Alternating-bit protocol

Modeled and verified in Concurrency Workbench

Michael Lippautz

michael.lippautz@gmail.com

July 15, 2011

This paper serves as mini project for the course **Introduction to Concurrency and Verification, Summer 2011**. It provides an implementation and specification of the alternating-bit protocol in the Calculus of Communication Systems (CSS) and a verification using Concurrency Workbench (CWB).

1. Introduction

The alternating bit protocol is a protocol that includes a sender, a receiver and a medium, also called channel. The sender acknowledges a message from some application, sends it to the receiver over a channel. The receiver then passes the message up (actually receives it) to an application. Furthermore (from the task definition):

- Each message sent by S contains an additional protocol bit, 0 or 1.
- When S sends a message , it sends it repeatedly (with its corresponding bit) until receiving an acknowledgement (ACK) from R that contains the same protocol bit as the message being sent.
- When R receives a message, it sends an ACK to S and includes the protocol bit of the message received. When a message is received for the first time, the receiver delivers it for processing, while subsequent message with the same bit are simply acknowledged.
- When S receives an ACK containing the same bit as the message it is currently transmitting, it stops transmitting that message, flips the protocol bit, and repeats the protocol for the next message.

Figure 1 shows the general architecture of the components including the sending and receiving applications, the actual sender S and receiver R, and a channel C.

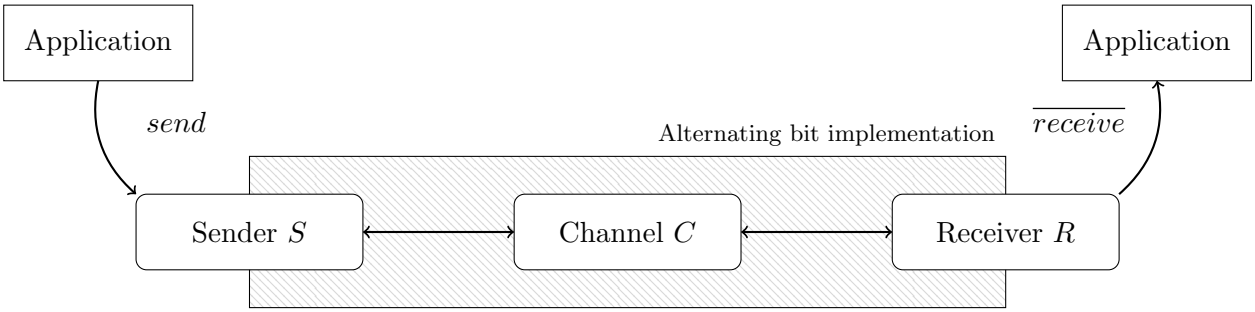


Figure 1: Alternating bit architecture

Note

All commands and explanations for CWB have been taken from [MS99] and [HÜ98]. Of course, the main book [AIK+07] and the lecture notes have also been used.

2. CSS specification

From the description above one can model the specification that should be observed from the "outside" of the system. This specification is later used to prove observational equivalence (i.e. a weak bisimilarity). In CSS notation this process can be modeled as in (1). The specification formulates that only *send* and $\overline{receive}$ can be observed from the "outside".

$$AlternatingBitSpec \stackrel{def}{=} send.\overline{receive}.AlternatingBitSpec \tag{1}$$

Figure 2 shows the corresponding LTS of the *AlternatingBitSpec* CSS process. This process will also called be specification or *Spec* throughout this paper.

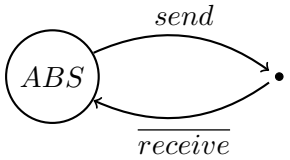


Figure 2: LTS for *AlternatingBitSpec* (as ABS)

3. CSS implementation

The following section deals with the CSS implementation of the sender S, receiver R and various channels C_i. Moreover a timer process T will be used to trigger re-sends.

Before describing the actual sender, receiver and the channels the actions are briefly illustrated:

- $sdata_{0|1}$ is used to transfer data with the corresponding bit set from the sender S to any of the channels C_i.
- $rdata_{0|1}$ is used to transfer data with the corresponding bit set from any of the channels C_i to the receiver R.
- $sack_{0|1}$ is used to transfer an acknowledgement with the corresponding bit set from the receiver R to any of the channels C_i.
- $rack_{0|1}$ is used to transfer an acknowledgement with the corresponding bit set from any of the channels C_i to the sender S.
- $timeout_{0|1}$ is used to indicate a re-send action.

All actions exist in their incoming (a) and outgoing ($\overline{a}$) variants.

3.1. Sender S

Since the sender should send a message with either a 0 or 1 bit set, the basic sending states will be divided into two parts. In more detail the states of the sender S are defined by $\{S_0, S_0', S_1, S_1'\}$.

$$S_0 \stackrel{def}{=} send.\overline{sdata_0}.S_0' \tag{2}$$

$$S_0' \stackrel{def}{=} timeout_0.\overline{sdata_0}.S_0' + rack_0.S1 + rack_1.S_0' \tag{3}$$

$$S_1 \stackrel{def}{=} send.\overline{sdata_1}.S_1' \tag{4}$$

$$S_1' \stackrel{def}{=} timeout_1.\overline{sdata_1}.S_1' + rack_1.S0 + rack_0.S_1' \tag{5}$$

Figure 3 shows the set of processes that define S. The "starting" state is S_0, whenever the sender S (S_0 in the start) receives a message, it signals this with the input action *send*. It emits the message onto the channel through the $sdata_0$ action and switches the state into S_0' (2). In S_0' a resend may be triggered by receiving a $timeout_0$. Otherwise the sender waits for either $rack_0$ indicating an acknowledgement with a 0 bit set, or $rack_1$ for the corresponding 1 bit set. If it receives a $rack_0$ it switches into S_1, the symmetric system which only differs in handling 1-bits. If $rack_1$ is received it is ignored (3). The other case is just symmetric (4, 5).

In addition to the actual sending processes the sender also may need (see the actual systems for a description why and why not) a Timer T that triggers the re-sends. A single timer that using non-determinism can be used to trigger either $timeout_0$ or $timeout_1$ (6).

$$T \stackrel{def}{=} \overline{timeout_0}.T + \overline{timeout_1}.T \tag{6}$$

Figure 4 illustrates the LTS of T.

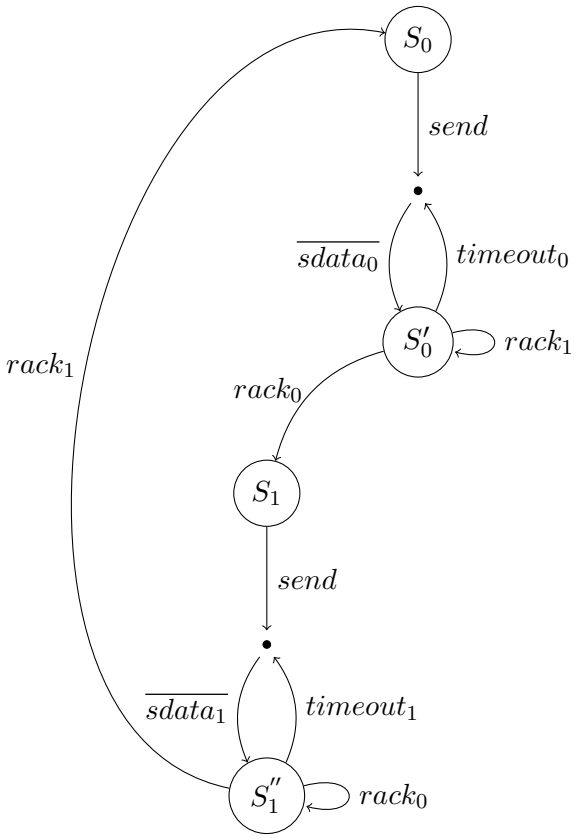


Figure 3: LTS of the sender S

3.2. Channels C_i

This section contains the various different transmission channels that are required by the task definition:

- A perfect channel, $i = perfect$
- A lossy channel that may loose packets that have been sent, $i = lossy$
- A faulty channel that may additionally to loosing packets, also be able to duplicate them, $i = faulty$

3.2.1. Perfect channel $C_{perfect}$

The perfect channel $C_{perfect}$ consists out of two perfectly transmitting sub channels, a sending one $C_{p,send}$ and a receiving one $C_{p,rec}$, that run in parallel.

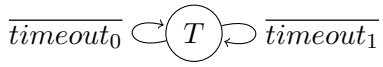


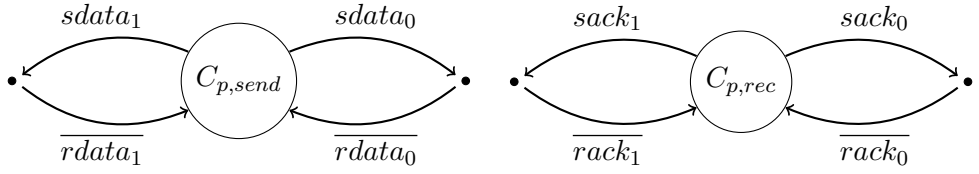
Figure 4: LTS of the timer T

$$C_{p,send} \stackrel{def}{=} sdata_0.\overline{rdata_0}.C_{p,send} + sdata_1.\overline{rdata_1}.C_{p,send} \tag{7}$$

$$C_{p,rec} \stackrel{def}{=} sack_0.\overline{rack_0}.C_{p,rec} + sack_1.\overline{rack_1}.C_{p,rec} \tag{8}$$

$$C_{perfect} \stackrel{def}{=} C_{p,send} | C_{p,rec} \tag{9}$$

The corresponding LTS are illustrated in Figure 5. A perfect channel (sub channel) receives the data using $sdata_0$ and emits then $\overline{rdata_0}$ to indicate that the data should be received by the receiver (7) (for the 0 bits). The receiving sub channel is just symmetric (8). The overall channel is defined by running the receiving and sending side in parallel (9).



(a) LTS for sending sub channel $C_{p,send}$ (b) LTS for the receiving sub channel $C_{p,rec}$

Figure 5: LTS for the sub channels of $C_{perfect}$

3.2.2. Lossy channel C_{lossy}

The lossy channel C_{lossy} consists out of two lossy transmitting sub channels, a sending one $C_{l,send}$ and a receiving one $C_{l,rec}$, that run in parallel. Both sub channels may receive data as in the perfect channel but have in contrast to the perfect ones, they have the possibility to drop the packets instead of sending $rdata$s or $rack$s.

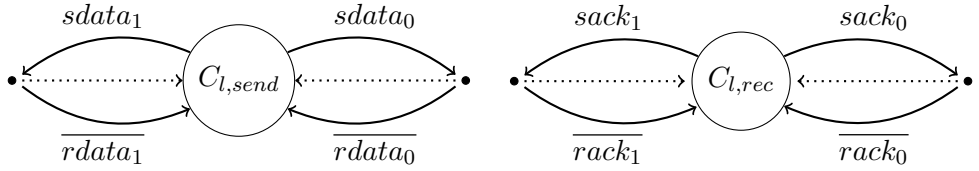
$$\begin{aligned} C_{l,send} \stackrel{def}{=}\ & sdata_0.(\overline{rdata_0}.C_{l,send} + C_{l,send}) \\ & + sdata_1.(\overline{rdata_1}.C_{l,send} + C_{l,send}) \end{aligned} \tag{10}$$

$$\begin{aligned} C_{l,rec} \stackrel{def}{=}\ & sack_0.(\overline{rack_0}.C_{l,send} + C_{l,send}) \\ & + sack_1.(\overline{rack_1}.C_{l,send} + C_{l,send}) \end{aligned} \tag{11}$$

$$C_{lossy} \stackrel{def}{=} C_{l,send} | C_{l,rec} \tag{12}$$

The corresponding LTS are illustrated in Figure 6. It is important that the branching takes place as late as possible, i.e. after receiving $sdata_0$ or $sdata_1$. Otherwise a deadlock

possibility would be created, since the channel would "have to decide" upon receiving the value which branch should be taken. Again the sending and receiving channels are symmetric, besides the different action labels.



(a) LTS for sending subchannel $C_{l,send}$ (b) LTS for the receiving subchannel $C_{l,rec}$

Figure 6: LTS for the sub channels of C_{lossy}

3.2.3. Faulty channel C_{faulty}

The faulty channel is similar to the lossy one, with the difference that it may also generate several copies of any outgoing messages, i.e. $rdata_{0|1}$ and $rack_{0|1}$. The channel is defined by the following CSS processes.

$$C_{f,send} \stackrel{def}{=} sdata_0.C_{f,send,0} + sdata_1.C_{f,send,1} \tag{13}$$

$$C_{f,send,0} \stackrel{def}{=} \overline{rdata_0}.C_{f,send,0} + C_{f,send} \tag{14}$$

$$C_{f,send,1} \stackrel{def}{=} \overline{rdata_1}.C_{f,send,1} + C_{f,send} \tag{15}$$

$$C_{f,rec} \stackrel{def}{=} sack_0.C_{f,rec,0} + sack_1.C_{f,rec,1} \tag{16}$$

$$C_{f,rec,0} \stackrel{def}{=} \overline{rack_0}.C_{f,rec,0} + C_{f,rec} \tag{17}$$

$$C_{f,rec,1} \stackrel{def}{=} \overline{rack_1}.C_{f,rec,1} + C_{f,rec} \tag{18}$$

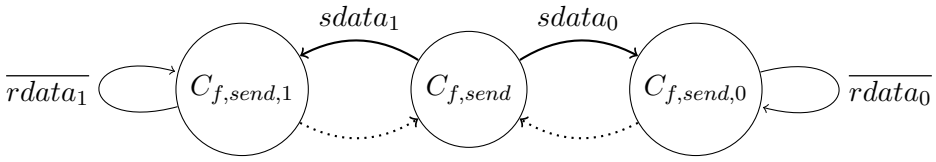
Figure 7 shows the sub channels as LTSes that are defined by the CSS processes above. It is important that the $rdata_{0|1}$ and $rack_{0|1}$ are performed under a *, i.e. can occor $0, 1 \ldots n$ times with $n \geq 0$.

3.3. Receiver R

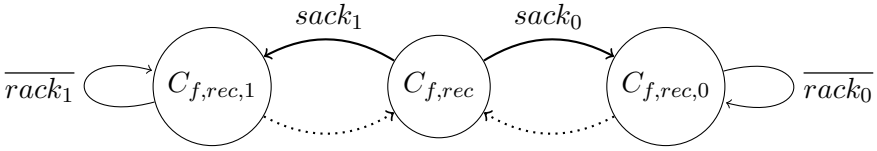
The receiver R is exactly implemented as described in the task definition. It performs a $\overline{receive}$ action whenever a message is received the first time and acknowledges it. Otherwise it just acknowledges the message that is received (which is either a duplicate or a re-send).

$$R_0 \stackrel{def}{=} rdata_0.\overline{receive}.\overline{sack_0}.R_1 + rdata_1.\overline{sack_1}.R_0 \tag{19}$$

$$R_1 \stackrel{def}{=} rdata_1.\overline{receive}.\overline{sack_1}.R_0 + rdata_0.\overline{sack_0}.R_1 \tag{20}$$



(a) LTS for sending subchannel $C_{f,send}$



(b) LTS for the receiving subchannel $C_{f,rec}$

Figure 7: LTS for sub channels of C_{faulty}

Figure 8 once more illustrates the receiver's LTS.

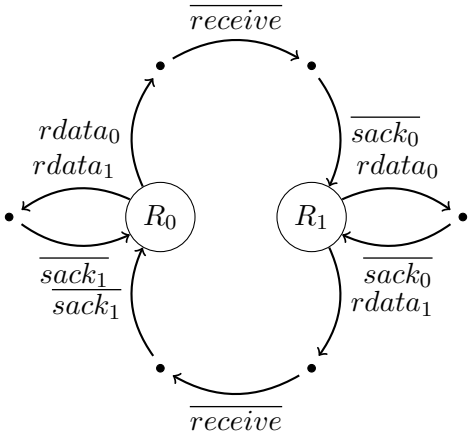


Figure 8: The receiver R

3.3.1. The implementations

In order to define the systems we also need to restrict communication. Thus a set *Restrictions* is defined as follows:

$$\begin{aligned} Restrictions = \{&sdata_0, sdata_1, rdata_0, rdata_1, sack_0, sack_1, rack_0, rack_1, \\ &timeout_0, timeout_1\} \end{aligned} \tag{21}$$

The implementations defined are:

- A perfect implementation utilizing also the timer T:

$$Impl_{perfect,1} \stackrel{def}{=} (S_0|T|C_{perfect}|R0)\backslash Restrictions \tag{22}$$

- A perfect implementation not utilizing the timer T:

$$Impl_{perfect,2} \stackrel{def}{=} (S_0|C_{perfect}|R0)\backslash Restrictions \tag{23}$$

- A lossy implementation:

$$Impl_{lossy} \stackrel{def}{=} (S_0|T|C_{lossy}|R0)\backslash Restrictions \tag{24}$$

- And a faulty implementation:

$$Impl_{faulty} \stackrel{def}{=} (S_0|T|C_{faulty}|R0)\backslash Restrictions \tag{25}$$

4. Verification using CWB

4.1. Implementation in CWB

Before checking any equivalences or deadlocks on the protocol, the following listing defines it using CWB syntax. Basically, this is a one-to-one translation from the already defined CSS processes.

```
1  ****************************************************************************
2  *                          Alternating bit protocol
3  ****************************************************************************
4
5  ****************************************************************************
6  * Definition of the sender
7  ****************************************************************************
8  agent S0 = send.'sdata0.S0';
9  agent S0' = timeout0.'sdata0.S0' + rack0.S1 + rack1.S0';
10 agent S1 = send.'sdata1.S1';
11 agent S1' = timeout1.'sdata1.S1' + rack1.S0 + rack0.S1';
12
13 * The timer that may be used in parallel to trigger resends
14 agent T = 'timeout0.T + 'timeout1.T;
15
16 ****************************************************************************
17 * Definition of the receiver
18 ****************************************************************************
19 agent R0    = rdata0.'receive.'sack0.R1 + rdata1.'sack1.R0;
20 agent R1    = rdata1.'receive.'sack1.R0 + rdata0.'sack0.R1;
21
22 ****************************************************************************
23 * Definition of a perfect channel
24 ****************************************************************************
25 agent Cp_send = sdata0.'rdata0.Cp_send + sdata1.'rdata1.Cp_send;
```

```
agent Cp_rec = sack0.'rack0.Cp_rec + sack1.'rack1.Cp_rec;
agent Cperfect = (Cp_send | Cp_rec);

******************************************************************************
* Definition of a lossy channel
******************************************************************************
agent Cl_send = sdata0.('rdata0.Cl_send + Cl_send) + sdata1.('rdata1.Cl_send
  + Cl_send);
agent Cl_rec = sack0.('rack0.Cl_rec + Cl_rec) + sack1.('rack1.Cl_rec
  + Cl_rec);
agent Clossy = (Cl_send | Cl_rec);

******************************************************************************
* Definition of a faulty channel
******************************************************************************
agent Cf_send = sdata0.Cf_send0 + sdata1.Cf_send1;
agent Cf_send0 = 'rdata0.Cf_send0 + Cf_send;
agent Cf_send1 = 'rdata1.Cf_send1 + Cf_send;
agent Cf_rec = sack0.Cf_rec0 + sack1.Cf_rec1;
agent Cf_rec0 = 'rack0.Cf_rec0 + Cf_rec;
agent Cf_rec1 = 'rack1.Cf_rec1 + Cf_rec;
agent Cfaulty = (Cf_send | Cf_rec);

******************************************************************************
* Internals that need to be hidden to force communication
******************************************************************************
set Restrictions = {sdata0, sdata1, rack0, rack1, rdata0, rdata1, sack0,
  sack1, timeout0, timeout1};

******************************************************************************
* Perfect implementation suffering from a deadlock, because the timer
* triggers resends while no packets are lost. This causes a deadlock on the
* proposed perfect channel
******************************************************************************
agent Impl_Perfect1  = (S0 | Timer | Cperfect | R0) \ Restrictions;

******************************************************************************
* Perfect implementation without timer and thus without deadlock.
******************************************************************************
agent Impl_Perfect2 = (S0 | Cperfect | R0) \ Restrictions;

******************************************************************************
* Lossy implementation
******************************************************************************
agent Impl_Lossy = (S0 | Timer | Clossy | R0) \ Restrictions;

******************************************************************************
* Faulty implementation
******************************************************************************
agent Impl_Faulty = (S0 | Timer | Cfaulty | R0) \ Restrictions;

******************************************************************************
* The specification
******************************************************************************
```

```
80 agent Spec = send.'receive.Spec;
```

4.2. Verification of the implementation

In order to verify the implementation against the specification the weak bisimilarity has been chosen as equivalence. It provides the observational equivalence by abstracting away any τ sequences. Of course, strong bisimilarity cannot be used, as the specification does not provide any intermediate steps that would correspond to the τ sequences of the implementations.

4.2.1. Verification using weak bisimilarity in CWB

In order to verify the implementations the command `eq` is used. It checks two CCS processes against weak bisimilarity ($\approx$).

```
Command: input  ``alternating-bit/alternating-bit.cwb'';
Command: eq(Spec, Impl_Perfect1);
false
Command: eq(Spec, Impl_Perfect2);
true
Command: eq(Spec, Impl_Lossy);
true
Command:  eq(Spec, Impl_Faulty);
true
```

Before going into details, CWB shows that

$$AlternatingBitSpec \approx Impl_{perfect,2} \approx Impl_{lossy} \approx Impl_{faulty}$$

CWB also shows that

$$AlternatingBitSpec \not\approx Impl_{perfect,1}$$

The following section shows why, $Impl_{perfect,1}$ is not weakly bisimilar with the specification.

4.3. Deadlocks

As already proposed, $Impl_{perfect,1}$, the implementation of the perfect channel using a timer is not weakly bisimilar to the provided specification. This is a result of the deadlock capability of this implementation:

```
Command: fd Impl_Perfect1;
--- send tau tau 'receive tau tau send tau tau tau tau 'receive tau tau tau tau
send tau tau tau tau tau 'receive ---> ('sdata0.S0' | Timer | ('rdata0.Cp_send |
'rack1.Cp_rec) | 'sack0.R1)\Restrictions
--- send tau tau tau tau tau 'receive tau tau tau tau tau tau send tau tau --->
('sdata1.S1' | Timer | ('rdata1.Cp_send | 'rack0.Cp_rec) |
'sack0.R1)\Restrictions
--- send tau tau tau tau tau 'receive tau tau tau tau ---> ('sdata0.S0' | Timer
```

```
| ('rdata0.Cp_send | 'rack0.Cp_rec) | 'sack0.R1)\Restrictions
--- send tau tau 'receive tau tau send tau tau tau tau tau 'receive tau tau tau
tau ---> ('sdata1.S1' | Timer | ('rdata1.Cp_send | 'rack1.Cp_rec) |
'sack1.R0)\Restrictions
--- send tau tau 'receive tau tau send tau tau tau tau tau 'receive tau tau tau
tau tau tau send tau tau ---> ('sdata0.S0' | Timer | ('rdata0.Cp_send |
'rack1.Cp_rec) | 'sack1.R0)\Restrictions
--- send tau tau tau tau 'receive tau tau tau tau send tau tau tau tau tau
'receive ---> ('sdata1.S1' | Timer | ('rdata1.Cp_send | 'rack0.Cp_rec) |
'sack1.R0)\Restrictions
```

As the output of CWB shows there are several deadlock positions possible. Appendix A illustrates how one of these deadlocks can be reached showing the exact transitions (not only τs).

A distinguishing HML formulae can also be found.

```
Command: dfweak (Spec, Impl_Perfect1);
<<send>>[['receive]]<<send>>T
```

Moreover CWB can show that

$$AlternatingBitSpec \models < send > [\overline{receive}] < send > tt$$

and

$$Impl_{perfect,1} \not\models < send > [\overline{receive}] < send > tt$$

In CWB the function `cp` (or `checkproperty`) can be used to whether processes satisfy certain properties.

```
Command: cp(Spec, <<send>>[['receive]]<<send>>T);
true
Command: cp(Impl_Perfect1,  <<send>>[['receive]]<<send>>T);
false
```

The CWB output shows that while the specification satisfies the weak HML fomulae, the implementation does not.

All other implementations and the specification have no deadlocks:

```
Command: fd Spec;
None.
Command: fd Impl_Perfect2;
None.
Command: fd Impl_Lossy;
None.
Command: fd Impl_Faulty;
None.
```

4.4. Livelocks

CWB provides no internal function to directly check for livelocks. However, it is possible to define properties (even recursive ones) and check against these. In order to check for livelocks, it is necessary to define the properties in CWB syntax. As already proposed

in the ICV lecture, it is necessary to define a possibility HML formulae (26) that then is used to check for possible livelocks (27).

$$Pos(\varphi) \overset{min}{\equiv} \varphi \vee < Act > Pos(\varphi) \tag{26}$$

$$LivelockNow \overset{max}{\equiv} < \tau > LivelockNow \tag{27}$$

Both can be defined in CWB by the following terms. Note that the livelock has been defined as cyclecheck with a variable action `a`.

```
prop Pos(P) = min(Z.P | <->Z);
prop Cycle(x) = max(X . <x>T & [x]X);
```

In order to check for *LivelockNow* it is necessary to check for `Cycle(tau)` in CWB.

CWB can then be used to check for livelocks in all implementations and the specification.

```
Command: cp(Spec, Pos(Cycle(tau)));
false
Command: cp(Impl_Perfect1, Pos(Cycle(tau)));
false
Command: cp(Impl_Perfect2, Pos(Cycle(tau)));
false
Command: cp(Impl_Lossy, Pos(Cycle(tau)));
true
Command: cp(Impl_Faulty, Pos(Cycle(tau)));
true
```

As already expected,

$$AlternatingBitSpec \nvDash Pos(Cycle(\tau))$$
$$Impl_{perfect,1} \nvDash Pos(Cycle(\tau))$$
$$Impl_{perfect,2} \nvDash Pos(Cycle(\tau))$$

The specification cannot have any livelock, as its LTS is free of τ transitions. The implementations using the perfect channels can also not have any livelocks, since only the one using a timer could potentially have infinite τ sequences. But, as this one suffers from a deadlock it has also no livelock.

CWB also shows that

$$Impl_{lossy} \vDash Pos(Cycle(\tau))$$
$$Impl_{faulty} \vDash Pos(Cycle(\tau))$$

This also makes sense, since both implementations have τ cycles when passing messages over the channel. As the lossy and faulty channels could potentially loose every packet, these infinite τ loops are possible.

5. Conclusion

This mini project illustrates how the alternating-bit protocol could be implemented and specified using CSS processes. It also visualizes these processes showing parts of their corresponding LTSes. Moreover it shows that weak-bisimilarity can be used to show observational equivalence between some provided implementations and the specification. CWB has been used to show the limits of the implementation by providing information about deadlock and livelock capabilities.

References

[AIK+07] L. Aceto, Anna Ingólfsdóttir, Kim, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[Ḧ98] Hans Hüttel. *A short introduction to the Concurrency Workbench*, 1998.

[MS99] Faron Moller and Perdita Stevens. *The Edinburgh Concurrency Workbench user manual (Version 7.1)*, 1999.

A. $Impl_{perfect,1}$ deadlock example

The following listing shows one of the examples of the deadlock for $Impl_{perfect,1}$ and the exact (not only τ) transitions that have been taken.

```
--- send tau tau tau tau 'receive tau tau tau tau send tau tau tau tau tau 'receive
---> (sdata1.S1' | T | (rdata1.Cp_send | rack0.Cp_rec) | sack1.R0)\Restrictions

(S0 | T | Cperfect | R0) \ Restrictions
-(send)->
('sdata0.S0' | T | (Cp_send | Cp_rec) | R0) \ Restrictions
-(tau=sdata0)->
(S0'| T | ('rdata0.Cp_send | Cp_rec) | R0) \ Restrictions
-(tau=timeout0)->
('sdata0.S0' | T | ('rdata0.Cp_send | Cp_rec) | R0) \ Restrictions
-(tau=timeout0)->
('sdata0.S0' | T | (Cp_send | Cp_rec) | 'receive.'sack0.R1) \ Restrictions
-(tau=sdata0)->
(S0' | T | ('rdata0.Cp_send | Cp_rec) | 'receive.'sack0.R1) \ Restrictions
-('receive)->
(S0' | T | ('rdata0.Cp_send | Cp_rec) | 'sack0.R1) \ Restrictions

-(tau=sack0)->
(S0' | T | ('rdata0.Cp_send | 'rack0.Cp_rec) | R1) \ Restrictions
-(tau=rdata0)->
(S0' | T | (Cp_send | 'rack0.Cp_rec) | 'sack0.R1) \ Restrictions
-(tau=rack0)->
(S1 | T | (Cp_send | Cp_rec) | 'sack0.R1) \ Restrictions
-(tau=sack0)->
```

```
(S1 | T | (Cp_send | rack0.Cp_rec) | R1) \ Restrictions
-(send)->
('sdata1.S1' | T | (Cp_send | 'rack0.Cp_rec) | R1) \ Restrictions

-(tau=sdata1)->
(S1' | T | ('rdata1.Cp_send | 'rack0.Cp_rec) | R1) \ Restrictions
-(tau=rdata1)->
(S1' | T | (Cp_send | 'rack0.Cp_rec) | 'receive.'sack1.R0) \ Restrictions
-(tau=timeout1)->
('sdata1.S1' | T | (Cp_send | 'rack0.Cp_rec) | 'receive.'sack1.R0) \ Restrictions
-(tau=sdata1)->
(S1' | T | ('rdata1.Cp_send | rack0.Cp_rec) | 'receive.'sack1.R0) \ Restrictions
-(tau=timeout1)->
('sdata1.S1' | T | ('rdata1.Cp_send | 'rack0.Cp_rec) | 'receive.'sack1.R0) \ Restrictions
-('receive)->
('sdata1.S1' | T | ('rdata1.Cp_send | 'rack0.Cp_rec) | 'sack1.R0) \ Restrictions
```