Department of Computer Sciences
University of Salzburg

Introduction to Concurrency and Verification - Miniproject
SS 2011

Alternating Bit Protocol

July 22, 2011

Author:

Christian ALT

Christian.alt@gmail.com

1 Implementation and Specification

Listing 1: Implementation and Specification

```
1  * Alternating Bit Protocol
2  *
3  *Actions:
4  *sendmsg0 - send a message with additional bit 0
5  *sendmsg1 - send a message with additional bit 1
6  *recvmsg0 - receive a message with additional bit 0
7  *recvmsg1 - receive a message with additional bit 1
8  *sendack0 - send an acknowledgment with additional bit 0
9  *sendack1 - send an acknowledgment with additional bit 1
10 *recvack0 - receive an acknowledgment with additional bit 0
11 *recvack1 - receive an acknowledgment with additional bit 1
12 *deliver0 - deliver a message with additional bit 0
13 *deilver1 - deliver a message with additional bit 1
14 *timeout
15
16 * The Sender sends a message with a corresponding bit and then waits until
17 * it receives the acknowledgment or a timeout happens.
18 * In case of an acknowledgment it toggles the bit and sends the next
19 * message, in case of a timeout it resends the message.
20 * The agent Sender0 sends a message that has 0 as additional bit (sendmsg0),
21 * Sender 1 sends a message with a 1 as additional bit (sendmsg1).
22 agent Sender0 = 'sendmsg0.Sender0Waiting;
23 agent Sender0Waiting = recvack0.Sender1 + timeout.Sender0 ;
24 agent Sender1 = 'sendmsg1.Sender1Waiting;
25 agent Sender1Waiting = recvack1.Sender0 + timeout.Sender1;
26
27 *The Receiver receives a message, either recvmsg0 or recvmsg1
28 *(with 0 and 1 as additional bits)
29 *If the corresponding bit is different to the one in the last message it
30 *sends an acknowledgment and delivers the message to the application.
31 *If the corresponding bit is equal to the one of the last message it just
32 *sends the acknowledgment, because it already got that message.
33 *The agent Receiver0 is waiting for a message with the corresponding bit 0,
34 *agent Receiver1 waits for a message with corresponding bit 1.
35 agent Receiver0 = recvmsg0.'sendack0.'deliver0.Receiver1
36                  + recvmsg1.'sendack1.Receiver0;
37 agent Receiver1 = recvmsg1.'sendack1.'deliver1.Receiver0
38                  + recvmsg0.'sendack0.Receiver1;
39
40 agent Timeout = 'timeout.Timeout;
41
42
43 *The agent PerfectMedium is always delivering the sent message. It does not
44 *allow the sender to overwrite the message. Therefore it will deadlock
45 *if the sender has a timeout.
46 *If the timeouts are switched off it works perfectly.
47 agent PerfectMedium = sendmsg1.PerfectMediumMsg1 + sendmsg0.PerfectMediumMsg0
48                  + sendack0.PerfectMediumAck0 + sendack1.PerfectMediumAck1;
49 agent PerfectMediumMsg0 = 'recvmsg0.PerfectMedium;
50 agent PerfectMediumMsg1 = 'recvmsg1.PerfectMedium;
51 agent PerfectMediumAck0 = 'recvack0.PerfectMedium;
52 agent PerfectMediumAck1 = 'recvack1.PerfectMedium;
```

```
53
54 *The agent Medium is a perfect medium that does not lose messages, but
55 *messages can be overwritten by the sender.
56 *This is important because the sender could otherwise deadlock if a
57 *timeout happens.
58 *If the Timeout deadlock is left out(like in the agent ABP) it is not
59 *necessary for the medium to allow overwriting of messages.
60 agent Medium = sendmsg1.MediumMsg1 + sendmsg0.MediumMsg0
61                 + sendack0.MediumAck0 + sendack1.MediumAck1;
62 agent MediumMsg0 = 'recvmsg0.Medium + sendmsg1.MediumMsg1
63         + sendmsg0.MediumMsg0 + sendack0.MediumAck0 + sendack1.MediumAck1;
64 agent MediumMsg1 = 'recvmsg1.Medium + sendmsg1.MediumMsg1
65         + sendmsg0.MediumMsg0 + sendack0.MediumAck0 + sendack1.MediumAck1;
66 agent MediumAck0 = 'recvack0.Medium + sendmsg1.MediumMsg1
67         + sendmsg0.MediumMsg0 + sendack0.MediumAck0 + sendack1.MediumAck1;
68 agent MediumAck1 = 'recvack1.Medium + sendmsg1.MediumMsg1
69         + sendmsg0.MediumMsg0 + sendack0.MediumAck0 + sendack1.MediumAck1;
70
71 *The agent LossyMedium is a medium that might lose messages
72 agent LossyMedium = sendmsg1.LossyMediumMsg1 + sendmsg0.LossyMediumMsg0
73                 + sendack0.LossyMediumAck0 + sendack1.LossyMediumAck1;
74 agent LossyMediumMsg0 = 'recvmsg0.LossyMedium + tau.LossyMedium;
75 agent LossyMediumMsg1 = 'recvmsg1.LossyMedium + tau.LossyMedium;
76 agent LossyMediumAck0 = 'recvack0.LossyMedium + tau.LossyMedium;
77 agent LossyMediumAck1 = 'recvack1.LossyMedium + tau.LossyMedium;
78
79
80 *The agent DuplicatingMedium can lose and duplicate sent messages
81 agent DuplicatingMedium = sendmsg1.DuplicatingMediumMsg1
82         + sendmsg0.DuplicatingMediumMsg0 + sendack0.DuplicatingMediumAck0
83         + sendack1.DuplicatingMediumAck1;
84 agent DuplicatingMediumMsg0 = 'recvmsg0.DuplicatingMediumMsg0
85                 + tau.DuplicatingMedium;
86 agent DuplicatingMediumMsg1 = 'recvmsg1.DuplicatingMediumMsg1
87                 + tau.DuplicatingMedium;
88 agent DuplicatingMediumAck0 = 'recvack0.DuplicatingMediumAck0
89                 + tau.DuplicatingMedium;
90 agent DuplicatingMediumAck1 = 'recvack1.DuplicatingMediumAck1
91                 + tau.DuplicatingMedium;
92
93
94 *The agent ABP connects a Sender and a Receiver, both starting with
95 *the additional bit 0, using a perfect medium and without timeouts
96 agent ABP = (Sender0|Receiver0|PerfectMedium)\L;
97
98 *ABP0 also uses a perfect medium, but the sender might get timeouts
99 *and resend the last message.
100 agent ABP0 = (Sender0|Receiver0|Timeout|Medium)\L;
101
102 *ABP1 uses the LossyMedium that might lose messages.
103 agent ABP1 = (Sender0|Receiver0|Timeout|LossyMedium)\L;
104
105 *ABP2 uses the DuplicatingMedium that might duplicate and
106 *lose sent messages.
107 agent ABP2 = (Sender0|Receiver0|Timeout|DuplicatingMedium)\L;
108
```

```
109 *ABP3 shows that the PerfectMedium in combination with timeouts
110 *will deadlock and therefore not satisfy the specification.
111 agent ABP3 = (Sender0|Receiver0|PerfectMedium|Timeout)\L;
112 set L = {sendmsg0,sendmsg1,recvmsg0,recvmsg1,sendack0,
113 sendack1,recvack0,recvack1,timeout};
114
115 *The Specification for the alternating bit protocol:
116 *The agent is alternately delivering messages with
117 *additional bits 0 and 1.
118 agent ABPSpec0 = 'deliver0.ABPSpec1;
119 agent ABPSpec1 = 'deliver1.ABPSpec0;
```

2 Verification

To verify the protocol with various different channels weak bisimilarity checking ('eq' in the concurrency workbench) is used. We will see that the agents ABP, ABP0, ABP1, and ABP2 are weakly bisimilar to the specification ABPSpec0 and ABP3 is not because it contains a deadlock state when a timeout occurs.

Listing 2: Weak Bisimilarity

```
1  Command: eq(ABP,ABPSpec0);
2  true
3  Command: eq(ABP0,ABPSpec0);
4  true
5  Command: eq(ABP1,ABPSpec0);
6  true
7  Command: eq(ABP2,ABPSpec0);
8  true
9  Command: eq(ABP3,ABPSpec0);
10 false
```

Weak bisimilarity is used because the internal tau-actions should be abstracted away in the equivalence check, as they are unobservable. Strong bisimilarity would be too strong and the implementation could not satisfy the same properties as the specification with respect to strong bisimilarity.

Listing 3: Deadlocks

```
1  Command: deadlocks ABP;
2  None.
3  Command: deadlocks ABP0;
4  None.
5  Command: deadlocks ABP1;
6  None.
7  Command: deadlocks ABP2;
8  None.
9  Command: deadlocks ABP3;
10 --- tau tau tau tau tau ---> (Sender0 | 'sendack0.'deliver0.Receiver1
11 | PerfectMediumMsg0 | Timeout)\L
12 --- tau tau tau tau tau tau 'deliver0 tau tau tau --->
13 (Sender1 | 'sendack1.'deliver1.Receiver0 | PerfectMediumMsg1 | Timeout)\L
```

```
14 --- tau tau tau tau ---> (Sender0Waiting | 'sendack0.'deliver0.Receiver1
15 | PerfectMediumMsg0 | Timeout)\L
16 --- tau tau tau tau tau tau 'deliver0 tau tau 'deliver1 --->
17 (Sender1 | Receiver0 | PerfectMediumAck1 | Timeout)\L
18 --- tau tau tau tau 'deliver0 ---> (Sender0 | Receiver1
19 | PerfectMediumAck0 | Timeout)\L
20 --- tau tau tau tau tau tau 'deliver0 tau tau ---> (Sender1Waiting
21 | 'sendack1.'deliver1.Receiver0 | PerfectMediumMsg1 | Timeout)\L
```

As seen above there are no deadlocks in ABP, ABP0, ABP1 and ABP2. Only ABP3(PerfectMedium and Timeouts) has deadlocks that happen when a timeout occurs an the sender tries to resend the last message.

2.1 Formulas for Deadlocks and Livelocks

Listing 4: Deadlocks

```
1  prop Poss(P) = min(Z.P | <<->>Z);*Poss is taken from the CWB-Examples
2  prop Deadlock = Poss([-]F);
3  Command: checkprop(ABP, Deadlock);
4  false
5
6  Command: checkprop(ABP0, Deadlock);
7  false
8
9  Command: checkprop(ABP1, Deadlock);
10 false
11
12 Command: checkprop(ABP2, Deadlock);
13 false
14
15 Command: checkprop(ABP3, Deadlock);
16 true
```

As can be seen only ABP3 has a deadlock state.
Next we will check which of the Implementations have livelocks. Livelocks are states from where an infinite number of tau-steps are possible.

Listing 5: Livelocks

```
1  *Livelock: infinite sequence of tau-steps,
2  *therefore a tau-step is possible and there exists a tau
3  *step after which the same property holds.
4  Command: prop Livelock = max(Z.<tau>T & <tau>Z);
5
6  Command: checkprop(ABP, Livelock);
7  false
8
9  Command: checkprop(ABP0, Livelock);
10 true
11
12 Command: checkprop(ABP1, Livelock);
13 true
14
15 Command: checkprop(ABP2, Livelock);
```

```
16 true
17
18 Command: checkprop(ABP3, Livelock);
19 false
```

As can be seen ABP and ABP3 don't have any livelocks. As intuitively suspected the implementations with faulty channels have livelocks (when every message gets lost) and the implementation with the channel that allows the overwriting of messages has livelocks (when the sender constantly gets timeouts and retransmits the message). Therefore the implementations with livelocks can't guarantee that a message will eventually be delivered. That is down to the quality of the channel (like in real life). If the channel loses every message, clearly no message can be delivered.

3 Conclusion

The implementation of the protocol satisfies the specification. The messages are delivered with alternating additional bit. In combination with faulty channels there are livelocks, but if the channel eventually doesn't lose a message the correctness of the transmission is preserved. So the livelock appears only in the situation where every message is lost, in which case every communication protocol will fail to deliver messages.
Therefore we have verified that this simple protocol for the retransmission of messages works and the order of the sequence of messages is preserved. We also verified that the protocol can handle different qualities of channels (perfect, lossy and duplicating channels). Working with the CWB was quite easy and comfortable (especially the functions to interactively simulate an LTS and to find deadlocks/distinguishing formulas).