Simple Protocol

Abstract
This is a small toy example describing a simple protocol by which a sender can transfer a number of packets to a receiver. The communication medium may loose packets and packets may overtake each other. Hence, it may be necessary to retransmit packets and to ignore doublets and packets that are out of order.

The example illustrates how the results of a lengthy simulation can be recorded for later inspection and analysis. One way is to add "reporting places" and another is to use an output file.

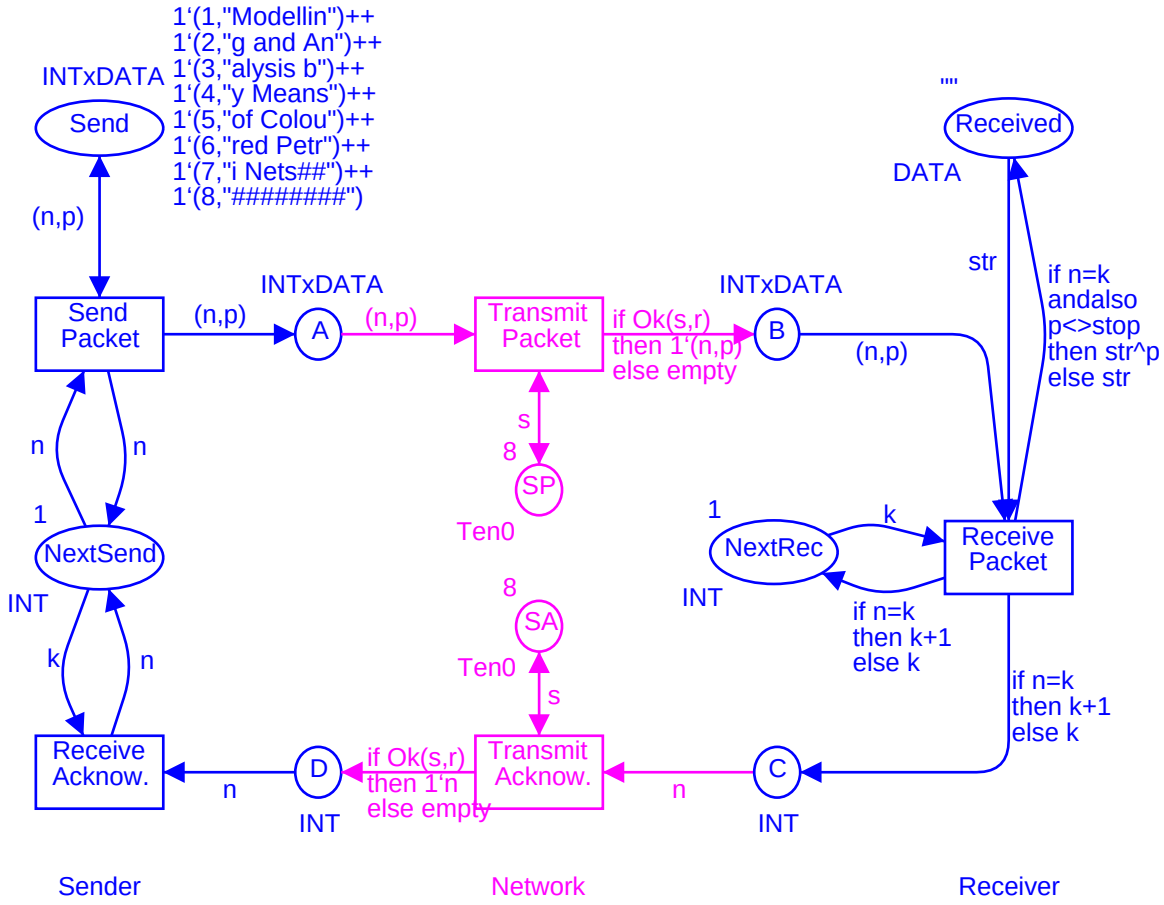
The example is a modified version of a timed CP-net presented in Sect. 5.5 of Vol. 2 of the CPN book.

Developed and Maintained by:
Kurt Jensen, Aarhus University, Denmark (kjensen@daimi.au.dk).

CPN Model

This example describes a simple protocol where a sequence of packets is sent from one site to another via a network where packets may be delayed or lost. We do not claim that the described protocol is optimal (it is not). However, the protocol is interesting enough to deserve a closer investigation, and it is also complex enough for such an investigation to be valuable.

The CPN model of the protocol system is shown below. It consists of three parts. The *Sender* part has two transitions which can *Send Packets* and




```
color INT = int;
color DATA = string;
color INTxDATA= product INT*DATA;
var n,k: INT;
var p, str: DATA;
val stop = ”########”;
color Ten0 = int with 0..10;
color Ten1 = int with 1..10;
var s: Ten0; var r: Ten1;
fun Ok(s:Ten0, r:Ten1) = (r<=s);
```

Receive Acknowledgments. The *Network* part has two transitions which can *Transmit Packets* and *Transmit Acknowledgments*. Finally, the *Receiver* part has a single transition which can *Receive Packets* (and send acknowledgments). The interface between the *Sender* and the *Network* consists of the places *A* and *D*, while the interface between the *Network* and the *Receiver* consists of the places *B* and *C*.

The packets to be sent are positioned at the place *Send* (in the upper left corner). Each token on this place contains a packet number and the data contents of the packet (represented as a text string). The place *Next Send* contains the number of the next packet to be sent. Initially this number is 1, and it is updated each time an acknowledgment is received.

The content of the received message is kept at the place *Received* (in the upper right corner). This place contains a single token with a text string which is the concatenation of the text strings contained in the received packets (ignoring the contents of duplicates and packets received out of order). Initially the text string at *Received* is empty, i.e., "". At the end of the transmission we expect *Received* to contain the text string "Modelling and Analysis by Means of Coloured Petri Nets". The place *Next Rec* contains the number of the next packet to be received. Initially this number is 1, and it is updated each time a packet is successfully received.

We do not model how the *Sender* splits a message into a sequence of packets or how the *Receiver* reassembles the packets into a message. Neither do we model how the tokens at *Send* and *Received* are removed at the end of the transmission or how the packet numbers in *Next Send* and *Next Rec* are reset to 1. Now let us take a closer look at the five different transitions in the protocol system.

Send Packet sends a packet to the *Network* by creating a copy of the packet on place *A*. The number in *Next Send* specifies which packet to send. It should be noted that the packet is not removed from *Send*. Neither is the counter at *Next Send* increased. The reason is that the packet may be lost and hence need to be retransmitted. Our protocol is pessimistic, in the sense that it continues to repeat the same packet – until it gets an acknowledgment telling that the packet has been successfully received. *Transmit Packet* transmits a packet from the *Sender* site of the *Network* to the *Receiver* site by moving the corresponding token from *A* to *B*. The boolean expression *Ok(s,r)* determines whether the packet is successfully transmitted or lost. The variable *r* will be bound to an arbitrary value in its colour set (i.e., to any integer between 1 and 10). CPN Tools makes a fair choice between the 10 values. The *Ok* function returns true if the value of *r* is less than or equal to the value of *s*. This means that the probability of successful transmission is determined by the token at place *SP*. We have given SP a token with value 8. Hence we have 80 % chance for successful transmission. However, it is easy to modify the success rate, simply by changing the token value at *SP*.

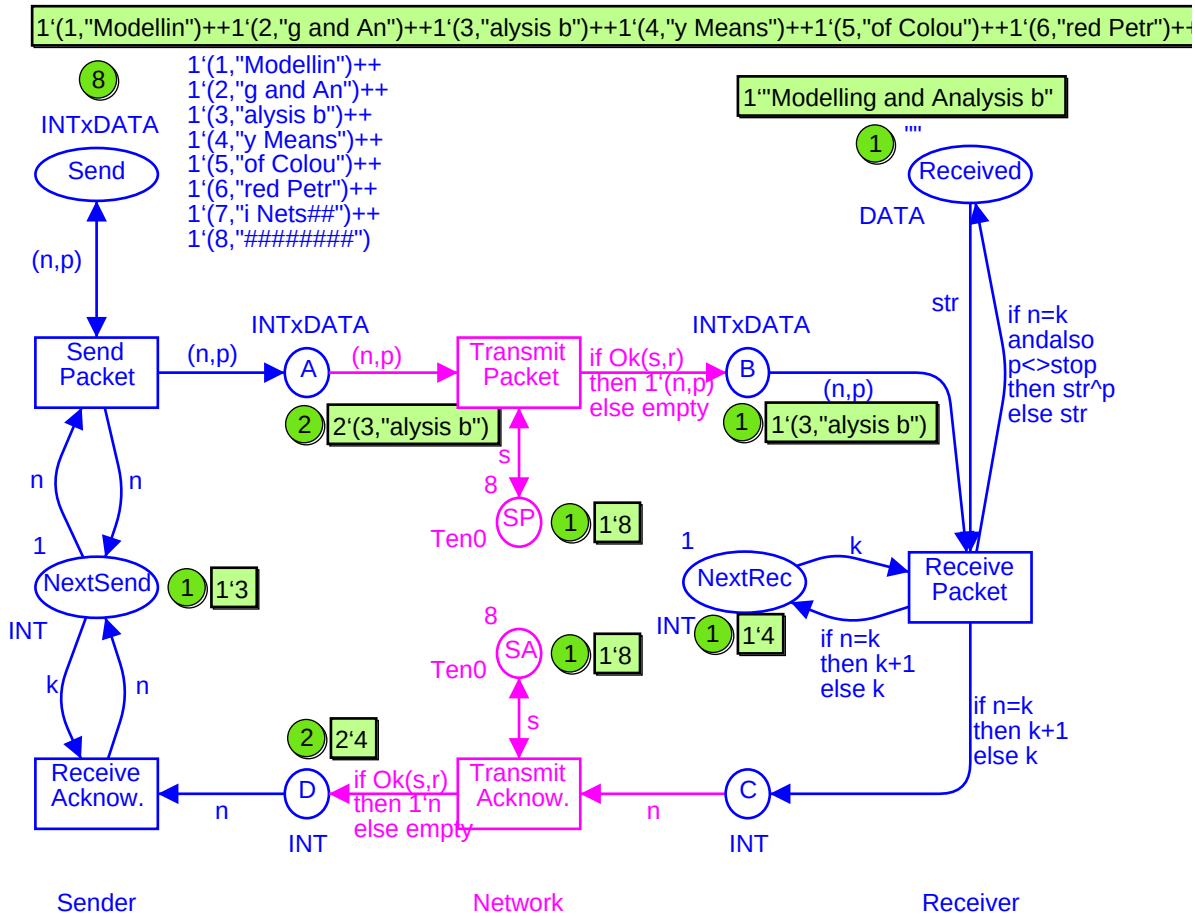
Receive Packet receives a packet and checks whether the packet number *n* is identical to the number *k* in *Next Rec*. When the two numbers match, the number in *Next Rec* is increased by 1 and the text string in the packet is concatenated to the text string in *Received* – unless it is stop = "########", which by convention indicates end-of-message. Otherwise, the packet is ignored and the number in *Next Rec* is left unchanged. In both cases an acknowledgment is sent containing the number of the next packet which the *Sender* should send.

Transmit Acknowledgment transmits an acknowledgment from the *Receiver* site of the *Network* to the *Sender* site by moving the corresponding token from *C* to *D*. The transition works in a similar way as *Transmit Packet*. This means that the acknowledgment may be lost, with a probability determined by the token at place *SA*.

Receive Acknowledgment receives an acknowledgment and updates the number in *Next Send* by replacing the old value with the one contained in the acknowledgment.

After a number of steps the CP-net may have reached the intermediate marking shown below. From the left-hand part of the net, we see that the sender is sending packet number three. We also see that three copies of this packet are present at places *A* and *B*. From the right-hand part of the net we see that the

Intermediate Marking



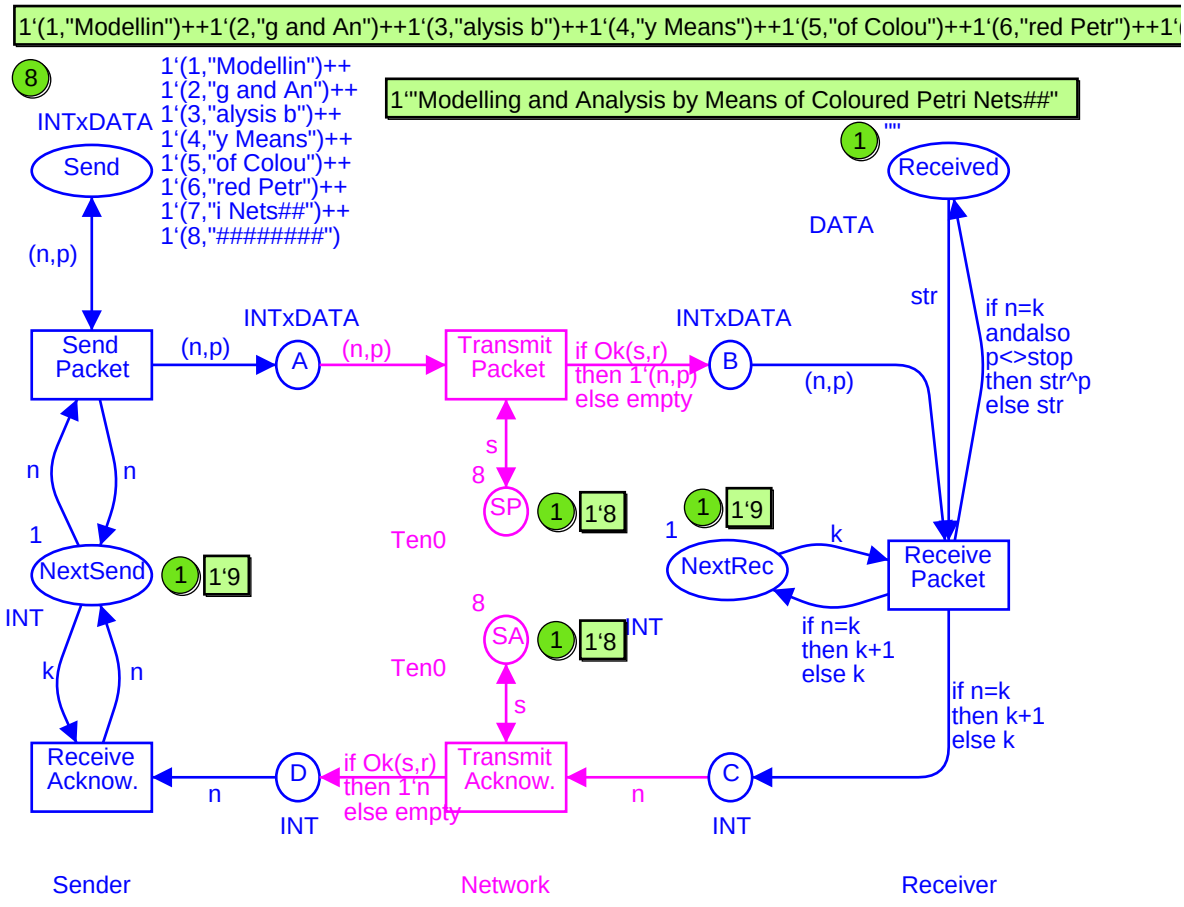
string "Modelling and Analysis b" has been *Received*. This is the contents of the first three packets and the receiver is now waiting for packet number four. Hence the packets on *A* and *B* will be ignored when they reach the receiver. We can also see that two acknowledgments are present at place *D*. When *Receive Acknowledgment* occurs, *Next Send* will be updated to four, and then the sender will start sending packet number four.

When the last packet (with "########") is successfully received by the receiver, *Next Rec* gets the value nine (one larger than the number of packets). This value will (via an acknowledgment) be communicated to the sender, *Next Send* will be updated to nine, and the sending will stop – because no packet with this number exists. After a few more steps, where the places *A*, *B*, *C* and *D* are cleared for packets/acknowledgments, the CP-net will reach a dead final marking, which looks as shown below.

Even though this protocol is rather simple, it is not that easy to see that it actually works correctly. What happens, for instance, if the "last" acknowledgment gets lost? By making a number of simulations – interactive and automatic – the user can greatly increase his confidence in the protocol. He may also make a proof of correctness by using the occurrence graph tool. For more information about this, see the example: "Simple Protocol for Occurrence Graph".

It is often convenient to be able to record the things that happen during a

Final Marking

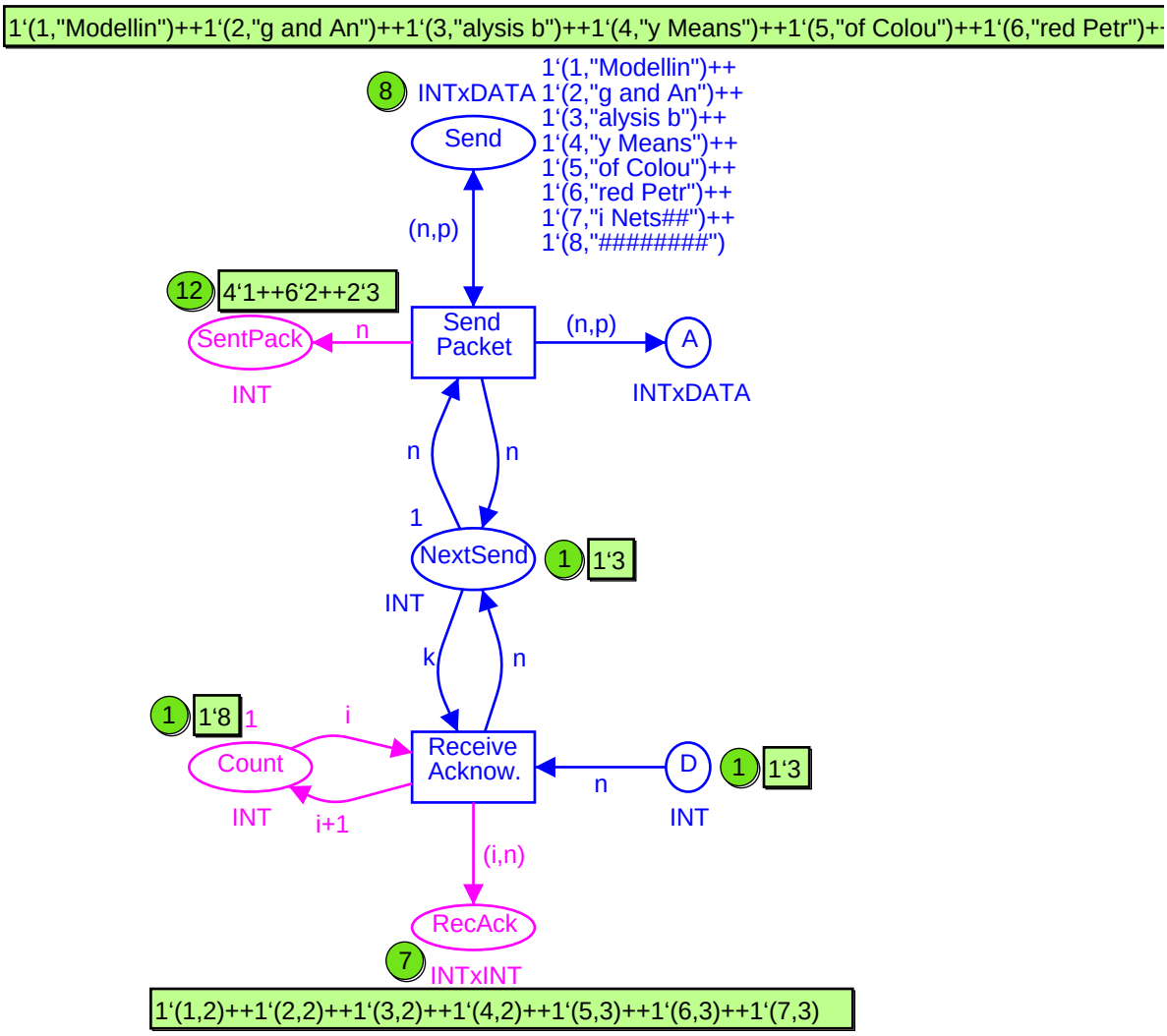


simulation. This is in particular the case when we perform a fast-forward simulation where we have no chance to observe the individual steps. This can be done by adding a number of "reporting places", i.e., places that gather historical information about the simulation run – without influencing the simulation. The use of reporting places is illustrated below. To see the *reporting places* in CPN Tools, you have to use the CPN model called "SimpleProtocol_RP". First we add a few extra declarations:

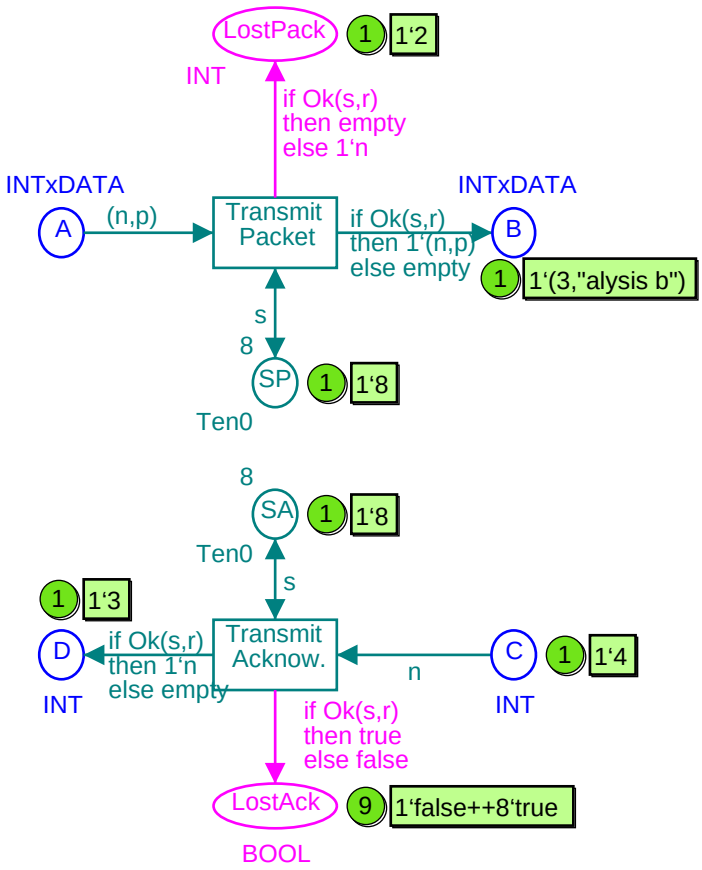
```
color BOOL = bool;
color INTxINT = product INT*INT;
color PackSeq = list INTxDATA;
var i: INT;
var plist: PackSeq;
```

Then we add one or two extra places for each of the five transitions. The place *SentPack* tells us how many times the individual packets have been sent. In our example, packet number one has been sent four times, packet number two six times and packet number three twice). The place *RecAck* tells us which acknowledgments the sender has received. Each acknowledgment is recorded as a pair, where the first element is a sequence number while the second is the contents of the acknowledgment. The sequence number is obtained from the place *Count*. In our example, we have first received four acknowledgments with value two and then three acknowledgments with value three.

Reporting Facilities for the Sender

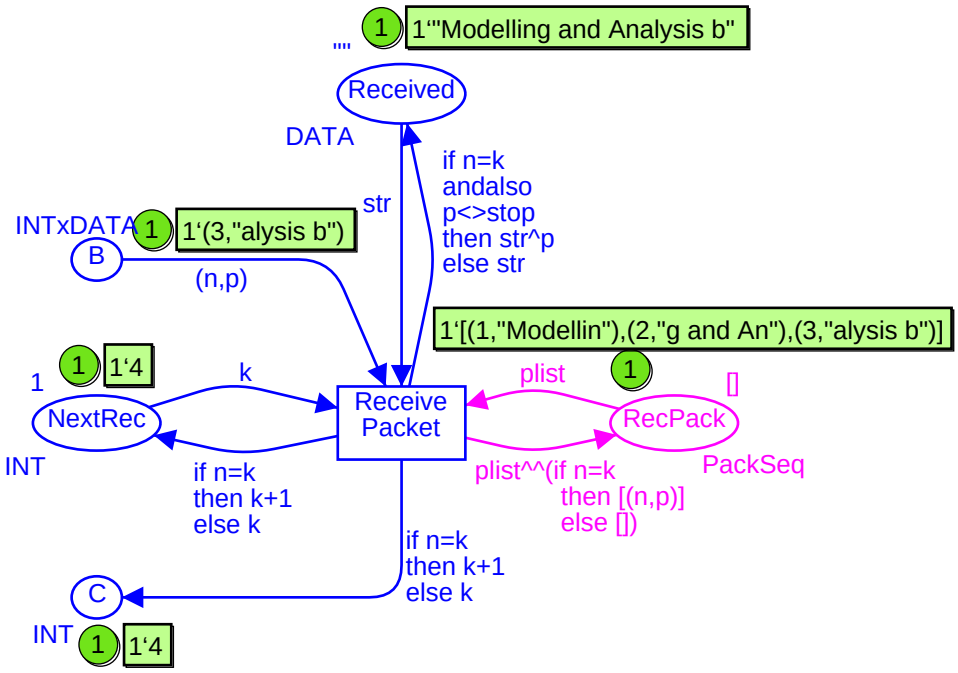


Reporting Facilities for the Network



The place *LostPack* tells us about the lost packets. In our example, we have only lost one copy of packet number two. The place *LostAck* tells us how many acknowledgments we have transmitted/lost. In our example, we have lost one acknowledgment and successfully transmitted eight.

Reporting Facilities for the Receiver



The place *RecPack* tells us about the successfully received packets (the ^^ operator at the arc from *ReceivePacket* to *RecPack* concatenates the two lists). In our example, we have received three packets. First we received packet number one with data "Modellin", then we received packet number two with data "g and An", and finally packet number three with data "alysis b".

A second way to record the results of a simulation is to use code segments to write selected results on an output file. The file can then later be read – by a human being or by another computer program (e.g., a spreadsheet program). To see the use of an output file (and an input file) in CPN Tools itself, you have to use the CPN model called "SimpleProtocol_IO". We first add a few extra declarations:

```
color INTxDATA = product  INT * DATA;
color E = with e;
globref packets = empty: INTxDATA ms;
globref outfile = TextIO.stdOut;
fun getPackets() = (!packets);
```

The predefined function *INTxDATA.output* writes the value of a specified expression to a specified file. The expression must be of type *INTxDATA* (if the expression is a *multi-set* over *INTxDATA*, you have to use the function *INTxDATA.output_ms*). Analogously, the predefined function

INTxDATA.input_ms reads a *multi-set* over *INTxDATA* from a specified file (if you want to read a *single valu*e of type *INTxDATA*, you have to use the function *INTxDATA.input*).

It should be noted that the variable called *packets* is a *global reference* variable, and its type is *INTxDATA ms*. This means that it will be bound to *multi-sets* over *INTxDATA* (instead of a *single value* of *INTxDATA*). The variable is initialized to the empty multi-set. The function *getPackets* simply returns the multi-set that *packets* refers to. The global reference variable *outfile* will be used to provide a handle to our output file. The reference variable is initialised to TextIO.stdOut (standard output), but this will later be overwritten. Next, we add a code segment to transition *Receive Packet:*

```
input (n,p,k);
action
if n=k then
  (if n=1 then
     outfile := TextIO.openOut("SimpleProtocol.SimRes")
   else ();
   INTxDATA.output(!outfile, (n,p));
   if p=stop then TextIO.closeOut(!outfile) else ())
else ();
```

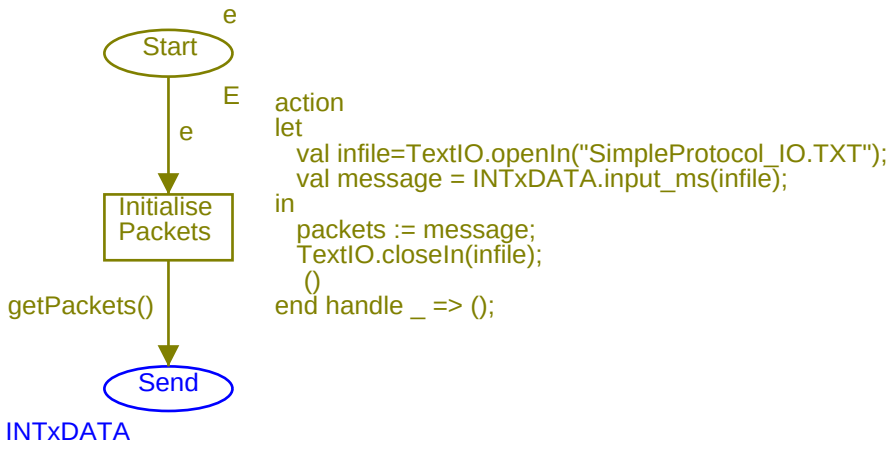
The first line specifies that the action part of the code segment is allowed to refer to the values of the variables n, p and k of the transition. The action part tests whether the received packet is a success ($n = k$) or a failure ($n \neq k$). For failures nothing is done, while successes imply the following operations on the output file:

- When the first packet is received ($n = 1$), the output file, called *SimpleProtocol.SimRes* is opened.
- For each successfully received packet, we use the function *INTxDATA.output* to write the value of *(n,p)* to the output file (a space is automatically appended after the value).
- When the last packet is received *(p = stop)*, the output file is closed.

After a simulation (with the usual eight packets) the output file will have the following contents:

```
(1, "Modellin") (2, "g and An") (3, "alysis b") (4,
"y Means ") (5, "of Colou") (6, "red Petr") (7, "i
Nets##") (8, "########")
```

Next let us show how an input file can be used to initialise a simulation. To do this we add an extra transition, called *Initialise Packets*. When a simulation starts, this transition reads a multi-set from a specified file and adds the tokens specified by the multi-set to place *Send* (which initially is empty).



Again we use a fixed file name (as we did for the output file). First, a multi-set over *INTxDATA* is read and bound to the global reference variable *packets*. Then the input file is closed. When the arc inscription *getPackets*() is evaluated, it will return the current value of *packets*, i.e. it will return the multi-set that was read in from the file. If the input file is not present or an error occurs while reading the file, an exception is raised. This exception is handled in the last line of the code segment (in this case the value of *packets* is unchanged, and the function *getPackets*() will return the empty multi-set).

The file "SimpleProtocol_IO.TXT" (provided together with this example) specifies a multi-set with the "usual" eight packets. It looks as follows:

```
%This file is used by to initialise
%the packets on Send

1`(1,"Modellin")++ % first packet
1`(2,"g and An")++ % second packet
1`(3,"alysis b")++ % third packet
1`(4,"y Means ")++ % fourth packet
1`(5,"of Colou")++ % fifth packet
1`(6,"red Petr")++ % sixth packet
1`(7,"i Nets##")++ % seventh packet
1`(8,"########")   % eighth packet
```

It is easy to modify the CPN model for the "Simple Protocol", e.g., to obtain a blast protocol – in which all the packets are sent immediately after each other – without waiting for acknowledgments for the individual packets. The reader is

encouraged to make one or more such modifications and to use the CPN Tools simulator to validate the new protocols. It is surprisingly "easy" to get a protocol which is "nearly correct" but not totally correct. One of the problems is to be sure that we always stop. Remember that both the last packet and the last acknowledgment may be lost.

In the present CP-net we have described that retransmissions *may* occur. However, we have not provided any details telling when and how often this will happen – the transition *Send Packet* is always enabled. As shown in "Timed Protocol" it is possible to augment the CP-net by specifying how long time the individual operations take and how long time the sender should wait before making a retransmission. With such a description it becomes possible to experiment with different waiting times to determine which one is the best – in the sense that it transmits the message fast without using the network too much (i.e. without making too many retransmissions).