VI. Bounded Re-transmission Protocol

In this chapter, we extend the *file transfer protocol* example of chapter 4. The added constraint with regards to the previous simple example is that we suppose now that the data and acknowledgment channels situated between the two sites are *unreliable*. As a consequence, the effect of the execution of the Bounded Re-transmission Protocol (for short BRP) is to only *partially* copy (but sometimes totally also) a sequential file from one site to another. The purpose of this example is precisely to study how we can cope with this kind of problems dealing with *fault tolerance* and how we can formally reason about them. Notice that in this chapter, we do not develop proofs as much as in the previous chapters: we only give some hints and let the reader developing the formal proof by himself. This example has been studied in many papers among which is the one by J.F. Groote and J.C. Van de Pool [1].

1 Informal Presentation of the Bounded Retransmission Protocol

1.1 Normal Behavior.

The sequential file to be transmitted is supposed to be transported piece by piece from one site, the sender site, to another one, the receiver site. For that purpose, the sender sends a certain data item on the, so-called, Data Channel connecting the sender to the receiver. As soon as the receiver receives this data item, it stores it in its own file and sends back an acknowledgment to the sender on the, so-called, Acknowledgment Channel connecting the receiver to the sender. As soon as the sender receives this acknowledgment, it sends the next data item, and so on. We suppose that the *last* data item sent by the sender contains a special information so that the receiver is able to know when the file transmission is completed. Notice that it has nevertheless to send a final acknowledgment.

All this can be represented in the diagram of Fig. 1 where the events (SND_snd, RCV_rcv, RCV_snd, and SND_rcv) are supposed to represent the various phases we have just described together with their synchronization as indicated by the arrows:

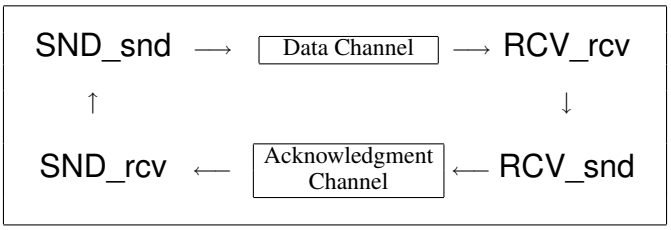


Fig. 1. Schematic View of the Transmission Protocol

What we have just described is the normal behavior of the protocol, where an entire file is transmitted from the sender to the receiver. We shall also describe below a degraded behavior, where the sender's file is transmitted only partially to the receiver due to some problems on the transmission channels.

1.2 Unreliability of the Communications.

The transmission channels (Data and Acknowledgement) situated between the sender and the receiver might be faulty: that is, some data items sent by the sender or some acknowledgments sent by the receiver

might be lost. In order to cope with the unreliability of these channels, the sender starts a timer when it sends a data item. This device is adjusted so that it wakes up the sender (provided, of course, it has not received an acknowledgment in the meantime) after a certain delay. This delay is *guaranteed* to be greater than the *maximum* delay dl which is required to first send a data item and subsequently receive back the corresponding acknowledgment. In other words, the sender can conclude that a message has necessarily been lost when the time is over, that is if a delay dl has passed since the last data item has been sent to the receiver without receiving a corresponding acknowledgment.

But, of course, when the timer wakes it up, the sender does not know whether the lost message corresponds to the data item that it has just sent or to the corresponding acknowledgment supposed to be sent back by the receiver. In any case, the sender re-transmits the previous data item and waits for the corresponding acknowledgment. This is the reason why the protocol is called a *re-transmission* protocol.

1.3 Protocol Abortion.

In case of successive losses of messages, the process of data re-transmission can be repeated a number of times: this is recorded at the sender site in the, so-called, *retry counter*. When this counter reaches a certain pre-defined limit M, the sender decides that the transmission is definitely broken and aborts the protocol (from its own point of view). This is the reason why the protocol is called the *bounded* re-transmission protocol.

The question that arises immediately is then, of course, that of the synchronization with the receiver. In other words, how does the receiver know that the protocol has aborted? Clearly, the sender cannot communicate any longer with the receiver in order to send it this abortion information because the communication is now broken.

This problem is solved by means of a second timer situated in the receiver's site. This timer is activated by the receiver when it receives a new data item (that is, not a re-transmitted one). This timer is adjusted so that it wakes up the receiver (provided, of course, it has not received a new data item in the meantime) after a certain delay that is *guaranteed* to be such that the receiver can be certain that the sender has already aborted the protocol. Clearly this delay has to be greater than or equal to the quantity $(M + 1) \times dl$, since after that delay the sender must have given up as we have seen above. When the second timer wakes it up, the receiver aborts the protocol (from its own point of view). As can be seen, in case of problems, the two participants are indirectly synchronized by means of these timers.

1.4 Alternating Bit

As we have seen above, the sender may re-transmit the same data item several times. But, it may also transmit two (or more) successive data items, which might happen to have the *same value*. Of course, this is annoying, since the receiver may confuse a re-transmitted data item with a new one that is identical to its predecessor. In order to solve this problem, each data item is accompanied by a bit whose value is alternating from one item to the next. When the receiver receives two successive items accompanied by the same bit, it can thus be certain (is it?) that the latter is a re-transmission of the former.

1.5 Final Situation of the Protocol

At the end of the protocol execution, we might be in one of the following three situations:

1. either, the protocol has successfully been able to transfer the entire file from the sender to the receiver and the sender has indeed received the last acknowledgment from the receiver. In that case, both the sender and the receiver know that the protocol has ended successfully: the file has been entirely copied and both sites knows it.

2. or the protocol has successfully been able to transfer the entire file from the sender to the receiver but the sender has never received the last acknowledgment (in spite of successive re-transmissions, this message is definitely lost in the Acknowledgment Channel) so that the sender aborts the protocol whereas the receiver doesn't,

3. or else the protocol has aborted on both sites.

Notice that the fourth possibility, where the receiver would have aborted the protocol whereas the sender wouldn't, is not possible (is it true?).

1.6 A Pseudo-code Description of the BRP

In this section, we present a pseudo-code version of our protocol. The rôle of this description is to make a little more precise the completely informal presentation of the previous section. Each event of the protocol (that is, SND_snd, RCV_rcv, RCV_snd, and SND_rcv) and the two additional events corresponding to the timers (which we call, SND_timer and RCV_timer) are described in terms of an *enabling condition*, introduced as we have done in previous chapters by the keyword **when**, followed by an *action part*, introduced by the keyword **then**. The former contains the condition under which the event *may* be enabled, whereas the latter contains a description of what the event is supposed to do once it is enabled.

Event SND_snd**.** Our first event, SND_snd, is enabled by a condition expressing that this event is indeed waken up (we shall see below that this is done either by the event SND_rcv or by the event SND_timer). The action of SND_snd consists in acquiring the next data item from the sender's file, storing it on the Data Channel together with the corresponding alternating bit, starting the sender's timer, and finally activating the Data Channel (effectively sending the data and the bit). Here is on the left the pseudo-code of this event:

```
SND_snd
  when
    SND_snd is waken up
  then
    Acquire data from Sender's file;
    Store acquired data on Data Channel;
    Store Sender's bit on Data Channel;
    Start Sender's timer;
    Activate Data Channel;
  end
```

```
RCV_rcv
  when
    Data Channel interrupt occurs
  then
    Acquire Sender's bit from Data Channel;
    if Sender's bit = Receiver's bit then
      Acquire Data from Data Channel;
      Store data on Receiver's file;
      Modify Receiver's bit;
      if data is not the last one then
        Start Receiver's timer;
      end
    end
    Reset Data Channel Interrupt;
    Wake up event RCV_snd;
  end
```

Event RCV_rcv**.** The next event, RCV_rcv proposed above on the right, is enabled by the interrupt of the Data Channel on the receiver's site. The action consists first in testing whether the alternating bit sent by the sender is identical to the alternating bit previously stored by the receiver. If this is the case, then this means, by convention, that we have a new data item: this item is extracted from the Data Channel, it is subsequently stored on the receiver's file, the receiver's alternating bit is modified, and, finally, the receiver's timer is started if the received item is not the last one. In any case, the interrupt of the Data Channel is de-activated whereas event RCV_snd is waken up.

Event RCV_snd**.** The next event RCV_snd is enabled by event RCV_rcv as we have seen in the previous section. Its action simply consists in activating the Acknowledgment Channel. This event is shown below on the left hand side.

```
RCV_snd
  when
    RCV_snd is waken up
  then
    Activate Acknowledgment Channel;
  end
```

```
SND_rcv
  when
    Acknowledgment Channel interrupt occurs;
  then
    Remove Data from Sender's file;
    Reset retry counter;
    Modify Sender's bit;
    Reset Acknowledgment Channel interrupt;
    if Sender's file is not empty then
      Wake up event SND_snd;
    end
  end
```

Event SND_rcv**.** The next event, SND_rcv, is enabled by the interrupt of the Acknowledgment Channel on the sender's site. The action consists in removing the previously sent item from the sender's file (although that data item has already been sent, it was nevertheless kept in the file in case of a re-transmission; now it can be definitely removed since we have just received the acknowledgment telling us that the receiver has indeed received it). The sender's alternating bit can now be modified for the next data item, the event SND_snd is waken up, and, finally, the Acknowledgment Channel is de-activated.

Event SND_timer**.** The event SND_timer is enabled when the sender's timer reaches its specified delay. The action consists in testing whether the re-try counter has reached its maximum value, in which case the protocol is aborted (from the point of view of the sender). When this is not the case, then the re-try counter is incremented and, of course, the event SND_snd is waken up for a re-transmission.

```
SND_timer
  when
    sender's timer interrupt occurs
  then
    if retry counter is equal to M+1 then
      Abort protocol on Sender's site
    else
      Increment retry counter;
      Wake up event SND_snd;
    end
  end
```

```
RCV_timer
  when
    Receiver's timer interrupt occurs
  then
    Abort protocol on Receiver's site
  end
```

Event RCV_timer**.** The event RCV_timer is enabled when the receiver's timer reaches its specified delay. The action consists in aborting the protocol (from the point of view of the receiver).

Note The Sender knows that the file has been successfully sent and received when event SND_rcv observes that the file is empty (we suppose that the file is not empty at the beginning). It seems (but are we sure?) that event SND_timer cannot wake up event SND_snd while the file is empty.

Likewise, the Sender knows that the file has been entirely sent but that the last data has not been necessarily received. This happens when event SND_timer aborts the protocol while the Sender's file has just got one piece of data left.

The Receiver knows that the protocol ends successfully when it receives the last data: this is supposed to be indicated by a special information put on the last data itself.

1.7 About the Pseudo-code.

The definition of our protocol by means of this pseudo-code (or by means of any other similar descriptive notation) raises a number of questions. Are we sure that such a description is correct in the sense that it effectively corresponds to a *file transfer* protocol? Are we sure that the described protocol does terminate (no infinite loop, no deadlock)? What kind of properties should this protocol maintain?

It is our opinion that these questions cannot be answered on the basis of such an informal description only. Nevertheless, we believe that it is quite useful to have such a description at one's disposal, since it may act as a *goal* to our future protocol construction. In the sequel, and as said above, we shall formally construct our protocol starting from a mathematical specification of its main properties, and ending up in a formal description of its components, which we might then fruitfully *compare* to their informal pseudo-code counterparts.

The main drawback of such descriptions, which are often said to constitute the *specification* of these protocols is that they rather describe an informal *implementation*. This is the reason why it is so important to rewrite clearly our informal specification as a proper *requirement document*. This is what we intend to do in the next section.

2 Requirement Document

The requirement document which we propose now is *far less precise* than the previous informal explanations we have given. It is far less precise in that *it does not propose an implementation*. It essentially consists in explaining what kind of *belief* each site may have at the end of the protocol. We also make precise when such beliefs are indeed true. Here are our requirements for the Bounded Retransmission Protocol. We first make precise the overall purpose of the protocol:

The Bounded Retransmission Protocol is a file transfer protocol. Its goal is to totally or partially transfer a certain non-empty original sequential file from one site, the sender, to another, the receiver.	FUN-1

Then we explain what a "total transfer" means:

A "total transfer" means that the transmitted file is an exact copy of the original one.	FUN-2

We also explain what a "partial transfer" means:

A "partial transfer" means that the transmitted file is a prefix of the original one.	FUN-3

We describe now what both sites may *believe* at the end of the protocol:

Each site may end up in any of the two situations: either it believes that the protocol has terminated successfully , or it believes that the protocol has aborted before being successfully terminated.	FUN-4

We relate the beliefs of both the sender and the receiver:

When the sender believes that the protocol has terminated successfully then the receiver believes so too. Conversely, when the receiver believes that the protocol has aborted then the sender believes so too.	FUN-5

We explain that it is possible that these beliefs are not shared by both participants:

However, it is possible for the sender to believe that the protocol has aborted while the receiver believes that it has terminated successfully.	FUN-6

We explain finally that the belief of the receiver is always true:

When the receiver believes that the protocol has terminated successfully, this is because the original file has been entirely copied on the receiver's site. In other words, the receiver's belief is true.	FUN-7

When the receiver believes that the protocol has aborted, this is because the original file has not been copied entirely on the receiver's site. Again, the receiver's belief is true	FUN-8

3 Refinement Strategy

In this short section, we present our strategy for constructing the Bounded Re-transmission Protocol. This will be done by means of an initial model followed by six refinements.

– The initial model set up the scene by taking account of requirements FUN-4 stating the final situation of both participants of the protocol.

– In the first and second refinements refinement, we take care of the requirements FUN-5 and FUN-6 stating some relationship between the status of the two participants.

– In the third refinement, we introduce the transmitted file. It takes account of requirement FUN-1 to FUN-3. In this refinement, the receiver only enters into the scene.

– In the fourth refinement we introduce the sender which sends messages to the receiver and vice-versa.

– In the fifth refinement, we introduce the unreliability of the channels.

– In the last refinement, we optimize the information transmitted between the sender and the receiver.

4 Initial Model

Our initial model contains a very partial specification of the Bounded Re-transmission Protocol. It deals with requirements FUN-4:

Each site may end up in any of the two situations: either it believes that the protocol has terminated successfully , or it believes that the protocol has aborted before being successfully terminated.	FUN-4

4.1 The State

In this initial very abstract model, we introduce the concept of status. For this, we define a carrier set named $STATUS$. It is made of three distinct elements: $working$, $success$, and $failure$ as shown below:

sets: $STATUS$

constants: $working$, $success$, $failure$

axm1_1: $STATUS = \{working, success, failure\}$

axm0_2: $working \neq success$

axm0_3: $working \neq failure$

axm0_4: $success \neq failure$

There are two variables s_st and r_st defining the status of the two participants:

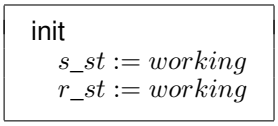
variables: s_st, r_st

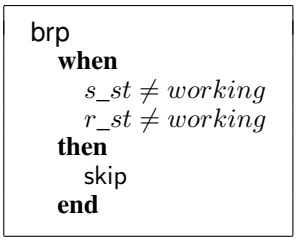
inv0_1: $s\_st \in STATUS$

inv0_2: $r\_st \in STATUS$

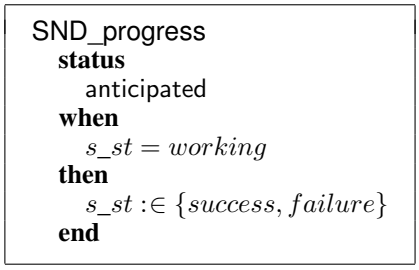
4.2 The Events

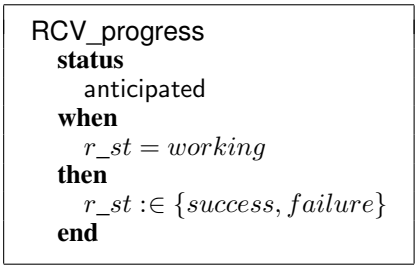
Initially, the participants are $working$. We have then an *observer* event named brp, which is fired when both participants are not working any more.





In what follows, we use the technique of *anticipated* events which was introduced and motivated in section 7 of chapter 4. We have thus two *anticipated* events claiming to have participants being eventually either in status $success$ or $failure$.





5 First and Second Refinements

These refinements take account of requirement FUN-5

When the sender believes that the protocol has terminated successfully then the receiver believes so too. Conversely, when the receiver believes that the protocol has aborted then the sender believes so too.	FUN-5

and of requirement FUN-6.

However, it is possible for the sender to believe that the protocol has aborted while the receiver believes that it has terminated successfully.	FUN-6

Finally, it makes more precise what is meant by the previous *anticipated* event.

5.1 The State

Invariant **inv1_1** below formalises requirement FUN-4. As it is not an equivalence it take accounts indirectly of requirement FUN-6.

inv1_1: $s\_st = success \Rightarrow r\_st = success$

5.2 Events of First Refinement

We split now events progress in success and failure events. Notice that events SND_success (in this section) and RCV_failure (in the next section) are both "cheating" as they contain the status of the other participant in their guards. We prove that these events are indeed convergent: it is done in two separate refinements.

SND_success
refines
SND_progress
status
convergent
when
$s\_st = working$
$r\_st = success$
then
$s\_st := success$
end

SND_failure
refines
SND_progress
status
convergent
when
$s\_st = working$
then
$s\_st := failure$
end

variant1: $\{success, failure\} \setminus \{s\_st\}$

5.3 Events of Second Refinement

RCV_success
refines
RCV_progress
status
convergent
when
$r\_st = working$
then
$r\_st := success$
end

RCV_failure
refines
RCV_progress
status
convergent
when
$r\_st = working$
$s\_st = failure$
then
$r\_st := failure$
end

variant2: $\{success, failure\} \setminus \{r\_st\}$

6 Third Refinement

In this refinement, we consider requirements FUN-1 to FUN-3 concerned with the transfer of the file. We also take account of requirement FUN-7 and FUN-8 expressing that the receiver belief is true.

6.1 The State

First, we extend our context by defining the sequential file f to be transmitted from the sender to the receiver.

sets: D

constants: n, f

axm0_1: $0 < n$

axm0_2: $f \in 1 \mathinner{..} n \rightarrow D$

The transmitted file is denoted by a variable g of length r. Invariant **inv3_2** formalises that the transmitted file is always a prefix of the original file. Invariant **inv3_3** formalises that the receiver succeeds exactly when the file has been transmitted entirely.

variables: r, g

inv3_1: $r \in 0 \mathinner{..} n$

inv3_2: $g = 1 \mathinner{..} r \lhd f$

inv3_3: $r\_st = success \Leftrightarrow r = n$

6.2 The Events

New Event RCV_rcv_current_data and refined event RCV_success both cheat as they contain direct references to information belonging to the sender, namely $f(r+1)$ and n. Event init is not shown here: it sets r to 0.

RCV_rcv_current_data
status
convergent
when
$r\_st = working$
$r + 1 < n$
then
$r := r + 1$
$g := g \cup \{r + 1 \mapsto f(r + 1)\}$
end

RCV_success
when
$r\_st = working$
$r + 1 = n$
then
$r\_st := success$
$r := r + 1$
$g := g \cup \{r + 1 \mapsto f(n)\}$
end

variant3: $n - r$

6.3 Synchronization of the Events

In this refinement, the events are synchronized according to the diagram of Fig. 2. In this figure, the new events are written in *italic* and the dashed line corresponds to the only synchronization we had in the abstraction.

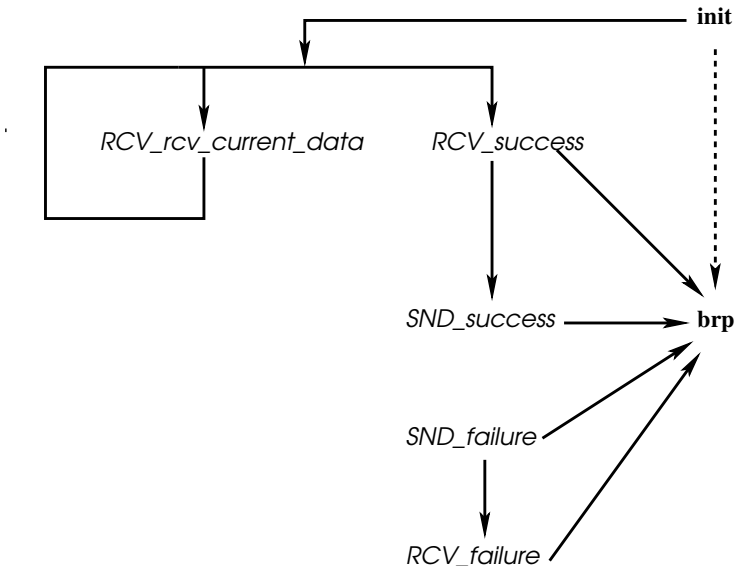


Fig. 2. Synchronisation of the Events

7 Fourth Refinement

In this refinement, the sender will enter into the scene by cooperating with the receiver in order to transmit the file. In fact, the receiver will not access any more directly the original file f as was the case in the previous refinement, this will be done by the sender who then sends the corresponding data to the receiver through the, so called, *data channel*. We then introduce this data channel and also the symmetric *acknowledgment channel*. Such channels are situated between the two sites. Notice that we do not introduce yet the fact that these channels are unreliable: this will be done in the next refinement only.

7.1 The State

The state is first enlarged with an activation bit, w, to be used by the sender. This variable is boolean as indicated implicitly in invariants **inv2_3**. When w is equal to TRUE, it means that the sender event sending information to the receiver can be activated.

variables: $\ldots$
w
s
d

inv4_1: $s \in 0 \,..\, n-1$

inv4_2: $r \in s \,..\, s+1$

inv4_3: $w = \text{FALSE} \Rightarrow d = f(s+1)$

The state is also enlarged with a sender pointer s which is such that $s+1$ points the next item, $f(s+1)$, of the original file f to be transmitted to the receiver. It is defined by invariant **inv4_1**. Also notice the very important property relating pointer s to the size r of the transmitted file: r is either equal to s or to $s+1$ as indicated by invariant **inv4_2**.

The state is further enlarged with the data container d, which is part of the data channel and which contains the next item to be transmitted. Its main property is defined in invariant **inv4_3**, which states that d is equal to $f(s+1)$ when the data channel is active, that is when $w = \text{FALSE}$.

7.2 The Events

Events brp, SND_failure, and RCV_failure are not modified in this refinement. The initialization event is extended in a straightforward fashion as indicated below. The activation bit w is set to TRUE at the beginning so that the only two events which can be fired are the ones described next.

init
$r := 0$
$g := \varnothing$
$r\_st := working$
$s\_st := working$
$w := \text{TRUE}$
$s := 0$
$d :\in D$

The next event SND_snd_ data is new. It corresponds to the main action of the sender, namely to prepare the information to be sent through the data channel. What is send through this channel are the data d, and the sender pointer s:

SND_snd_data
when
$s\_st = working$
$w = \text{TRUE}$
then
$d := f(s+1)$
$w := \text{FALSE}$
end

The next two events correspond to the receiver receiving information on the data channel. As can be seen, the receiver checks that the received pointer s from the sender is equal to its own pointer r. The first event, RCV_rcv_current_data, corresponds to the receiver receiving an information which is not the last one ($r+1 < n$). The second one corresponds to the receiver receiving the last item ($r+1 = n$): in this case, the receiver succeeds:

RCV_rcv_current_data
when
$r\_st = working$
$w = \text{FALSE}$
$r = s$
$r + 1 < n$
then
$r := r + 1$
$g := g \cup \{r + 1 \mapsto d\}$
end

RCV_success
when
$r\_st = working$
$w = \text{FALSE}$
$r = s$
$r + 1 = n$
then
$r\_st := success$
$r := r + 1$
$g := g \cup \{r + 1 \mapsto d\}$
end

Notice that the receiver is still "cheating" as it is able (in the guards above) to check the value of its pointer r against the constant size n of the original file, which is in the Sender's site. This anomaly will be corrected in the next refinement.

The next two events correspond to the sender receiving the acknowledgment from the receiver. The first one, SND_rcv_current_ack, is a new event. When the sender receives the last acknowledgment (when $s + 1 = n$ in event SND_success), the sender succeeds, otherwise (when $s + 1 < n$ in event SND_rcv_current_ack) it increments its pointer s and activates the events SND_snd_data by setting the activation bit w to TRUE:

SND_rcv_current_ack
when
$s\_st = working$
$w = \text{FALSE}$
$s + 1 < n$
$r = s + 1$
then
$w := \text{TRUE}$
$s := s + 1$
end

SND_success
when
$s\_st = working$
$w = \text{FALSE}$
$s + 1 = n$
$r = s + 1$
then
$s\_st := success$
end

We finally introduce an event that modifies the activation pointer w. This event will receive a full explanation in the next refinement:

SND_time_out_current
when
$s\_st = working$
$w = \text{FALSE}$
then
$w := \text{TRUE}$
end

7.3 Synchronization of the Events

In this refinements, the events are synchronized according to the diagram of Fig. 3 where the new events are written in *italic*. These new events are inserted in the previous synchronization diagram of Fig. 2. Events SND_failure, RCV_failure, and SND_time_out_current are presently "spontaneous". They will receive more explanations in the next refinement.

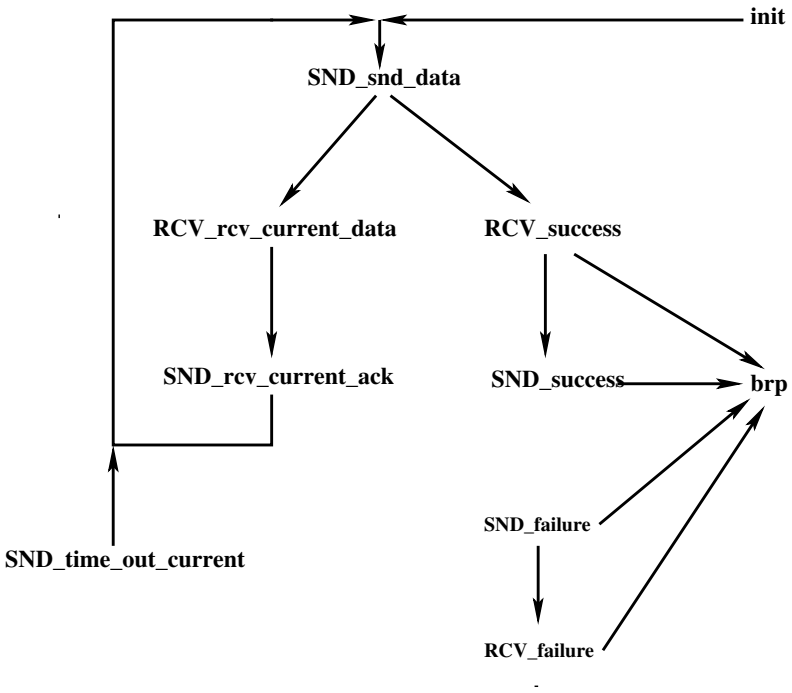


Fig. 3. Synchronisation of the Events in the Fourth Refinement

8 Fifth Refinement

8.1 The State

In this refinement, we introduce the unreliability of the channels.This is done by first adding three activation bits: db, ab and v. At most one of these bits, together with w already introduced in previous refinements, is equal to TRUE at a time: this is expressed in invariants **inv5_1** to **inv5_6**. The activation bits are used as indicated in Fig. 4.

variables:	$\ldots$
	db
	ab
	v

inv5_1:	$w = \text{TRUE} \Rightarrow db = \text{FALSE}$
inv5_2:	$w = \text{TRUE} \Rightarrow ab = \text{FALSE}$
inv5_3:	$w = \text{TRUE} \Rightarrow v = \text{FALSE}$
inv5_4:	$db = \text{TRUE} \Rightarrow ab = \text{FALSE}$
inv5_5:	$db = \text{TRUE} \Rightarrow v = \text{FALSE}$
inv5_6:	$ab = \text{TRUE} \Rightarrow v = \text{FALSE}$

We introduce an additional boolean variable, l, which denotes the last item indicator. It is send by the sender to the receiver (together with d and s). When equal to TRUE, this bit indicates that the sent item is the last one (invariants **inv5_7** and **inv5_8**).

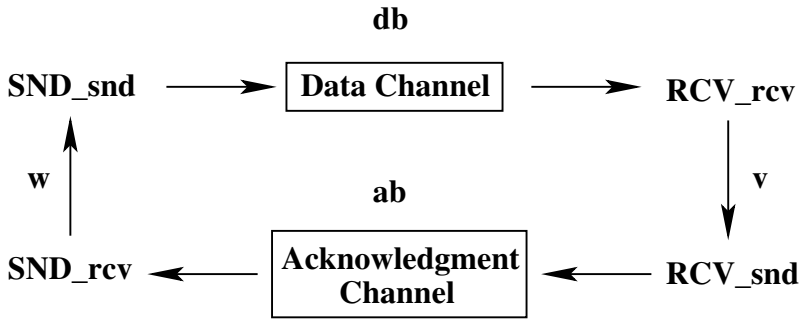


Fig. 4. The Activation Bits

variables: $\ldots$ l

inv5_7: $db = \text{TRUE} \ \wedge \ r = s \ \wedge \ l = \text{FALSE} \ \Rightarrow \ r + 1 < n$

inv5_8: $db = \text{TRUE} \ \wedge \ r = s \ \wedge \ l = \text{TRUE} \ \Rightarrow \ r + 1 = n$

Finally, we introduce a constant MAX and a variable c. Constant MAX denotes the maximum number of re-tries and variable c denotes the current number of re-tries. In invariant **inv3_10**, it is explained that when c exceeds MAX then the sender fails.

constants: $\ldots$ MAX

axm3_1: $MAX \in \mathbb{N}$

variables: $\ldots$ c

inv3_9: $c \in 0 \,..\, MAX + 1$

inv3_10: $c = MAX + 1 \Leftrightarrow s\_st = failure$

8.2 The Events

The initial event is extended in a straightforward fashion. Event brp is not modified in this refinement.

init
$r := 0$
$g := \varnothing$
$r\_st := working$
$s\_st := working$
$s := 0$
$d :\in D$
$w := \text{TRUE}$
$db := \text{FALSE}$
$ab := \text{FALSE}$
$v := \text{FALSE}$
$l := \text{FALSE}$
$c := 0$

brp
when
$r \neq working$
$s \neq working$
then
skip
end

The following events are modified as indicated by the underlined actions. We split now abstract event SND_snd_data into two events according to the sending of the last data or not. The activation bit, db, of the data channel is set to TRUE.

SND_snd_current_data
refines
SND_snd_data
when
$s\_st = working$
$w = \text{TRUE}$
$\underline{s + 1 < n}$
then
$d := f(s + 1)$
$w := \text{FALSE}$
$\underline{db := \text{TRUE}}$
$\underline{l := \text{FALSE}}$
end

SND_snd_last_data
refines
SND_snd_data
when
$s\_st = working$
$w = \text{TRUE}$
$\underline{s + 1 = n}$
then
$d := f(s + 1)$
$w := \text{FALSE}$
$\underline{db := \text{TRUE}}$
$\underline{l := \text{TRUE}}$
end

In the next two receiver events, the abstract "cheating" guards $r+1 < n$ and $r+1 = n$ have disappeared. They have been replaced by guards $l = \text{FALSE}$ and $l = \text{TRUE}$ respectively. Invariants **inv3_11** and **inv3_12** defined below ensure guard strengthening. The receiver activation bit v is set to TRUE.

RCV_rcv_current_data
when
$r\_st = working$
$\underline{db = \text{TRUE}}$
$r = s$
$\underline{l = \text{FALSE}}$
then
$r := r + 1$
$h := h \cup \{r + 1 \mapsto d\}$
$\underline{db := \text{FALSE}}$
$\underline{v := \text{TRUE}}$
end

RCV_success
when
$r\_st = working$
$\underline{db = \text{TRUE}}$
$r = s$
$\underline{l = \text{TRUE}}$
then
$r\_st := success$
$r := r + 1$
$h := h \cup \{r + 1 \mapsto d\}$
$\underline{db := \text{FALSE}}$
$\underline{v := \text{TRUE}}$
end

The next two events are new. Event RCV_rcv_retry correspond to the receiver receiving a re-try. The receiver detects this by the fact that its own pointer r is different from the one, s, it receives from the sender. The activation bit v is set to TRUE. The second event, RCV_snd_ack, is activated when v is equal to TRUE. It sends the acknowledgment to the sender by setting the activation bit ab of the Acknowledgment channel to TRUE. Notice that that no information is sent.

RCV_rcv_retry
when
$db = \text{TRUE}$
$r \neq s$
then
$db := \text{FALSE}$
$v := \text{TRUE}$
end

RCV_snd_ack
when
$v = \text{TRUE}$
then
$v := \text{FALSE}$
$ab := \text{TRUE}$
end

In the next two sender events, the abstract guard $r = s + 1$ has disappeared. It has been replaced by the guard $ab = \text{TRUE}$. In order to ensure guard strengthening, we have to add the following invariants:

inv3_11: $ab = \text{TRUE} \Rightarrow r = s + 1$

inv3_12: $v = \text{TRUE} \Rightarrow r = s + 1$

The second invariant helps proving the first one in event RCV_snd_ack.

SND_rcv_current_ack
when
$s\_st = working$
$\underline{ab = \text{TRUE}}$
$s + 1 < n$
then
$w := \text{TRUE}$
$s := s + 1$
$\underline{c := 0}$
$\underline{ab := \text{FALSE}}$
end

SND_success
when
$s\_st = working$
$\underline{ab = \text{TRUE}}$
$s + 1 = n$
then
$s\_st := success$
$\underline{c := 0}$
$\underline{ab := \text{FALSE}}$
end

The next two new events corresponds to the daemons breaking the channels. It results in activation bits w, db, v, and ab being all equal to FALSE. Notice that these events can occur asynchronously when the corresponding channels are active.

DMN_data_channel
when
$db = \text{TRUE}$
then
$db = \text{FALSE}$
end

DMN_ack_channel
when
$ab = \text{TRUE}$
then
$ab = \text{FALSE}$
end

The three next events corresponds to the timers. The first two are the sender timer. The first one occurs when the retransmission has not yet reach the maximum MAX, whereas the second one corresponds to this maximum: in this case, the sender fails. The last one corresponds to the receiver failure. This occurs when the sender has already failed according to invariant **inv3_10**. As can be seen, the time slot given to the receiver timer implicitly assume that this event can only occur when the sender has failed.

SND_time_out_current
when
$s\_st = working$
$w = \text{FALSE}$
$\underline{ab = \text{FALSE}}$
$\underline{db = \text{FALSE}}$
$\underline{v = \text{FALSE}}$
$\underline{c < MAX}$
then
$\underline{w := \text{TRUE}}$
$\underline{c := c + 1}$
end

SND_failure
when
$s\_st = working$
$w = \text{FALSE}$
$\underline{ab = \text{FALSE}}$
$\underline{db = \text{FALSE}}$
$\underline{v = \text{FALSE}}$
$\underline{c = MAX}$
then
$s\_st := failure$
$\underline{c := c + 1}$
end

RCV_failure
when
$r\_st = working$
$c = MAX + 1$
then
$r\_st := failure$
end

8.3 Synchronization of the Events

The last synchronization of the events is shown in the diagram of Fig. 5.

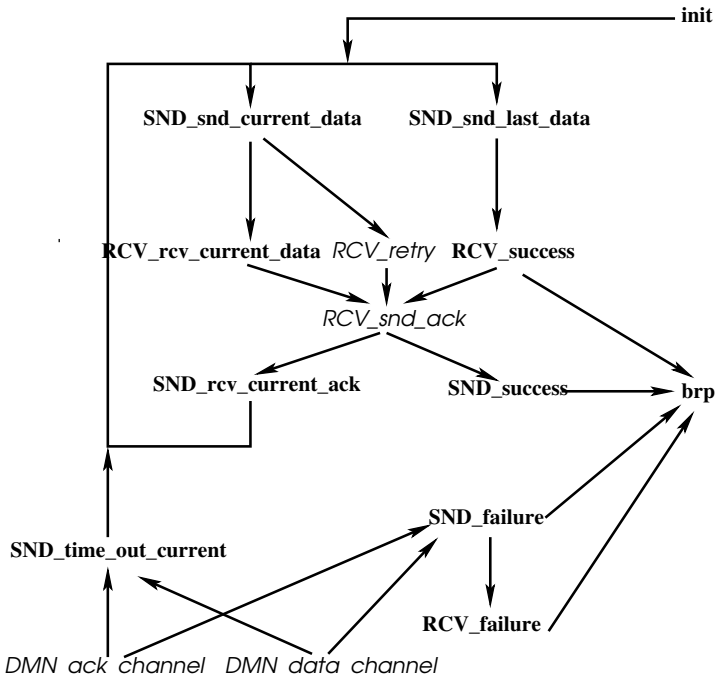


Fig. 5. Synchronisation of the Events in the Third Refinement

9 Sixth Refinement

The sixth refinement consists in sending the parity of pointer s from the sender to the receiver, and the parity of pointer r in the other direction. The definition of this refinement is left to the reader. The technique to be used is the one used in section 6 of chapter 4.

References

1. J.F. Groote and J.C. Van de Pol *A bounded retransmission protocol for large data packets - a case study in computer checked algebraic verification* Algebraic Methodology and Software Technology, 5th International Conference AMAST '96, Munich. Lecture Notes in Computer Science 1101.