

PART 1 – JSPIN

Task 1

The request was to model a system composed by two producers, one priority queue and one consumer. So, the first thing to do was to create an mtype (Jspin representation for enum) composed by the producer's name (A or B) and the type of message. Another solution to represent the type of message to send could be the definition of a type with attribute "id" and the name of the producer, the first method is chosen for simplicity.

The model proposed can now concurrently receive or produce the messages as requested and the production mentioned above is possible only if exactly three acks are received. The resetting of variables is done using the "d_step" key word, this permits to perform all the specified operation at once but without inserting additional states.

An important request of the system was to give the priority to all the messages that the "priority queue" would receive from producer A. This condition is satisfied by checking if there are any messages in channel "pa"; if so, it atomically goes straight to the consumer. The consumer, as its last operation, will check to whom send the acknowledgement (A or B) and so begin again the production cycle.

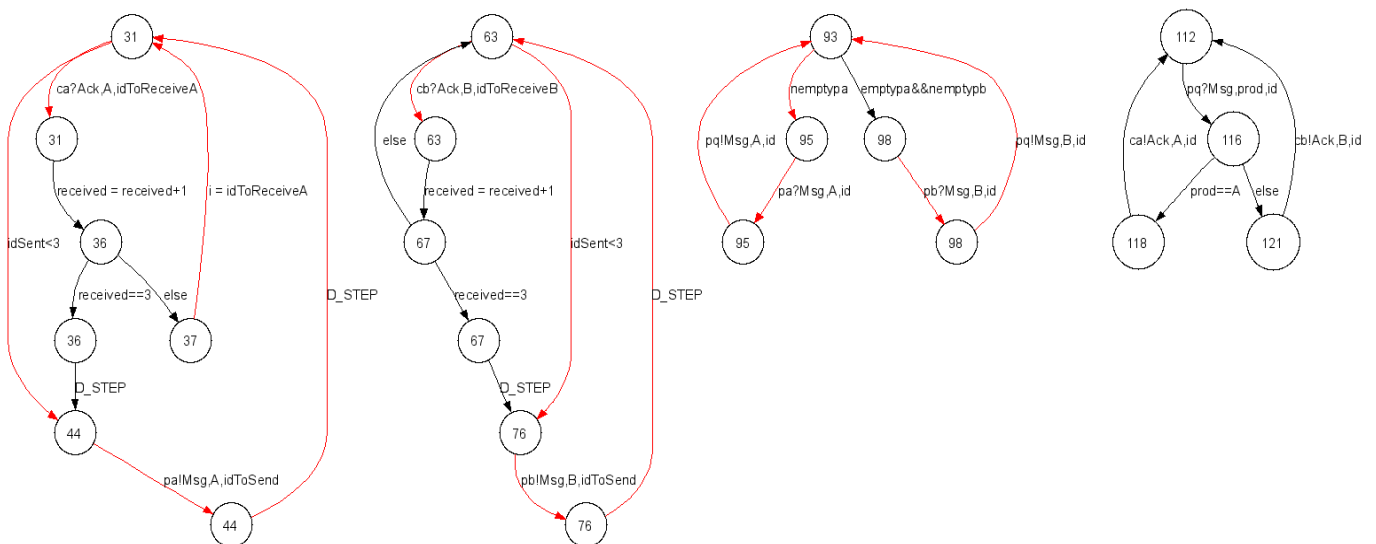


Figure 1 Automata of proposed model

Note: ProducerA presents an additional operation on "else" condition than ProducerB because there is the updating of variable "l", useful for realization of P1 LTL condition.

Task 2

- P1: The first property wants to know if the messages received by the producers respect the specified order (0,1,2). The variable “i” represents the previous message received. This is a safety property.

Spin Version 5.0

File Edit Spin Convert Options Settings Output SpinSpider Help

LTL formula p1

Open Check Random Interactive Guided Weak fairness ☒ Safety Verify Stop Translate Load LTL name SpinSpider Maximize

```

126 od
127 }
128
129 /* Whenever Producer A (resp. B) received an acknowledgment for a message with
130 i and a subsequent acknowledgment for a message with id j it holds that
131 j = (i + 1) mod 3 */
132 j = (i + 1) mod 3 */
133
134 lt1 p1 {[] (ProducerA@ackReceived -> (idToReceiveA == (i+1)%3)) }
135
136 // Producer A never stops
137 lt1 p2 {[] << (ProducerA@startProd))}
138
139
140 /* Whenever a message from B is delivered to the Consumer by the Priority Queue
141 currently no messages from Producer A that wait for being delivered */
142 lt1 p3 {[] (PriorityQueue@sendMessageFromB -> !PriorityQueue@sendMessageFromA)}
143
144 /* The Consumer will never wait for sending messages on channels ca or cb, that
145 say, there is always at least one free place in the buffer of the channel ca or
146 the Consumer wants to send a message through it */
147 lt1 p4 {[] ((Consumer@sendToA -> len(ca) < 3) && (Consumer@sendToB -> len(cb) < 3)) }
148
149 /* Whenever a message is produced, from Producer A or Producer B, a message will
150 be consumed */
151 lt1 p5 {[] ((ProducerA@prodAndSend -> << (Consumer@consumeMsg)) && (ProducerB@prodAndSend -> << (Consumer@consumeMsg))) }
152

```

warning: never claim + accept labels requires -a flag to fully verify
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
error: max search depth too small
(Spin Version 6.0.0 -- 5 December 2010)
+ Partial Order Reduction
Full statespace search for:
never claim +
assertion violations + (if within scope of claim)
cycle checks - (disabled by -DSAFETY)
invalid end states - (disabled by never claim)
State-vector 108 byte, depth reached 1999, *** errors: 0 ***
34167 states, stored
55292 states, matched
89459 transitions (= stored+matched)
22473 atomic steps
hash conflicts: 1228 (resolved)
6.102 memory usage (Mbyte)
unreached in proctype ProducerA
prodCons.pml:50, state 25, "-end-"
(1 of 25 states)
unreached in proctype ProducerB
prodCons.pml:83, state 23, "-end-"
(1 of 23 states)
unreached in proctype PriorityQueue
prodCons.pml:106, state 12, "-end-"
(1 of 12 states)
unreached in proctype Consumer

- P2: Liveness property. The property expressed specify that the task “startProd” will be reached infinitely often.

Spin Version 5.0

File Edit Spin Convert Options Settings Output SpinSpider Help

LTL formula p2

Open Check Random Interactive Guided Weak fairness ☒ Safety Verify Stop Translate Load LTL name SpinSpider Maximize

```

126 od
127 }
128
129 /* Whenever Producer A (resp. B) received an acknowledgment for a message with
130 i and a subsequent acknowledgment for a message with id j it holds that
131 j = (i + 1) mod 3 */
132 j = (i + 1) mod 3 */
133
134 lt1 p1 {[] (ProducerA@ackReceived -> (idToReceiveA == (i+1)%3)) }
135
136 // Producer A never stops
137 lt1 p2 {[] << (ProducerA@startProd))}
138
139
140 /* Whenever a message from B is delivered to the Consumer by the Priority Queue
141 currently no messages from Producer A that wait for being delivered */
142 lt1 p3 {[] (PriorityQueue@sendMessageFromB -> !PriorityQueue@sendMessageFromA)}
143
144 /* The Consumer will never wait for sending messages on channels ca or cb, that
145 say, there is always at least one free place in the buffer of the channel ca or
146 the Consumer wants to send a message through it */
147 lt1 p4 {[] ((Consumer@sendToA -> len(ca) < 3) && (Consumer@sendToB -> len(cb) < 3)) }
148
149 /* Whenever a message is produced, from Producer A or Producer B, a message will
150 be consumed */
151 lt1 p5 {[] ((ProducerA@prodAndSend -> << (Consumer@consumeMsg)) && (ProducerB@prodAndSend -> << (Consumer@consumeMsg))) }
152

```

warning: never claim + accept labels requires -a flag to fully verify
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
error: max search depth too small
(Spin Version 6.0.0 -- 5 December 2010)
+ Partial Order Reduction
Full statespace search for:
never claim +
assertion violations + (if within scope of claim)
cycle checks - (disabled by -DSAFETY)
invalid end states - (disabled by never claim)
State-vector 108 byte, depth reached 1999, *** errors: 0 ***
56302 states, stored
110718 states, matched
167020 transitions (= stored+matched)
48926 atomic steps
hash conflicts: 3888 (resolved)
8.641 memory usage (Mbyte)
unreached in proctype ProducerA
prodCons.pml:50, state 25, "-end-"
(1 of 25 states)
unreached in proctype ProducerB
prodCons.pml:83, state 23, "-end-"
(1 of 23 states)
unreached in proctype PriorityQueue
prodCons.pml:106, state 12, "-end-"
(1 of 12 states)
unreached in proctype Consumer
prodCons.pml:128, state 11, "-end-"

- P3: Safety property. The property checks that the “PriorityQueue” has not reached the state in which it can forwards messages from A if it can send messages from B instead.

```

126 od
127 }
128 }
129
130 /* Whenever Producer A (resp. B) received an acknowledgment for a message with
131 i and a subsequent acknowledgment for a message with id j it holds that
132 j = (i + 1) mod 3 */
133
134 !tl1 p1 {[] (ProducerA@ackReceived -> (idToReceiveA == (i+1)%3)) }
135
136 // Producer A never stops
137 !tl1 p2 {[] (<> (ProducerA@startProd))}
138
139
140 /* Whenever a message from B is delivered to the Consumer by the Priority Queue
141 currently no messages from Producer A that wait for being delivered */
142 !tl1 p3 {[] (PriorityQueue@sendMessageFromB -> !PriorityQueue@sendMessageFromA)}
143
144 /* The Consumer will never wait for sending messages on channels ca or cb, that
145 say, there is always at least one free place in the buffer of the channel ca or
146 the Consumer wants to send a message through it */
147 !tl1 p4 {[] ((Consumer@sendToA -> len(ca) < 3) && (Consumer@sendToB -> len(cb) <
148 3))}
149
150 /* Whenever a message is produced, from Producer A or Producer B, a message will
151 be consumed */
152 !tl1 p5 {[] ((ProducerA@prodAndSend -> <> (Consumer@consumeMsg)) && (ProducerB@prodAndSend -> <> (Consumer@consumeMsg)))}

```

Warning: never claim + accept labels requires -a flag to fully verify
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
error: max search depth too small
(Spin Version 6.0.0 -- 5 December 2010)
+ Partial Order Reduction
Full statespace search for:
never claim +
assertion violations + (if within scope of claim)
cycle checks - (disabled by -DSAFETY)
invalid end states - (disabled by never claim)
State-vector 108 byte, depth reached 1999, *** errors: 0 ***
34167 states, stored
55292 states, matched
89459 transitions (= stored+matched)
22473 atomic steps
hash conflicts: 1192 (resolved)
6.102 memory usage (Mbyte)
unreached in proctype ProducerA
prodCons.pml:50, state 25, "-end-"
(1 of 25 states)
unreached in proctype ProducerB
prodCons.pml:83, state 23, "-end-"
(1 of 23 states)
unreached in proctype PriorityQueue
prodCons.pml:106, state 12, "-end-"
(1 of 12 states)
unreached in proctype Consumer
prodCons.pml:128, state 11, "-end-"

- P4: Safety property. Whenever the consumer wants to send a message through the channel “ca” or “cb”, it will never wait because the size of these channels is exactly 3.

```

126 od
127 }
128 }
129
130 /* Whenever Producer A (resp. B) received an acknowledgment for a message with
131 i and a subsequent acknowledgment for a message with id j it holds that
132 j = (i + 1) mod 3 */
133
134 !tl1 p1 {[] (ProducerA@ackReceived -> (idToReceiveA == (i+1)%3)) }
135
136 // Producer A never stops
137 !tl1 p2 {[] (<> (ProducerA@startProd))}
138
139
140 /* Whenever a message from B is delivered to the Consumer by the Priority Queue
141 currently no messages from Producer A that wait for being delivered */
142 !tl1 p3 {[] (PriorityQueue@sendMessageFromB -> !PriorityQueue@sendMessageFromA)}
143
144 /* The Consumer will never wait for sending messages on channels ca or cb, that
145 say, there is always at least one free place in the buffer of the channel ca or
146 the Consumer wants to send a message through it */
147 !tl1 p4 {[] ((Consumer@sendToA -> len(ca) < 3) && (Consumer@sendToB -> len(cb) <
148 3))}
149
150 /* Whenever a message is produced, from Producer A or Producer B, a message will
151 be consumed */
152 !tl1 p5 {[] ((ProducerA@prodAndSend -> <> (Consumer@consumeMsg)) && (ProducerB@prodAndSend -> <> (Consumer@consumeMsg)))}

```

Warning: never claim + accept labels requires -a flag to fully verify
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
error: max search depth too small
(Spin Version 6.0.0 -- 5 December 2010)
+ Partial Order Reduction
Full statespace search for:
never claim +
assertion violations + (if within scope of claim)
cycle checks - (disabled by -DSAFETY)
invalid end states - (disabled by never claim)
State-vector 108 byte, depth reached 1999, *** errors: 0 ***
34167 states, stored
55292 states, matched
89459 transitions (= stored+matched)
22473 atomic steps
hash conflicts: 1138 (resolved)
6.102 memory usage (Mbyte)
unreached in proctype ProducerA
prodCons.pml:50, state 25, "-end-"
(1 of 25 states)
unreached in proctype ProducerB
prodCons.pml:83, state 23, "-end-"
(1 of 23 states)
unreached in proctype PriorityQueue
prodCons.pml:106, state 12, "-end-"
(1 of 12 states)
unreached in proctype Consumer
prodCons.pml:128, state 11, "-end-"

- P5: Liveness property. The property expressed says that if producer A (resp. B) has reached state “prodAndSend”, sooner or later will reach the state in which the messages are consumed (“consumeMsg”).

```

File Edit Spin Convert Options Settings Output SpinSpidey Help LTL formula p5
Open Check Random Interactive Guided Weak fairness Acceptance Verify Stop Translate Load LTL name SpinSpidey Maximize

126 od
127 }
128 }
129
130 /* Whenever Producer A (resp. B) received an acknowledgment for a message with
131 i and a subsequent acknowledgment for a message with id j it holds that
132 j = (i + 1) mod 3 */
133
134 !t1 p1 {[] (ProducerA@ackReceived -> (idToReceiveA == (i+1)%3)) }
135
136 // Producer A never stops
137 !t1 p2 {[] (<> (ProducerA@startProd))}
138
139
140 /* Whenever a message from B is delivered to the Consumer by the Priority Queue
141 currently no messages from Producer A that wait for being delivered */
142 !t1 p3 {[] (PriorityQueue@sendMessageFromB -> !PriorityQueue@sendMessageFromA)}
143
144 /* The Consumer will never wait for sending messages on channels ca or cb, that
145 say, there is always at least one free place in the buffer of the channel ca or
146 the Consumer wants to send a message through it */
147 !t1 p4 {[] ((Consumer@sendToA -> len(ca) < 3) && (Consumer@sendToB -> len(cb) < 3))}
148
149 /* Whenever a message is produced, from Producer A or Producer B, a message will
150 be consumed */
151 !t1 p5 {[] ((ProducerA@prodAndSend -> <> (Consumer@consumeMsg)) && (ProducerB@prodAndSend -> <> (Consumer@consumeMsg)))}
152

```

```

warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
error: max search depth too small
(Spin Version 6.0.0 -- 5 December 2010)
+ Partial Order Reduction

Full statespace search for:
never claim
assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 108 byte, depth reached 1999, *** errors: 0 ***
62681 states, stored (90156 visited)
202539 states, matched
292695 transitions (= visited+matched)
77843 atomic steps

hash conflicts: 8727 (resolved)
Stats on memory usage (in Megabytes):
7.412 equivalent memory usage for states (stored*(State-vector +
overhead))
5.908 actual memory usage for states (compression: 79.71%)
state-vector as stored = 83 byte + 16 byte overhead
2.000 memory used for hash table (-w19)
0.061 memory used for DFS stack (-m2000)
7.957 total actual memory usage

unreached in proctype ProducerA
prodCons.pml:50, state 25, "-end-"
(1 of 25 states)

unreached in proctype ProducerB

```

Part 2

Task 1

The model proposed for the simplified version of the previous system has one template for each component of the system itself: Producer, Consumer, Queue.

Producer and Consumer present similar behaviour; since both production and consumption have to start after producing and consuming a message, the time is reset after every transition. The Producer is composed by two states in which are distinguishable the starting of a production and its continuation; the first state manages the sending only if the id of the message is set to 0, so it starts or restart to produce again. The second state continue to produce until it reaches the maximum number of messages, when this happens it goes to the starting state. The Consumer has the same behaviour but with the exception that the transitions are activated only when receiving from the Queue.

The Queue has only one state, this is because it is always ready to receive or to send a message. To control the incoming messages a variable "idToReceive" has been added, so every time it synchronizes with the Producer this variable is incremented or reset inside method "enqueue".

Task 2

Question 1 (Check that the system is deadlock free).

The system is deadlock free.

Question 2 (Determine whether or not it is possible that the Queue becomes full).

The queue cannot become full because it can immediately free the messages, so the condition is not satisfied.

Question 3 (Determine whether or not in every run sooner or later the Queue will become full).

Same as above.

Question 4 (Determine whether or not the Producer is alive, i.e., whenever it produces a message, it will eventually produce another message).

This property is satisfied, since after every production another will begin.

Question 5 (Now change the capacity N to 2 and repeat the checks of the previous 4 points. Is there any difference? Explain why).

Changing the capacity of the queue will not affect the results since the queue will free immediately the messages, even because the intersection between t_1 and t_2 is not empty.

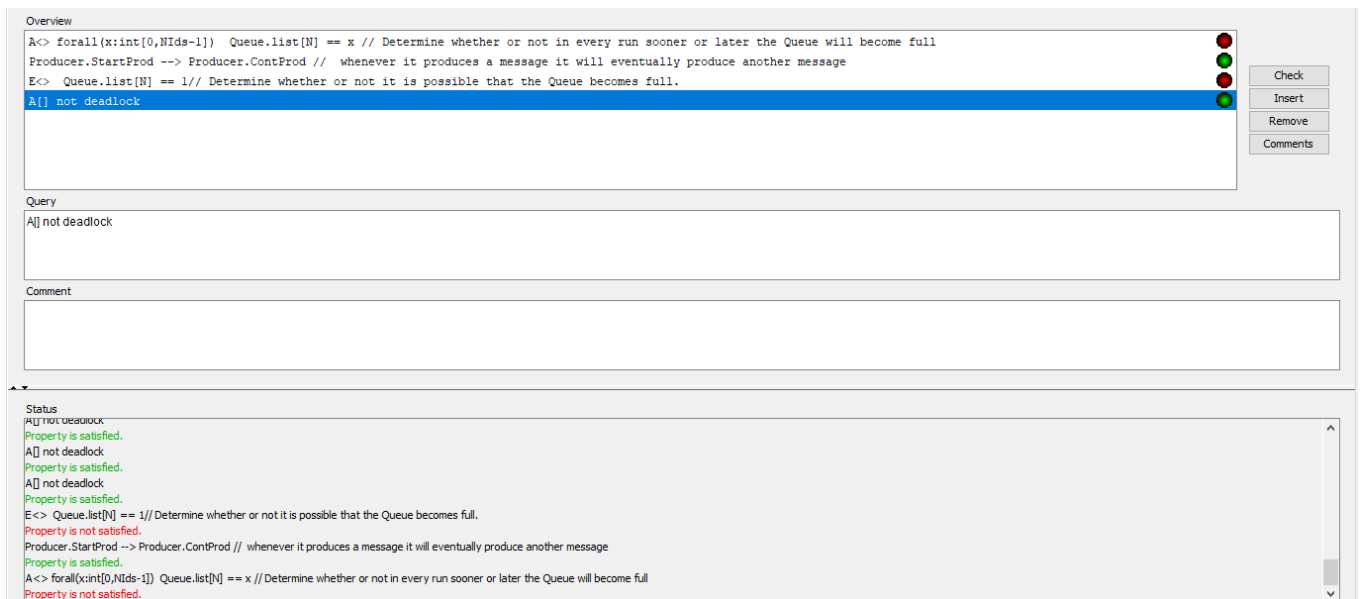


Figure 2 - Results from requested conditions

Question 6(Now change the time interval for the Consumer from [2, 3] to [3, 4]and check that the system can reach a deadlock situation. Why is it so? Is it possible to find a value for the capacity of the Queue in order to avoid the deadlock? Justify your answers).

The system can actually reach a deadlock situation. This is because when entering the queue, it has to wait for the Consumer to be into [3,4] range; so, the producer that is already producing another message waits too long (since the queue becomes full in this case) and cannot perform actions anymore.

Finally, is not possible to avoid the deadlock changing the capacity of the queue since sooner or later the Producer will wait for the Queue to become free.

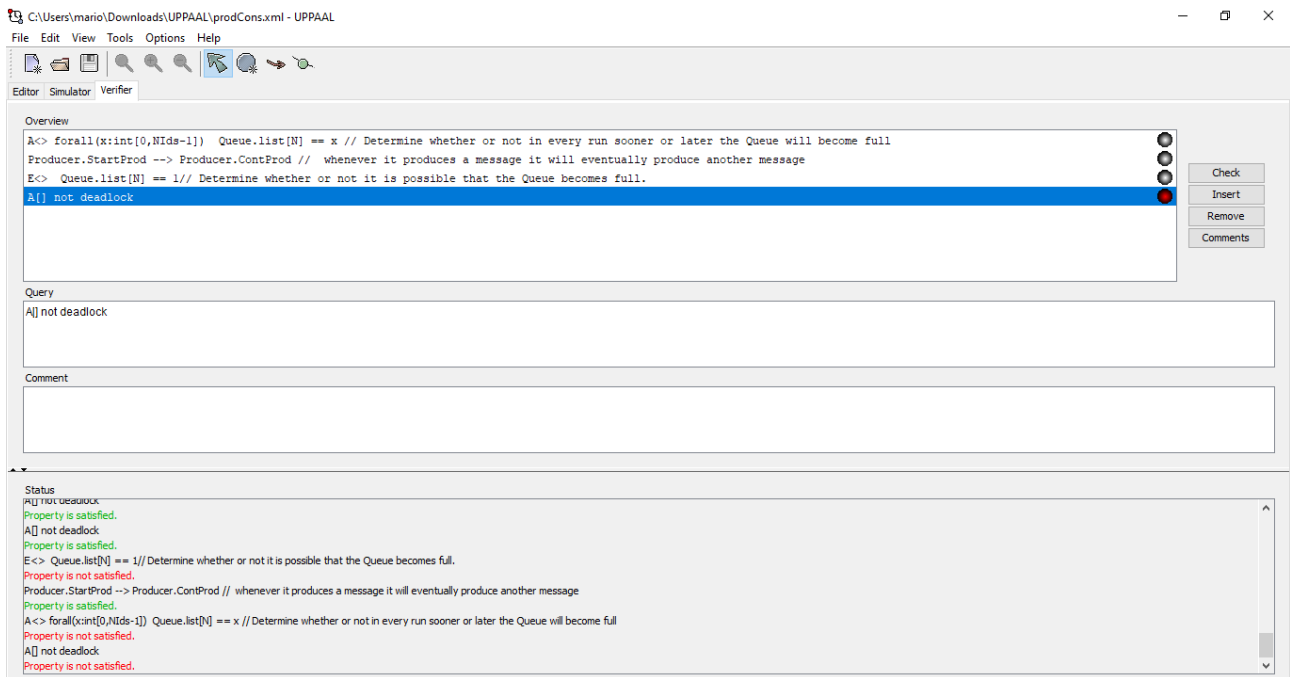


Figure 3 - Changing the consumption time the system is not deadlock free.