# MiniProject: The Alternating Bit Protocol

Stephanie Stroka
stephanie.stroka@stud.sbg.ac.at

August 5, 2011

## 1 Introduction

This mini-project is part of the course *Introduction to Concurrency and Verification*. The goal was to model the Alternating Bit Protocol as a CCS system and to verify it with the Edinburgh Concurrency Workbench (CWB).

## 2 The Protocol

In the following, the CCS system of the Alternating Bit Protocol specification and implementation in CWB syntax is presented and explained. Figure 1 gives an overview of the CCS system in form of a labeled transition system. The protocol in CWB syntax is given in Listing 1.

Listing 1: The alternating bit protocol in CWB syntax

```
*************************************************
* The sender either sends a 0 or a 1 and has      *
* a timeout or receivesthe ACK from the receiver  *
*************************************************
agent S_0 = ('send_0.(timeout.S_0 + transmack_0.S_1));
agent S_1 = ('send_1.(timeout.S_1 + transmack_1.S_0));

agent S = S_0 + S_1;

*************************************************
* The timer sends out periodic timeout signals    *
*************************************************
agent Timer = 'timeout.Timer;

*************************************************
* The sender and receiver communicate through a   *
* channel with respective interfaces for the      *
* sender as well as for the receiver              *
*************************************************
agent Interface_S = send_0.('transm_0.Interface_S + Interface_S)
                  + send_1.('transm_1.Interface_S + Interface_S);
agent Interface_R = sendack_0.('transmack_0.Interface_R + Interface_R)
                  + sendack_1.('transmack_1.Interface_R + Interface_R);
```

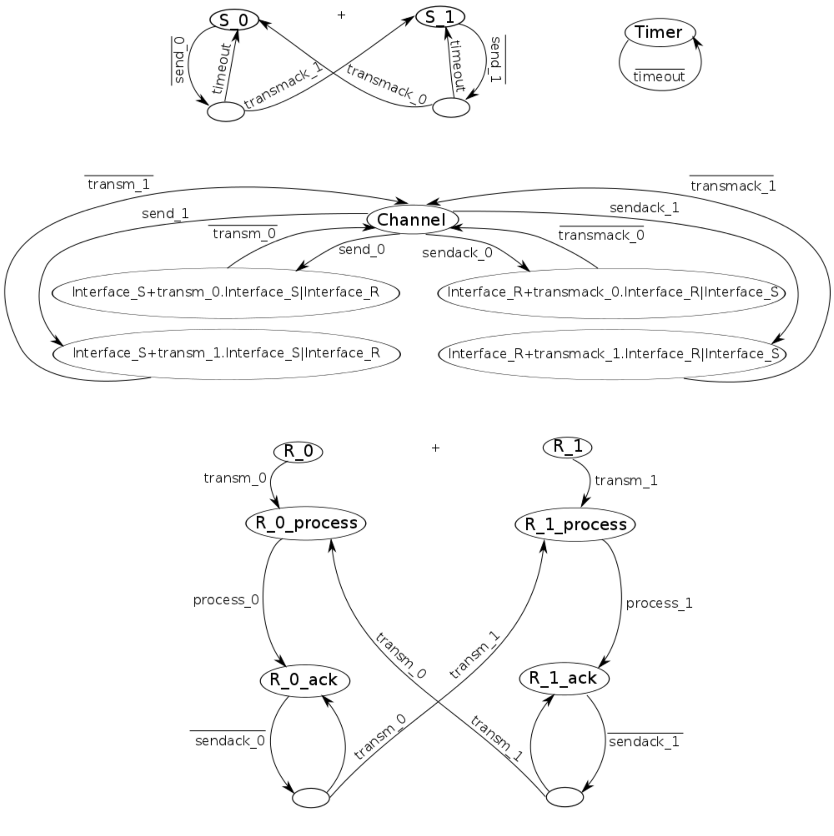


Figure 1: The alternating bit protocol

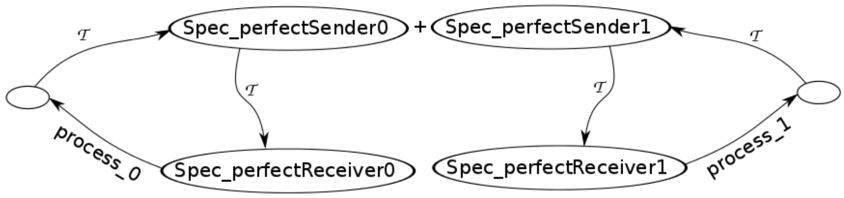


Figure 2: A perfect channel behavior. Messages do not get lost.

```
agent Channel = (Interface_S | Interface_R);

**************************************************
* The receiver processes the message when he      *
* receives a certain control bit (0 or 1) for the *
* first time. Then it sends an ACK to the         *
* interface. Subsequently received messages with  *
* the same control bit are just acknowledged.     *
**************************************************
agent R_0 = (transm_0.R_0_progress);
agent R_0_progress = progress_0.R_0_ack;
agent R_0_ack = 'sendack_0.(transm_0.R_0_ack + transm_1.R_1_progress);

agent R_1 = (transm_1.R_1_progress);
agent R_1_progress = progress_1.R_1_ack;
agent R_1_ack = 'sendack_1.(transm_1.R_1_ack + transm_0.R_0_progress);

agent R = (R_0 + R_1);

**************************************************
* The Alternating Bit Protocol sets the sender,   *
* receiver and channel in parallel to allow for   *
* communication. We restrict the communicating    *
* ports (send_[01], transm_[01], sendack_[01],    *
* transmack_[01], timeout) of the processes to be *
* only used in the Alternating Bit Protocol.      *
**************************************************
set Res = {send_0,send_1,transm_0,transm_1, sendack_0,
           sendack_1, transmack_0, transmack_1, timeout};
agent AltBitProt = (S | Timer | R | Channel)\Res;
```

Additionally to the alternating bit protocol implementation, specifications for perfect, lossy and lossy/duplicating channels have been modeled. Figures 2, 3 and 4 give an overview of the specifications.
The specifications have also been modeled in CWB and are listed in Listing 2.

Listing 2: The specifications in CWB syntax.

```
**************************************************
```

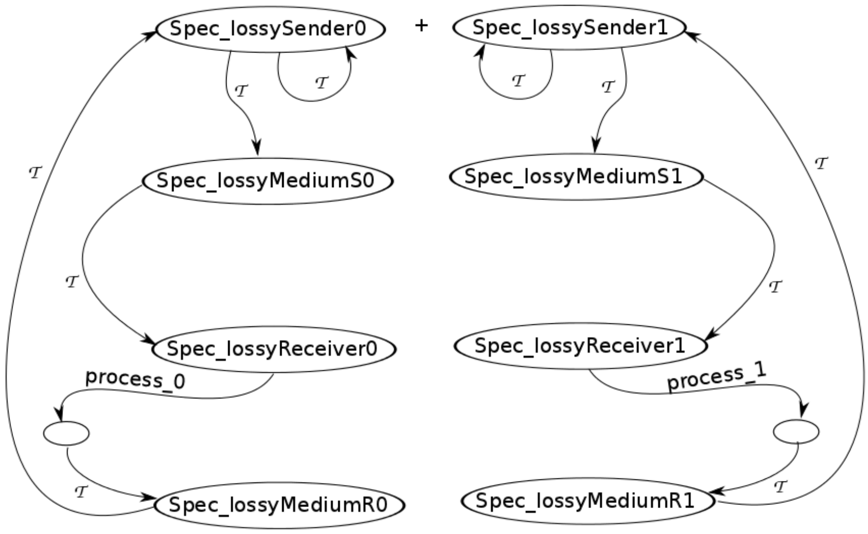


Figure 3: A lossy channel behavior. Messages get lost without warning.

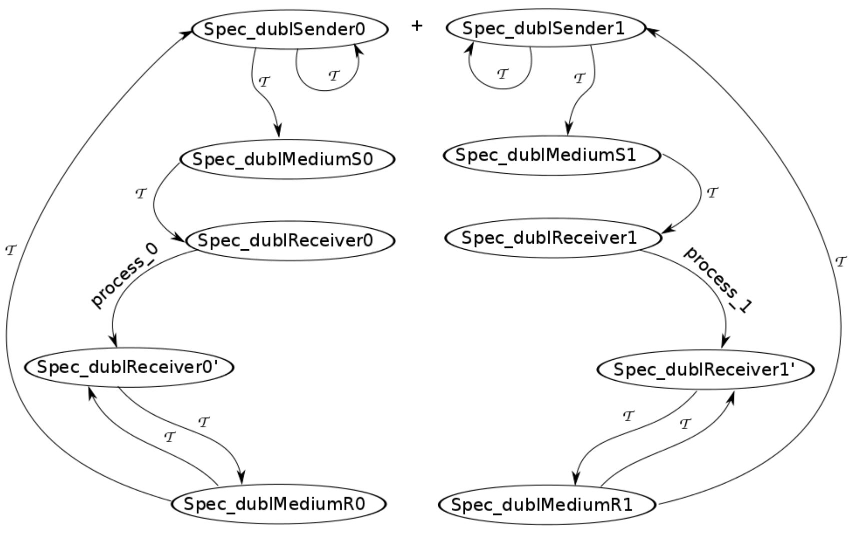


Figure 4: A lossy and dublicating channel behavior. Messages get lost and the receiver must be able to handle multiple messages with the same control bit.

```
* The specification of a perfect sender           *
***************************************************
agent Spec_perfectSender0 = tau.Spec_perfectReceiver0;
agent Spec_perfectSender1 = tau.Spec_perfectReceiver1;
agent Spec_perfectReceiver0 = progress_0.tau.Spec_perfectSender1;
agent Spec_perfectReceiver1 = progress_1.tau.Spec_perfectSender0;

***************************************************
* The specification of a lossy sender             *
***************************************************
agent Spec_lossySender = Spec_lossySender0 + Spec_lossySender1;
agent Spec_lossySender0 = tau.Spec_lossyMediumS0;
agent Spec_lossySender1 = tau.Spec_lossyMediumS1;
agent Spec_lossyMediumS0 = tau.Spec_lossyReceiver0 + Spec_lossySender0;
agent Spec_lossyMediumS1 = tau.Spec_lossyReceiver1 + Spec_lossySender1;
agent Spec_lossyMediumR0 = tau.Spec_lossySender1;
agent Spec_lossyMediumR1 = tau.Spec_lossySender0;
agent Spec_lossyReceiver0 = progress_0.tau.Spec_lossyMediumR0;
agent Spec_lossyReceiver1 = progress_1.tau.Spec_lossyMediumR1;

***************************************************
* The specification of a lossy sender.            *
* The receiver tolerated dublications.            *
***************************************************
agent Spec_dublSender = Spec_dublSender0 + Spec_dublSender1;
agent Spec_dublSender0 = tau.Spec_dublMediumS0;
agent Spec_dublSender1 = tau.Spec_dublMediumS1;
agent Spec_dublMediumS0 = tau.Spec_dublReceiver0 + Spec_dublSender0;
agent Spec_dublMediumS1 = tau.Spec_dublReceiver1 + Spec_dublSender1;
agent Spec_dublMediumR0 = tau.Spec_dublSender1 + tau.Spec_dublReceiver0';
agent Spec_dublMediumR1 = tau.Spec_dublSender0 + tau.Spec_dublReceiver1';
agent Spec_dublReceiver0 = progress_0.Spec_dublReceiver0';
agent Spec_dublReceiver0' = tau.Spec_dublMediumR0;
agent Spec_dublReceiver1 = progress_1.Spec_dublReceiver1';
agent Spec_dublReceiver1' = tau.Spec_dublMediumR1;
```

# 3 Verifications

The protocol implementation has been checked against the specifications by using weak bisimulation equivalence. CWB offers the command `eq(A,B):` to check two processes A and B for weak bisimilarity. Listing 3 provides an overview of the commands used in the project. In this example, all commands verify with `true` that the implementation and the specifications are weakly bisimilar.

Listing 3: Checking for weak bisimiliarity.

```
***************************************************
* Checking of weak bisimilarity between the       *
* alternating bit protocol and the perfect, lossy, *
* and dublicating sender specifications.          *
```

```
*************************************************
eq(Spec_perfectSender,AltBitProt);
eq(Spec_lossySender,AltBitProt);
eq(Spec_dublSender,AltBitProt);
```

Weak bisimulation equivalence has been chosen, because the Alternating Bit Protocol makes immense use of communication between processes, which is represented as $\tau$-steps. The specifications, on the other hand, do not use $\tau$-transitions as heavily as the protocol implementation. Transitions are made directly from one process to another. In fact, the specification could also be modeled without any $\tau$-steps.

# 4 Deadlocks & Livelocks

This Alternating Bit Protocol implementation does not contain any states in which no further transition is possible. Thus, no deadlocks can be found. Though the timer introduces livelocks, which means that there exist some states in which only $\tau$-steps are possible.
The deadlock and livelock behavior has been defined as HML formulae as seen in Listing 4.

Listing 4: Checking for deadlock and livelock behavior..

```
*************************************************
* Checking for deadlocks.                       *
*************************************************
prop Bx(P) = max(Z.P & [-]Z);

cp(AltBitProt,Bx(<->T));
cp(Spec_perfectSender,,Bx(<->T));
cp(Spec_dublSender,,Bx(<->T));
cp(Spec_lossySender,,Bx(<->T));

*************************************************
* Checking for lifelocks.                       *
*************************************************
prop Ev(P) = min(Z.P | ([-]Z & <->T));

cp(AltBitProt,Ev(<progress_0>T | <progress_1>T));
cp(Spec_perfectSender,Ev(<progress_0>T | <progress_1>T));
cp(Spec_dublSender,Ev(<progress_0>T | <progress_1>T));
cp(Spec_lossySender,Ev(<progress_0>T | <progress_1>T));
```

The deadlock check has been defined as formula that checks whether *after every possible transition, there exists another transition*, and the livelock check has been defined as formula that checks whether *the transitions* `process_0` or `process_1` *are still possible after any transition*.
As expected,the deadlock check returns `true` as a result which means that no deadlocks could be found. The livelock check returns `false`, because livelocks are possible. The perfect channel (Figure 2) is the only process that does not contain any livelocks, since messages will never get lost.

## 5 Conclusion

The suggested protocol satisfies the expected behavior. A simple visualization of each process transitions has been computed with the command `vs(5,AltBitProt)`, `vs(5,Spec_perfectSender)`, `vs(5,Spec_lossySender)` and `vs(5,Spec_dublSender)`. The output is given in Listings 5, 6, 7, and 8.

Listing 5: Possible transitions for the AltBitProt process.

```
Command: vs(5,Spec_perfectSender);
=== process_0 process_1 process_0 process_1 process_0 ===>
=== process_1 process_0 process_1 process_0 process_1 ===>
```

Listing 6: Possible transitions for the Spec_perfectSender process.

```
Command: vs(5,Spec_perfectSender);
=== process_0 process_1 process_0 process_1 process_0 ===>
=== process_1 process_0 process_1 process_0 process_1 ===>
```

Listing 7: Possible transitions for the Spec_lossySender process.

```
Command: vs(5,Spec_lossySender);
=== process_0 process_1 process_0 process_1 process_0 ===>
=== process_1 process_0 process_1 process_0 process_1 ===>
```

Listing 8: Possible transitions for the Spec_dublSender process.

```
Command: vs(5,Spec_dublSender);
=== process_0 process_1 process_0 process_1 process_0 ===>
=== process_1 process_0 process_1 process_0 process_1 ===>
```

A possible protocol extension would be the addition of data and data-checksums in the message, which could then be verified by the receiver before sending an ACK.