1a) Formalize the property that no communication between stations is aborted in the above protocol, i.e. any communication that is started (with a start signal) is finished (with a stop signal).

#define p0 (up[0] == start)

#define q0 (down[0] == stop)

#define p1 (up[1] == start)

#define q1 (down[1] == stop)

**LTL:**

{ []( (p0 -> (<> q0) ) && (p1 -> (<> q1) ) ) }

**Explanation:**

Globally, whenever ‘start’ is sent through a channel, ‘stop’ is eventually sent through the same channel.

1b) Describe the counter example you get from SPIN and explain how it can violate the property. Include the screenshot of the counter example (e.g. message sequence chart) in the report.

**Situation A:**

1. Both busy[id] flags are set (busy[0] = true; busy[1] = true) by branches that sent ‘start’ and are blocked waiting for ‘ack’

:: atomic { !busy[id] -> busy[id] = true };

out!start;

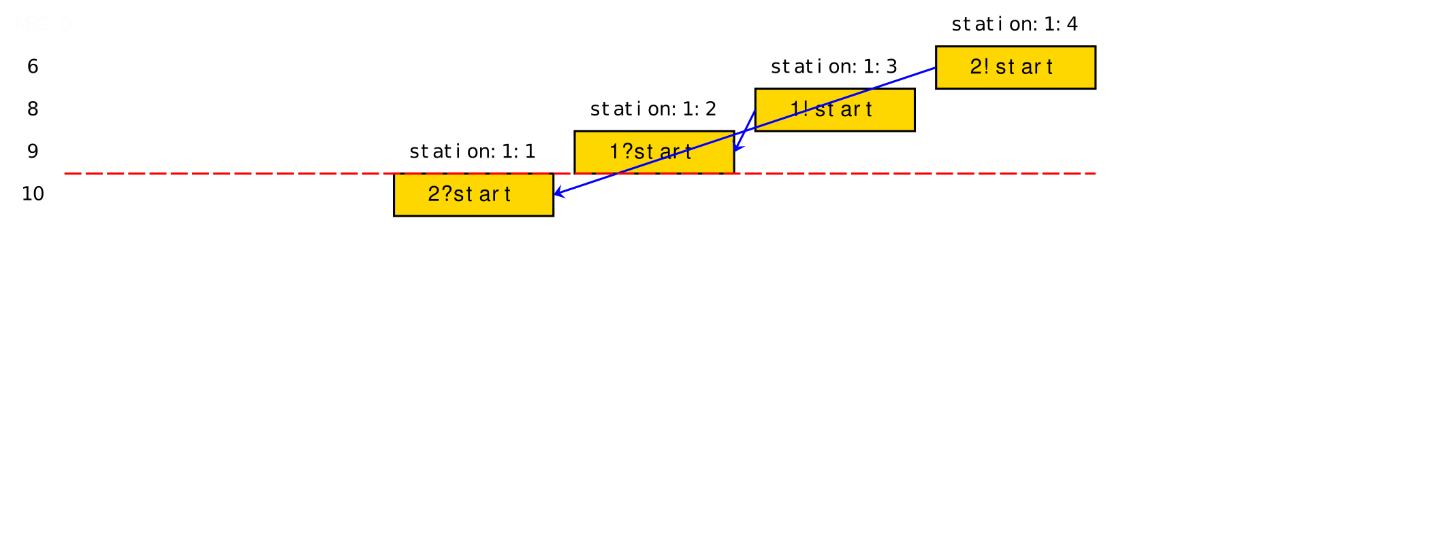
in?ack;

1. Branches that just received ‘start’ will be blocked by the busy flag statement and will be unable to send the ‘ack’

:: in?start ->

atomic { !busy[id] -> busy[id] = true };

out!ack;



The above MSC shows a deadlock when the original model was verified.

**Solution A:**

Remove the check for busy[id] flags in the branch that receives ‘start’

Code Snippet:

:: in?start ->

// atomic { !busy[id] -> busy[id] = true };

…

// busy[id] = false

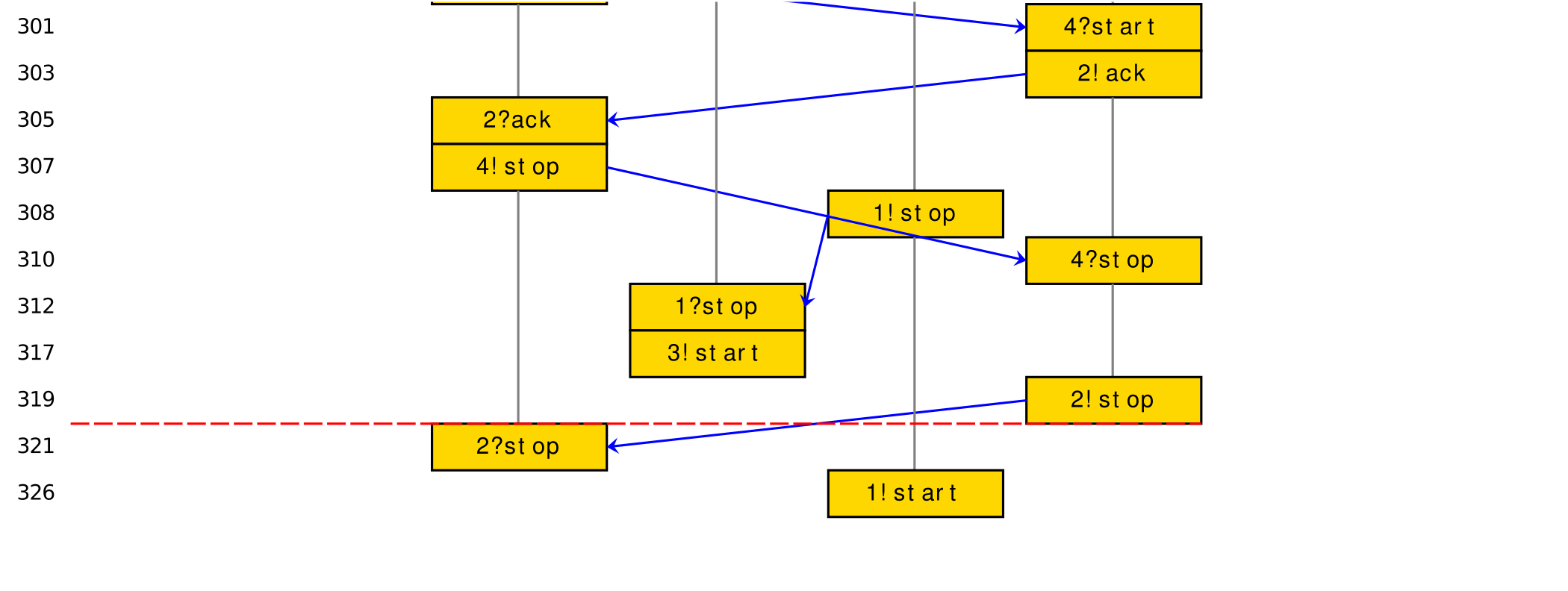
**Situation B:**

1. Branches in stations with different ids and sharing the same channels both try to send ‘start’ and are both blocked waiting for ‘ack’

E.g. Process 2: run station(id=1, up[0], down[0]);

Process 3: run station(id=0, down[0], up[0]);

1. Both processes will block indefinitely since no process sends ‘ack’



The above MSC shows the deadlock after Situation A was fixed

**Solution B:**

If the process with id=0 sends ‘start’, the corresponding process with id=1 must be ready to receive ‘start’ The converse will also be true if id=1 sends ‘start’. In general, it can't be that both processes send ‘start’ and neither receives ‘start’.

This behavior can be achieved by using a flag start\_sent to keep track that 1 of the 2 sister processes have sent ‘start’.

Code Snippet:

proctype station(byte id; chan in; chan out; byte chan\_id)

:: atomic { (busy[id] == false && start\_sent[chan\_id] == false) -> busy[id] = true; start\_sent[chan\_id] = true };

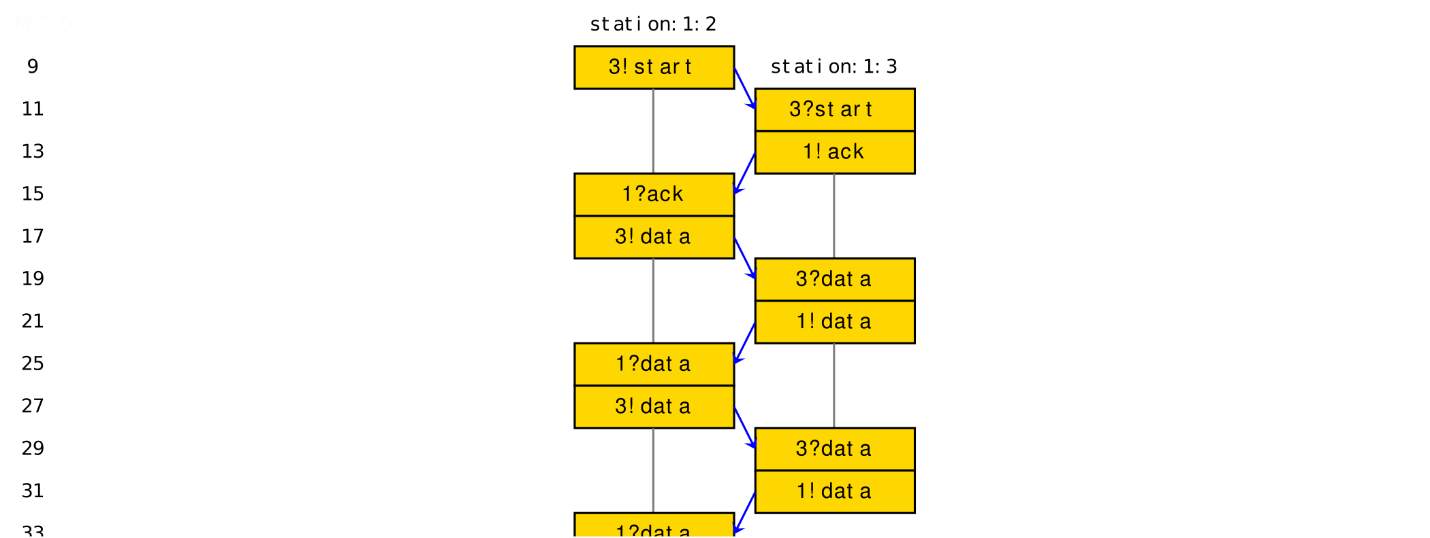
…

atomic { busy[id] = false; start\_sent[chan\_id] = false }

Process 2: run station(id=1, up[0], down[0], chan\_id=0);

Process 3: run station(id=0, down[0], up[0], chan\_id=0);

**Situation C:**



The above MSC shows the model getting stuck in an infinite loop, where communication goes on forever if out!data executes infinitely many times in the do loop

Original Code:

do

:: out!data -> in?data

:: out!stop -> break

od;

**Solution C:**

Limit the number of times [:: out!data -> in?data] is executed. Thereafter execute [:: out!stop -> break] to proceed towards completing the communication protocol. For example, the code snippet below shows the limit set to 1.

Code Snippet:

do

:: out!data -> in?data

out!stop -> break

od;

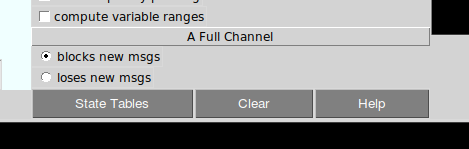
2b ) Explain how the reassembly deadlock can occur for the given model, provide a detailed description (textual explanation) for your answer.

Deadlock can happen when:

1. Local buffer (channel of size=5) is FULL

chan buffer = [5] of { mtype };

1. Full channel blocks the processes trying to send new messages



1. Process assemble() is blocked because not all 3 types of messages are present in the buffer

Code Snippet:

:: (red\_count > 0 && green\_count > 0 && blue\_count > 0) ->

cs: atomic {

buffer??red;

buffer??green;

buffer??blue;

red\_count = red\_count - 1;

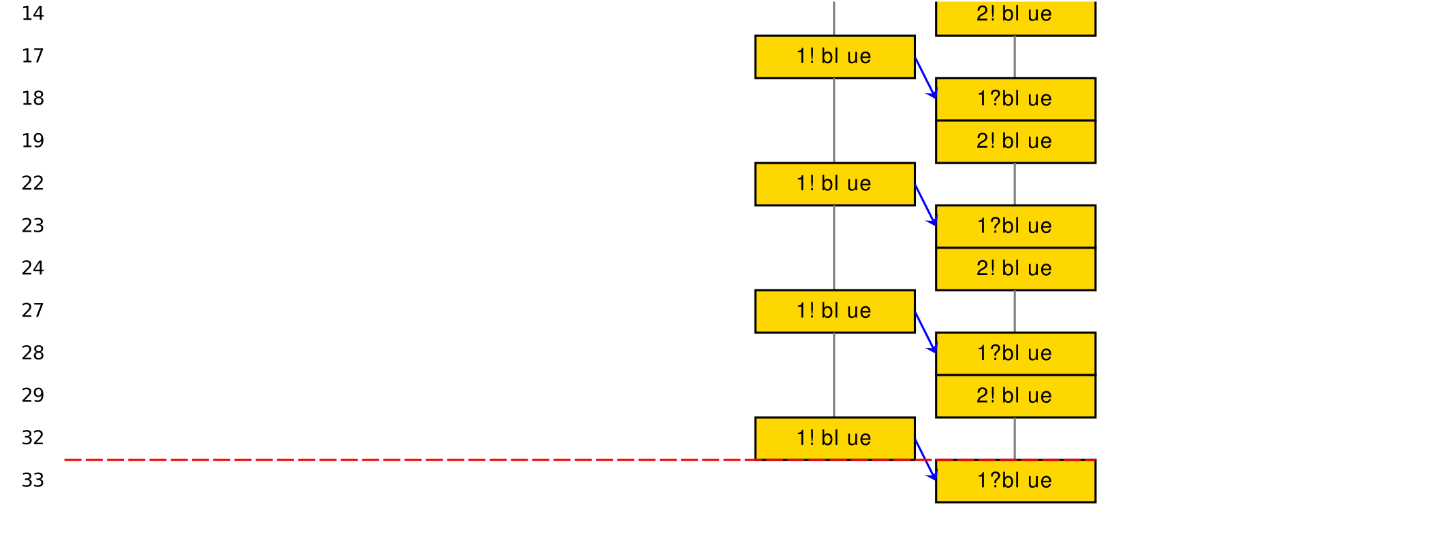
green\_count = green\_count - 1;

blue\_count = blue\_count - 1;

}

2c) Define the key property in Linear-time temporal logic (LTL), use SPIN to prove that it can be violated,

and show the counter-example.



The above MSC shows how deadlock occurs when verification is run on the model and the buffer is filled with only 1 type of message (blue in this case)

Code Snippet:

:: (red\_count > 0 && green\_count > 0 && blue\_count > 0) ->

cs: atomic {

buffer??red;

buffer??green;

buffer??blue;

red\_count = red\_count - 1;

green\_count = green\_count - 1;

blue\_count = blue\_count - 1;

}

**LTL:**

{ [] <> (assemble@cs) }

**Explanation:**

Globally, eventually control point cs will be reached by process assemble(). Thus assemble@cs should be executed infinitely many times given an infinite-length trace.