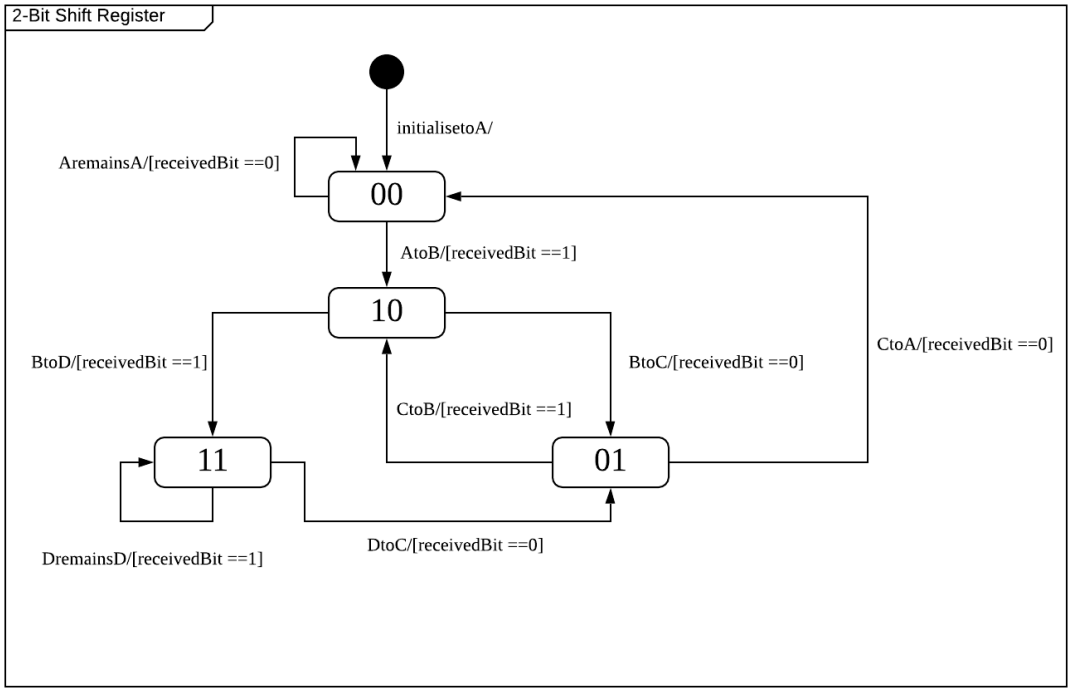
Bryan Leow Xuan Zhen (bryanlxz@kth.se)
Cevat Aykan Sevinç (sevinc@kth.se)

Question 1



Question 2

Complete definition of next state functions for Bit1 and Bit 2

```
next( Bit1 ) := case
    Bit1 = TRUE & input = FALSE : FALSE;
    Bit1 = FALSE & input = TRUE : TRUE;
    TRUE : Bit1;
 esac;

next( Bit2 ) := case
    Bit1 = TRUE & Bit2 = FALSE : TRUE;
    Bit1 = FALSE & Bit2 = TRUE : FALSE;
    TRUE : Bit2;
esac;
```

Output of NuSMV

For LTLSPEC G (Bit1 <-> (X Bit2)):

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- no counterexample found with bound 3
-- no counterexample found with bound 4
-- no counterexample found with bound 5
-- no counterexample found with bound 6
-- no counterexample found with bound 7
-- no counterexample found with bound 8
-- no counterexample found with bound 9
-- no counterexample found with bound 10
```

For LTLSPEC G (Bit1 <-> (X Bit1)):

```
-- no counterexample found with bound 0
-- specification  G (Bit1 <->  X Bit1)    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    Bit1 = FALSE
    Bit2 = FALSE
    state = s1
  -> Input: 1.2 <-
    input = TRUE
  -> State: 1.2 <-
    Bit1 = TRUE
    state = s2
```

It is verifiable that the completed definitions are correct.

The 1st LTLSPEC means that the value of Bit1 in the current time step will always be equal to the value of Bit2 in the next time step. Due to the pipeline effect where Bit1 is feeding Bit2, this LTLSPEC is obviously true.

The output of NuSMV for this LTLSPEC shows that no counterexample has been found from bound 0 to bound 10 (which is the default value for the variable *bmc_length* in NuSMV). While we have no control over the constraint solver input in each bound and not know the path taken for the execution, it is safe to assume that NuSMV has exhausted all possible paths (8 according to the UML State chart diagram) during the 10 bounds since it wants to find a counter-example. Hence, we can confidently say that no counterexample exists and our definition for the 1st LTLSPEC is correct.

The 2nd LTLSPEC means that the value of Bit1 at the current time step will always be equal to the value of Bit2 at the next time step. This is obviously false as the value of Bit1 will take over the value of the constraint solver input.

The output of NuSMV for the 2nd LTLSPEC shows that no counterexample has been found during bound 0 but a counter example is found in bound 1. At bound 0, Bit1, Bit2 and the state has just been initialised. Since Bit1 has not taken in any constraint solver input, a counterexample is not found. At bound 1, NuSMV feeds Bit1 with “TRUE” as the constraint solver input. This caused Bit1 to change its value from “FALSE” at the time step at bound 0 to “TRUE” at the time step of bound 1, hence prompting it to declare the 2nd LTLSPEC as false. Since Bit1 changes from “FALSE” at bound 0 to “TRUE” at bound 1, our definition is correct.

Bryan Leow Xuan Zhen (bryanlxz@kth.se)
Cevat Aykan Sevinç (sevinc@kth.se)

Question 3

(a)

For LTLSPEC G (! (state = stop)):

```
-- specification  G !(state = stop)    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    state = stop
```

Here, in the unmodified file, we can see that a counterexample exists at bound 0 where the **state = stop**. This happens at bound 0 because the state is initialised to stop.

(b)

Test requirement: *"cover state = stop of car controller"*

Trap property: *G (! (state = stop))*
Since state is initialised to stop here, strictly speaking no additional input values apart from the initialised state is required to reach *state = stop* according to the NuSMV screenshot in 3(a).

If there is really a need to provide input, the followings are valid inputs given that state is initialised to stop:
accelerate = FALSE & brake = FALSE
accelerate = FALSE & brake = TRUE

Output prediction: *state = stop*

(c)

The two additional trap properties and their respective counterexamples are shown below:

1.

```
-- For node coverage state = slow
LTLSPEC
    G( !(state = slow))
```

```
-- no counterexample found with bound 0
-- specification  G !(state = slow)    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -> State: 2.1 <-
    state = stop
  -> Input: 2.2 <-
    accelerate = TRUE
    brake = FALSE
  -> State: 2.2 <-
    state = slow
```

Test requirement: *"cover state = slow of car controller"*
Trap property: *G (! (state = stop))*
Inputs: *accelerate = TRUE & brake = FALSE | state = stop*
Output prediction: *state = slow*

2.

```
-- For node coverage state = fast
LTLSPEC
    G( !(state = fast))
```

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- specification  G !(state = fast)    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -> State: 3.1 <-
    state = stop
  -> Input: 3.2 <-
    accelerate = TRUE
    brake = FALSE
  -> State: 3.2 <-
    state = slow
  -> Input: 3.3 <-
  -> State: 3.3 <-
    state = fast
```

Test requirement: *"cover state = fast of car controller"*
Trap property: *G (! (state = fast))*
Inputs: *accelerate = TRUE & brake = FALSE | state = slow*
Output prediction: *state = fast*

Question 4

(a) **LTLSPEC G (state=stop & accelerate -> X (! (state=slow))):**

```
-- no counterexample found with bound 0
-- specification  G ((state = stop & accelerate) ->  X !(state = slow))    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -> State: 4.1 <-
    state = stop
  -> Input: 4.2 <-
    accelerate = TRUE
    brake = FALSE
  -> State: 4.2 <-
    state = slow
```

A counter example exists at bound 1 when *state = stop* and inputs are *accelerate = TRUE* and *brake = FALSE*. These state and input combinations lead to the next state that is slow.

(b) Test requirement: *"cover edge = stop to slow"*
Trap property: *G (state=stop & accelerate -> X (! (state=slow)))*
Inputs: *accelerate = TRUE & brake = FALSE | state = stop*
Output prediction: *state = slow*

(c)

The five additional trap properties and their respective counterexamples are shown below:

1.

```
-- For edge coverage slow to stop with brake
LTLSPEC
    G( state = slow & brake -> X( !(state = stop)  ) )
```

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- specification  G ((state = slow & brake) ->  X !(state = stop))    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 5.1 <-
    state = stop
  -> Input: 5.2 <-
    accelerate = TRUE
    brake = FALSE
  -> State: 5.2 <-
    state = slow
  -> Input: 5.3 <-
    accelerate = FALSE
    brake = TRUE
  -> State: 5.3 <-
    state = stop
```

Test requirement: *"cover edge = slow to stop via braking"*
Trap property: *G (state=slow & brake -> X (! (state=stop)))*
Inputs: *accelerate = FALSE & brake = TRUE | state = slow*
Output prediction: *state = stop*

2.

```
-- For edge coverage slow to stop with not acceleration
LTLSPEC
    G( state = slow & !accelerate & !brake -> X( !(state = stop) ) )
```

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- specification  G (((state = slow & !accelerate) & !brake) ->  X !(state = stop))    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 6.1 <-
    state = stop
  -> Input: 6.2 <-
    accelerate = TRUE
    brake = FALSE
  -> State: 6.2 <-
    state = slow
  -> Input: 6.3 <-
    accelerate = FALSE
  -> State: 6.3 <-
    state = stop
```

Test requirement: *"cover edge = slow to stop via not accelerating and not braking"*
Trap property: *G (state=slow &! accelerate &! brake -> X (! (state=stop)))*
Inputs: *accelerate = FALSE & brake = FALSE | state = slow*
Output prediction: *state = stop*

3.

```
-- For edge coverage slow to fast with acceleration
LTLSPEC
    G( state = slow & !brake & accelerate -> X( !(state = fast)  ) )
```

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- specification  G (((state = slow & !brake) & accelerate) ->  X !(state = fast))    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -> State: 7.1 <-
    state = stop
  -> Input: 7.2 <-
    accelerate = TRUE
    brake = FALSE
  -> State: 7.2 <-
    state = slow
  -> Input: 7.3 <-
  -> State: 7.3 <-
    state = fast
```

Test requirement: *"cover edge = slow to fast via accelerating and not braking"*
Trap property: *G (state=slow &! brake & accelerate -> X (! (state=fast)))*
Inputs: *accelerate = TRUE & brake = FALSE | state = slow*
Output prediction: *state = fast*

4.

```
-- For edge coverage fast to slow with not acceleration
LTLSPEC
    G( state = fast & !accelerate & !brake  -> X( !(state = slow)  ) )
```

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- specification  G (((state = fast & !accelerate) & !brake) ->  X !(state = slow))    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -> State: 8.1 <-
    state = stop
  -> Input: 8.2 <-
    accelerate = TRUE
    brake = FALSE
  -- Loop starts here
  -> State: 8.2 <-
    state = slow
  -> Input: 8.3 <-
  -> State: 8.3 <-
    state = fast
  -> Input: 8.4 <-
    accelerate = FALSE
  -> State: 8.4 <-
    state = slow
```

Test requirement: *"cover edge = fast to slow via not accelerating and not braking"*
Trap property: *G (state=fast &! accelerate &! brake -> X (! (state=slow)))*
Inputs: *accelerate = FALSE & brake = FALSE | state = fast*
Output prediction: *state = slow*

5.

```
-- For edge coverage fast to stop with brake
LTLSPEC
    G( state = fast & brake -> X( !(state = stop)  ) )
```

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- specification  G ((state = fast & brake) ->  X !(state = stop))    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 9.1 <-
    state = stop
  -> Input: 9.2 <-
    accelerate = TRUE
    brake = FALSE
  -> State: 9.2 <-
    state = slow
  -> Input: 9.3 <-
  -> State: 9.3 <-
    state = fast
  -> Input: 9.4 <-
    accelerate = FALSE
    brake = TRUE
  -> State: 9.4 <-
    state = stop
```

Test requirement: *"cover edge = fast to stop via braking"*
Trap property: G (state=fast & brake -> X (! (state=stop)))
Inputs: *brake = TRUE* | *state = fast*
Output prediction: *state = stop*