# An Introduction to the Practical Use of Coloured Petri Nets

Kurt Jensen
Department of Computer Science, University of Aarhus
Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark

Phone: +45 89 42 32 34, Telefax: +45 89 42 32 55
E-mail: kjensen@daimi.aau.dk, WWW: http://www.daimi.aau.dk/~kjensen/

**Abstract:** The development of Coloured Petri Nets (CP-nets or CPN) has been driven by the desire to develop a modelling language – at the same time theoretically well-founded and versatile enough to be used in practice for systems of the size and complexity found in typical industrial projects. To achieve this, we have combined the strength of Petri nets with the strength of programming languages. Petri nets provide the primitives for describing synchronisation of concurrent processes, while programming languages provide the primitives for definition of data types and manipulation of their data values.

The paper focuses on the practical use of Coloured Petri Nets. It introduces the basic ideas behind the CPN language, and it illustrates how CPN models can be analysed by means of simulation, state spaces and condensed state spaces. The paper also describes how CP-nets can be extended with a time concept. In this way it is also possible to use CP-nets for performance evaluation, i.e., to evaluate the speed by which a system operates. Finally, we describe a set of computer tools that support the use of CP-nets. This tool set is used by more than three hundred organisations in forty different countries – including seventy-five commercial companies. It is available free of charge, also for commercial use.

The present paper does not contain any formal definitions. Instead all ideas and concepts are introduced by means of a number of small examples. Readers who want to consult the formal definitions can find these in [1], [2], [3], and [4]. The latter is a 3-volume text book providing a detailed description of CP-nets and their use. Volume 1 introduces the basic concepts and definitions. Volume 2 describes the different analysis methods. Volume 3 describes experiences from nineteen projects in which CP-nets and the CPN tools have been put to practical use. Most of the projects have been carried out in an industrial setting.

**Keywords:** High-level Petri Nets, Coloured Petri Nets, Practical Use, Modelling, Validation, Verification, State Spaces, Tool Support.

## Table of Contents

To appear in the course material from the Advanced Course on Petri Nets, Dagstuhl Germany, 1996. The course material will be published in Lecture Notes in Computer Science, Springer-Verlag.

# 1 Introduction to CP-nets

This section contains an informal introduction to CP-nets. This is done by means of an example that models a simple protocol, Fig. 1. The example is far too small to illustrate the typical practical use of CP-nets, but it is large enough to illustrate the basic concepts of the CPN modelling language and the basic ideas behind the analysis methods, such as simulation and state spaces. Throughout this paper we shall develop a number of protocol models and use these to illustrate different aspects of CP-nets. We do not claim that the described protocols are optimal (they are not). However, the protocols are interesting enough to deserve a closer investigation, and they are also complex enough for such an investigation to be non-trivial.

In contrast to most specification languages, Petri nets are state and action oriented at the same time – providing an explicit description of both the states and the actions. This means that the modeller can determine freely whether – at a given moment of time – he wants to concentrate on states or on actions.

The states of a CP-net are represented by means of **places** (which are drawn as ellipses or circles). In the protocol system there are ten different places. By convention we write the names of the places inside the ellipses. The names have no formal meaning – but they have large practical importance for the readability of a CP-net (just like the use of mnemonic names in traditional programming). A similar remark applies to the graphical appearance of the places, i.e., the line thickness, size, colour, font, position, etc.

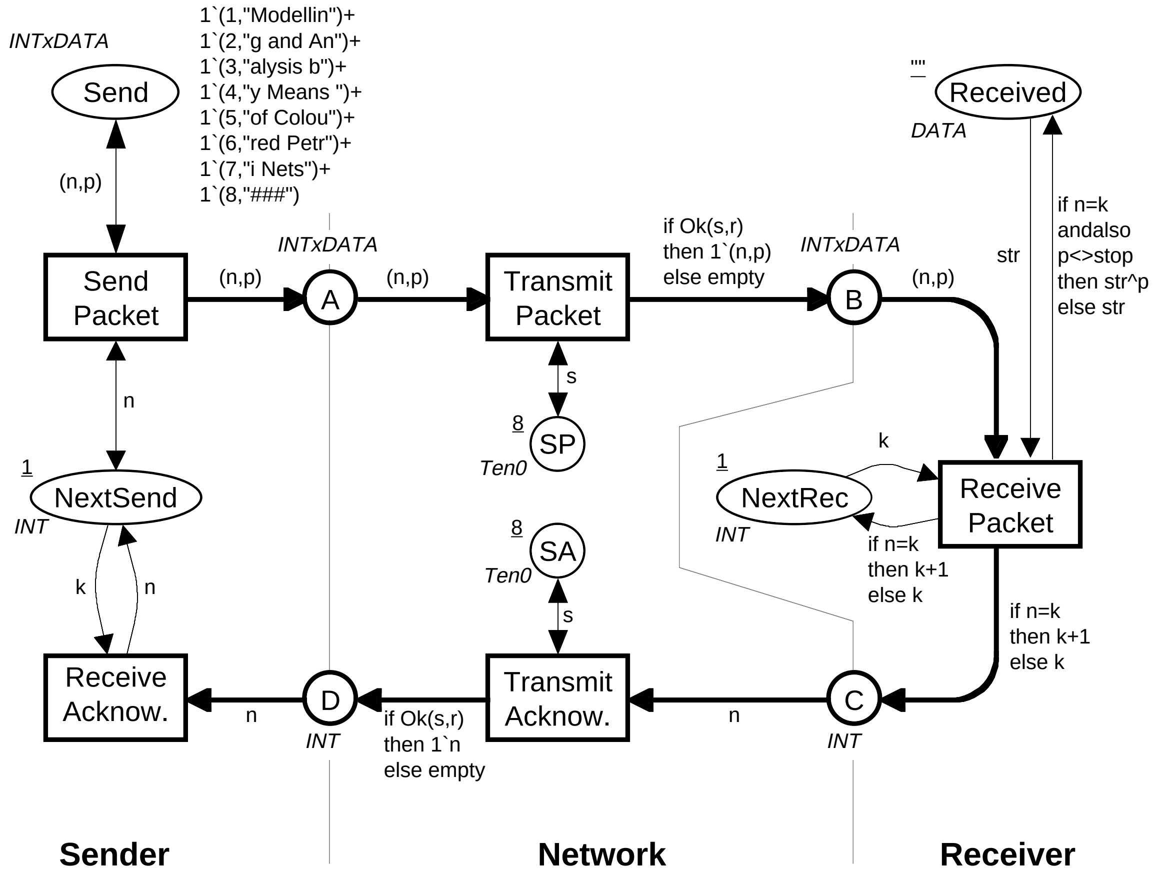


**Fig. 1.** CP-net describing a simple protocol

Each place has an associated **type** determining the kind of data that the place may contain (by convention the type name is written in italics, next to the place). In the protocol system we use four different types. Places *Send*, *A*, and *B* have the type *INTxDATA*. This type is the cartesian product of *INTegers* and *DATA*. The elements of the type represent packets to be transmitted over the *Network*. Each packet is a pair, where the first element is the packet number (of type *INT*), while the second element is the data contents of the packet, i.e., a text string (of type *DATA*).

During the execution of a CP-net each place will contain a varying number of **tokens**. Each of these tokens carries a data value that belongs to the type associated with the place. As an example, Fig. 1 shows that place *Send* starts with the following eight **token values**, which each represents a packet to be transmitted over the *Network*:

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

In Fig. 1 there is a *1`* in front of each token value. This tells us that there is exactly one token that carries the value. In general, several tokens may have the same token value, and then we have a **multi-set** of token values, such as:

1`(2,"g and An") + 2`(3,"alysis b") + 1`(5,"of Colou")

in which we have one token with value *(2,"g and An")*, two tokens with value *(3,"alysis b")*, and one token with value *(5,"of Colou")*. A multi-set is similar to a set, except that there may be several appearances of the same element. If we add the element *(3,"alysis b")* to the set:

{(2,"g and An"), (3,"alysis b"), (5,"of Colou")}

nothing happens, because the element already belongs to the set. However, if we add the element *(3,"alysis b")* to the multi-set:

1`(2,"g and An") + 1`(3,"alysis b") + 1`(5,"of Colou")

we get a multi-set with four elements instead of three:

1`(2,"g and An") + 2`(3,"alysis b") + 1`(5,"of Colou").

The integers in front of the `-operator are called **coefficients**. In our example *(2,"g and An")* and *(5,"of Colou")* have one as coefficient, while *(3,"alysis b")* has two as coefficient. All other values of the type have zero as coefficient (and hence they are omitted).

For multi-sets, we define operations for addition, scalar-multiplication, comparison, size and subtraction as illustrated in Fig. 2, where all multi-sets have elements from the set *{a,b,c,d,e}*. Notice that subtraction of two multi-sets $m_2 - m_1$ only is defined when $m_2 \geq m_1$.

Now let us consider the remaining nine places in Fig. 1. The place *A* represents packets that have been given to the *Network* by the *Sender* part of the protocol (but not yet transmitted by the *Network*). Analogously, place *B* represents packets that have been transmitted by the *Network* (but not yet taken by the *Receiver* part of the protocol). These two places have the same type as place *Send*, and initially they contain no tokens. Place *Received* will contain a single token representing the data in those packets that have been received (ignoring the contents of duplicates and packets received out of order). Initially, no data has been received and hence there is a token with the empty text string *""* (of type *DATA*). At the end of the transmission we expect *Received* to contain the text string:

"Modelling and Analysis by Means of Coloured Petri Nets".

Places *C* and *D* are analogous to places *A* and *B*, except that they represent acknowledgements being sent from the *Receiver* to the *Sender*. Each acknowledgement carries a number and no other data. Hence the types of *C* and *D* are *INT*. The places *NextSend* and *NextRec* represent counters that keep the number of the next packet to be sent/received. They have the type *INT* and each of them starts with a single token with value *1*. The two last places *SP* and *SA* have the type *Ten0*, which contains all integers between zero and ten. The use of these places will be explained later.

A state of a CP-net is called a **marking**. It consists of a number of tokens positioned on the individual places. Each token carries a value which belongs to the type of the place on which the token resides. The tokens that are present at a particular

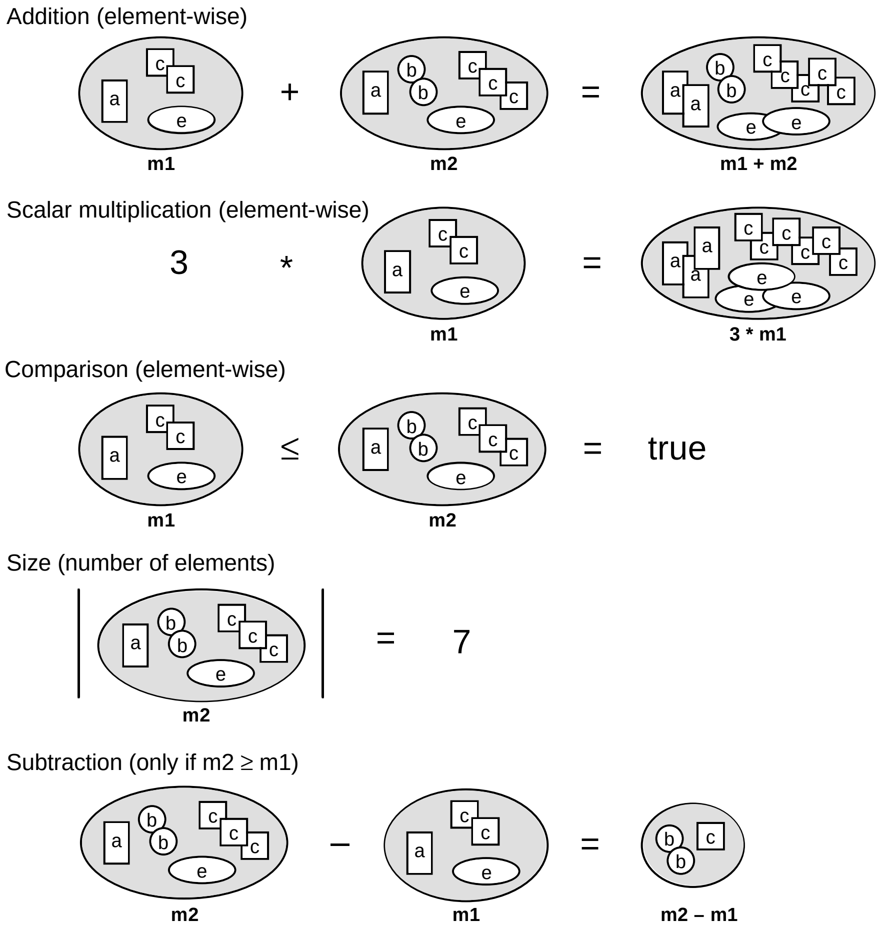

Addition (element-wise)
m1
+
m2
=
m1 + m2
Scalar multiplication (element-wise)
3
*
m1
=
3 * m1
Comparison (element-wise)
m1
≤
m2
=
true
Size (number of elements)
m2
=
7
Subtraction (only if m2 ≥ m1)
m2
–
m1
=
m2 – m1
a
b
c
e

**Fig. 2.** Some operations on multi-sets

place are called the marking of that place. By convention we write the initial marking with an underline, next to the place. When the specification of the initial marking is lengthy, we may omit the underlining (as done for *Send*).

For historical reasons we sometimes refer to token values as token **colours** and we also refer to data types as **colour sets**. This is a metaphoric picture where we consider the tokens of a CP-net to be distinguishable from each other and hence "coloured" – in contrast to ordinary low-level Petri nets which have "black" indistinguishable tokens. The types of a CP-net can be arbitrarily complex, e.g., a record where one field is a real, another a text string and a third a list of integers. Hence, it is much more adequate to imagine a continuum of colours (like in physics) instead of a few discrete colour values (like red, green and blue).

The actions of a CP-net are represented by means of **transitions** (which are drawn as rectangles). In the protocol system there are five different transitions. An incoming arc indicates that the transition may remove tokens from the corresponding place while an outgoing arc indicates that the transition may add tokens. The exact number of tokens and their data values are determined by the **arc expressions** (which are positioned next to the arcs). Transition *SendPacket* has three surrounding arcs with two different arc expressions: *(n,p)* and *n*. Two of the arcs are double arcs. Each of these is a shorthand for two opposite directed arcs with identical arc expression. Hence there are really five different arcs (two incoming arcs and three outgoing). The arc expressions contain two free variables: *n* of type *INT* and *p* of type *DATA*. To talk about an **occurrence** of the transition *SendPacket* we need to bind *n* to a value from *INT* and *p* to a value from *DATA*. Otherwise, we cannot evaluate the arc expressions *(n,p)* and *n*.

Now let us assume that we bind the variable *n* (of transition *SendPacket*) to the value *1*, while we bind the variable *p* to the value *"Modellin"*. This gives us the binding:

< n = 1,  p = "Modellin">

for which the arc expressions evaluate to:

(n,p) → (1,"Modellin")
n → 1.

This tells us that an occurrence of transition *SendPacket* (with the above binding) will remove a token with value *(1,"Modellin")* from place *Send* and a token with value *1* from place *NextSend*. Both tokens are available, i.e., present at the two places, and hence transition *SendPacket* is **enabled** with the given binding. This means that the transition may **occur**. When the transition occurs, the two specified tokens will be removed from the input places *Send* and *NextSend*. Simultaneously, three tokens will be added to the output places: *Send* and *A* will get a token with value *(1,"Modellin")*, while *NextSend* will get a token with value *1*. Hence the total effect of the occurring transition is to add a token representing packet number one to place *A*. Intuitively, this means that the *Sender* part of our model transfers a copy of packet number one to the input buffer of the *Network*. We do not remove the packet from place *Send*. This is because it may be necessary to retransmit it. Neither do we increase the counter *NextSend*. This is because our protocol is pessimistic, in the sense that it will keep retransmitting a packet, until it gets a positive acknowledgement confirming that the packet has been received.

There are of course many other bindings that we may try for transition *SendPacket*. However, none of these are enabled in the initial marking of the protocol system. This can be seen as follows. Place *NextSend* has only one token and this token carries the value *1*. Hence, we need to bind the variable *n* to *1*. This means that the arc expression on the incoming arc from place *Send* will evaluate to a value on the form *(1,…)*. However, *Send* only has one token on this form: *(1,"Modellin"),* and hence the variable *p* must be bound to *"Modellin"*.

A pair consisting of a transition and a binding (for the variables appearing on the surrounding arcs) is called a **binding element**. Above, we have seen that the binding element:

(SendPacket, <n = 1, p = "Modellin">)

is enabled in the initial marking. We have also seen that it will lead to a marking which is identical to the initial marking, except that a new token with value *(1,"Modellin")* has been added to place *A*. The new marking is shown in Fig. 3, which is a screen dump taken from the CPN simulator. The number of tokens on each place is indicated in the small circle next to the place, while the detailed token values are indicated in the text string next to the small circle. The token values can be shown or hidden. This is convenient, e.g., when the values are complex. As an example, there are CPN models in which a typical token value is a list of up to *50,000* bank records. When this is the case, no one will like to have the token value displayed directly on the CPN diagram.

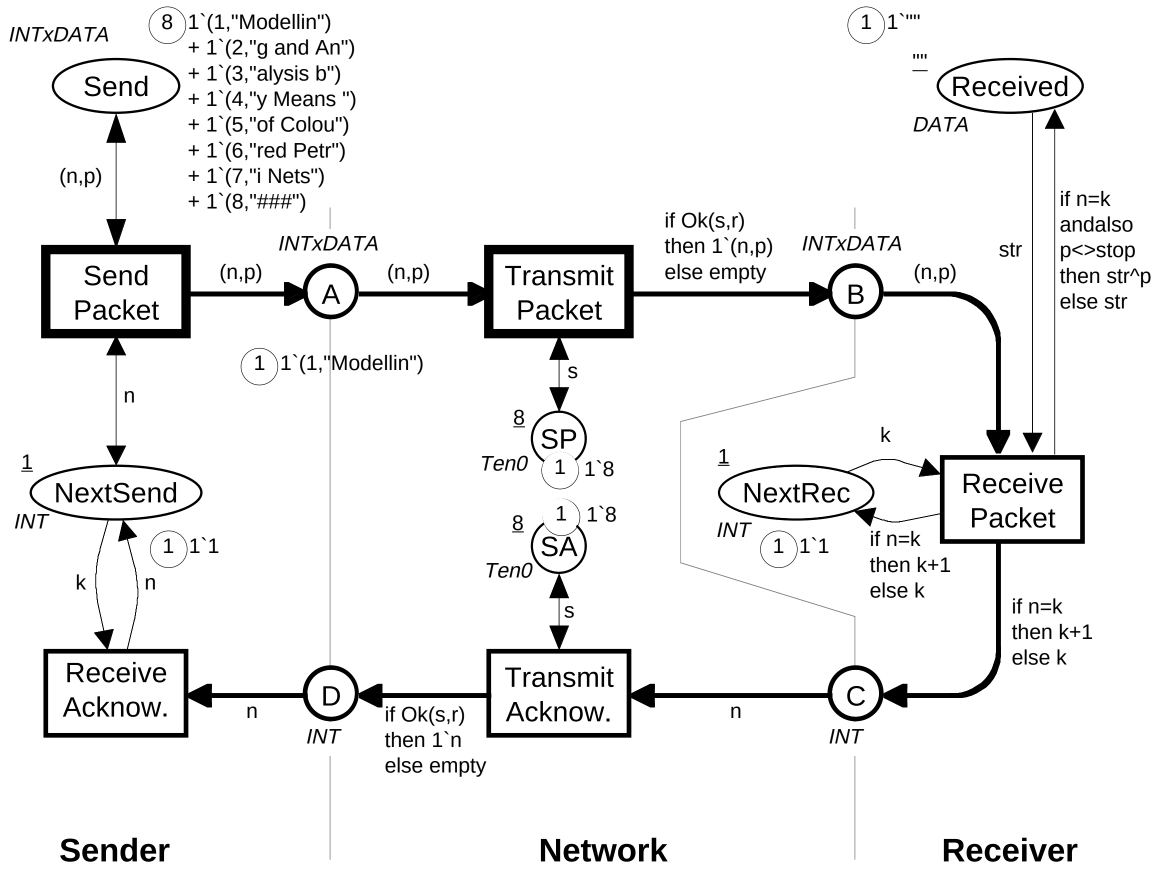


**Fig. 3.** Marking after occurrence of *SendPacket*

In Fig. 3, transitions *SendPacket* and *TransmitPacket* have a thicker border line. In this way the CPN simulator indicates that these two transitions have enabled bindings, while the other transitions have not. For *SendPacket*, we can use the binding which we used above (and no others). This corresponds to a retransmission of packet number one. For *TransmitPacket*, the situation is slightly more complex, since we now have four different variables: *n* of type *INT*, *p* of type *DATA*, *s* of type *Ten0* and *r* of type *Ten1*. *Ten0* contains all integers between zero and ten, while *Ten1* contains all integers between one and ten (all the mentioned values included). In the marking of Fig. 3, place *A* has a single token with value *(1,"Modellin")*. From this it follows that *n* must be bound to *1* while *p* must be bound to *"Modellin"*. Place *SP* has a single token with value *8*. From this it follows that *s* must be bound to *8*. The variable *r* only appears on an outgoing arc, and hence it can be bound to any value of its type – without influencing the enabling of the transition. This means that we get ten different enabled bindings:

< n = 1, p = "Modellin", s = 8, r = 1 >
< n = 1, p = "Modellin", s = 8, r = 2 >
< n = 1, p = "Modellin", s = 8, r = 3 >
..............................................
..............................................
< n = 1, p = "Modellin", s = 8, r = 9 >
< n = 1, p = "Modellin", s = 8, r = 10 >.

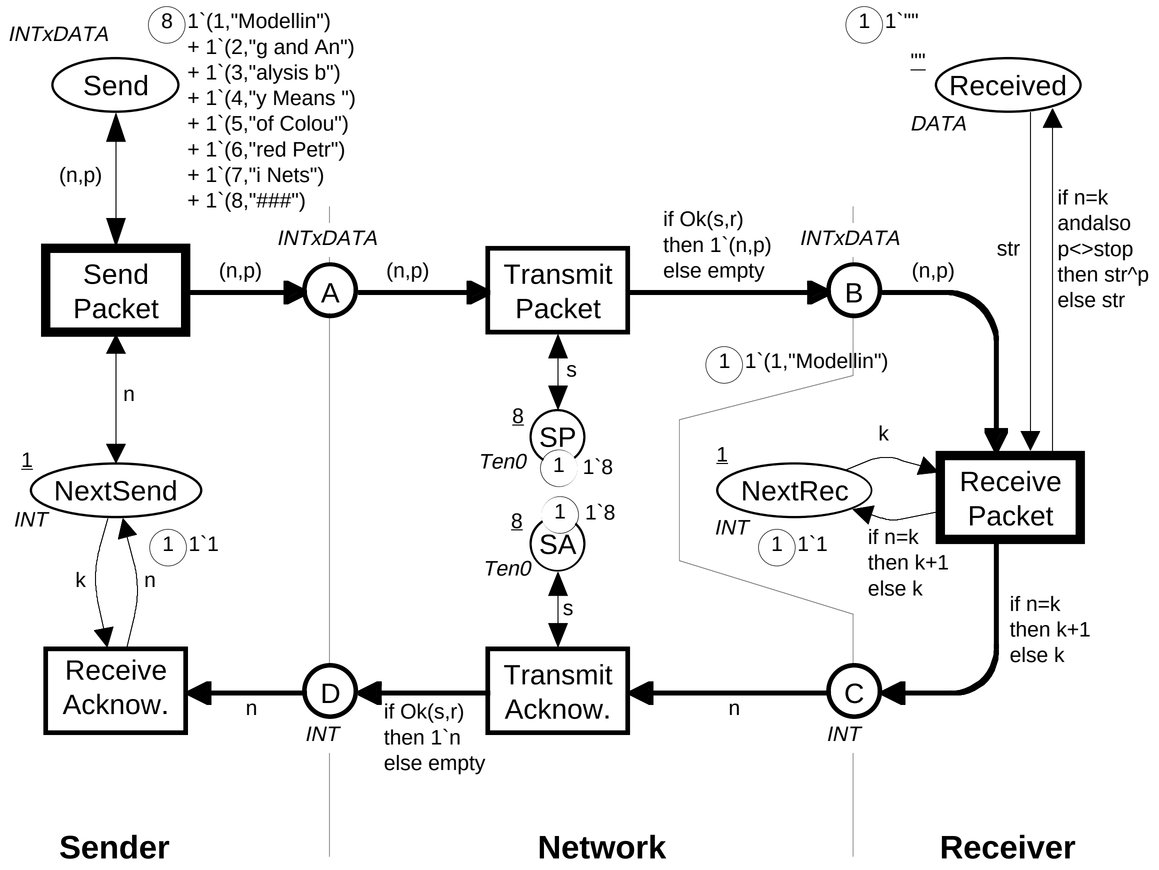


**Fig. 4.** Marking after occurrence of *SendPacket* and *TransmitPacket*

The function call *Ok(s,r)* compares the values of *r* and *s*, and it returns *true* if and only if $r \leq s$. This means that *Ok(s,r)* will evaluate to *true* for the first eight bindings, while it will evaluate to *false* for the last two bindings. When *Ok(s,r)* is *true* the transition adds a token *(1,"Modellin")* to *B*, otherwise no token is added (*empty* denotes the empty multi-set). The CPN simulator will make a fair selection between the ten enabled binding elements. Hence, the probability for successful transmission is *80%,* while the probability for losing the packet is *20%.* By changing the value of the token on *SP*, we can change the probabilities. If the token value is *10*, we never lose packets. If it is *0* we lose all packets. *SP* is a shorthand for success rate for packets.

Now, let us assume that transition *TransmitPacket* occurs with one of the first eight bindings. This will lead to the marking shown in Fig. 4. Again we have two enabled transitions. We can either retransmit packet number one by means of the binding element:

(SendPacket, <n = 1, p = "Modellin">),

or we can receive packet number one by means of the binding element:

(ReceivePacket, <n = 1, p = "Modellin", k = 1, str = "">).

When the latter binding element occurs, we remove/add the tokens shown in Fig. 5, which again is (part of) a screen dump from the CPN simulator. It is taken at a breakpoint during the occurrence of transition *ReceivePacket*.

The arc expressions on the three outgoing arcs compare the number *n* of the incoming packet with the number *k* of the expected packet. If the values are identical

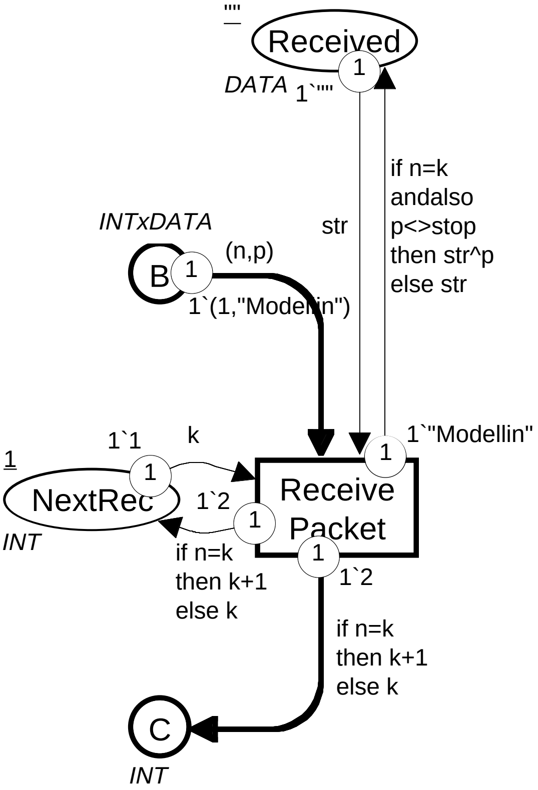


**Fig. 5.** Tokens involved in an occurrence of *ReceivePacket*

(as in our situation) the packet is the expected one. Then the *Receiver* adds the data *p* in the new packet to the data *str* which already has been *Received* (unless *p* is equal to *stop* which denotes the constant *"###"*). The token value at *NextRec* is increased by one, and an acknowledgement is sent via place *C*. By convention the acknowledgement contains the number of the next packet that the *Receiver* wants to get (i.e., the value at *NextRec*). If the values of *n* and *k* differ from each other, the packet is not the expected one. Then the packet is ignored. *Received* and *NextRec* remain unaltered, and an acknowledgement is sent via *C*. As before, the acknowledgement contains the number of the next packet which the *Receiver* wants to get (i.e., the value at *NextRec*).

After the occurrence shown in Fig. 5, we have a marking in which place *C* has a token with value *2*. The token represents an acknowledgement and it can be transmitted (or lost) by means of *TransmitAcknowledgement*. This transition works in a similar way as *TransmitPacket*. This means that the acknowledgement may be lost, with a probability determined by the token at place *SA*.

If the acknowledgement reaches place *D*, transition *ReceiveAcknowledgement* becomes enabled. It updates the number in *NextSend* by replacing the old value *k* with the number *n* contained in the acknowledgement (in our case *2*). This means that the *Sender* now starts sending packet number two, i.e., *(2,"g and An")*.

After the occurrence of approximately fifty binding elements, the CP-net may reach the intermediate marking shown in Fig. 6. From the left-hand part of the net, we see that the *Sender* is sending packet number three. We also see that a copy of

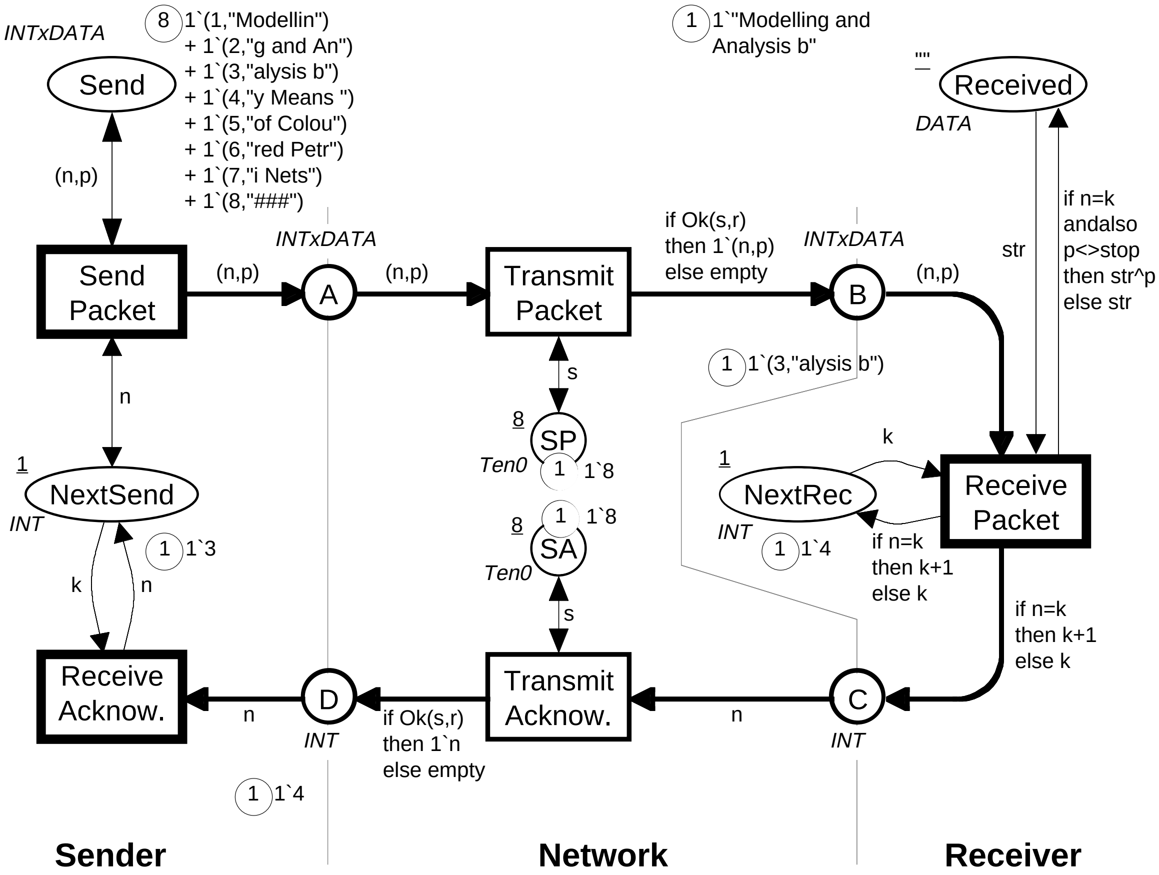


**Fig. 6.** Intermediate marking

this packet is present at place *B*. From the right-hand part of the net, we see that the 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 packet on *B* will be ignored by the *Receiver*. We also see that an acknowledgement is present at place *D*. When *ReceiveAcknowledgement* occurs, *NextSend* will be updated to *4*, and this means that the *Sender* will start sending packet number four.

Notice that there is no guarantee that tokens are removed from a place in the same order as they were added. During a simulation place *A* may contain several tokens and any of these may be selected as the next one to be transmitted to *B*. Analogously, for places *B*, *C*, and *D*. Hence packets may overtake each other at *A* and *B*, while acknowledgements may overtake each other at *C* and *D*. If desired, it is easy to specify a queuing discipline. To do this we equip the places *A*, *B*, *C*, and *D* with a type that contains all lists over the previous type of the place. Each of the places always has a single token. The initial value is the empty list, and we insert packets at one end of the list and remove packets from the other end. This construction is used so often that it will probably be directly supported in one of the next versions of the CPN tools. Then a place can be specified as being a **queuing place**, and there will be no need to make the explicit insert and remove operations. This will make the model more readable and faster to create. However, it is only syntactical sugar for the list construction explained above, and hence it does not alter the simulation or the state space analysis.

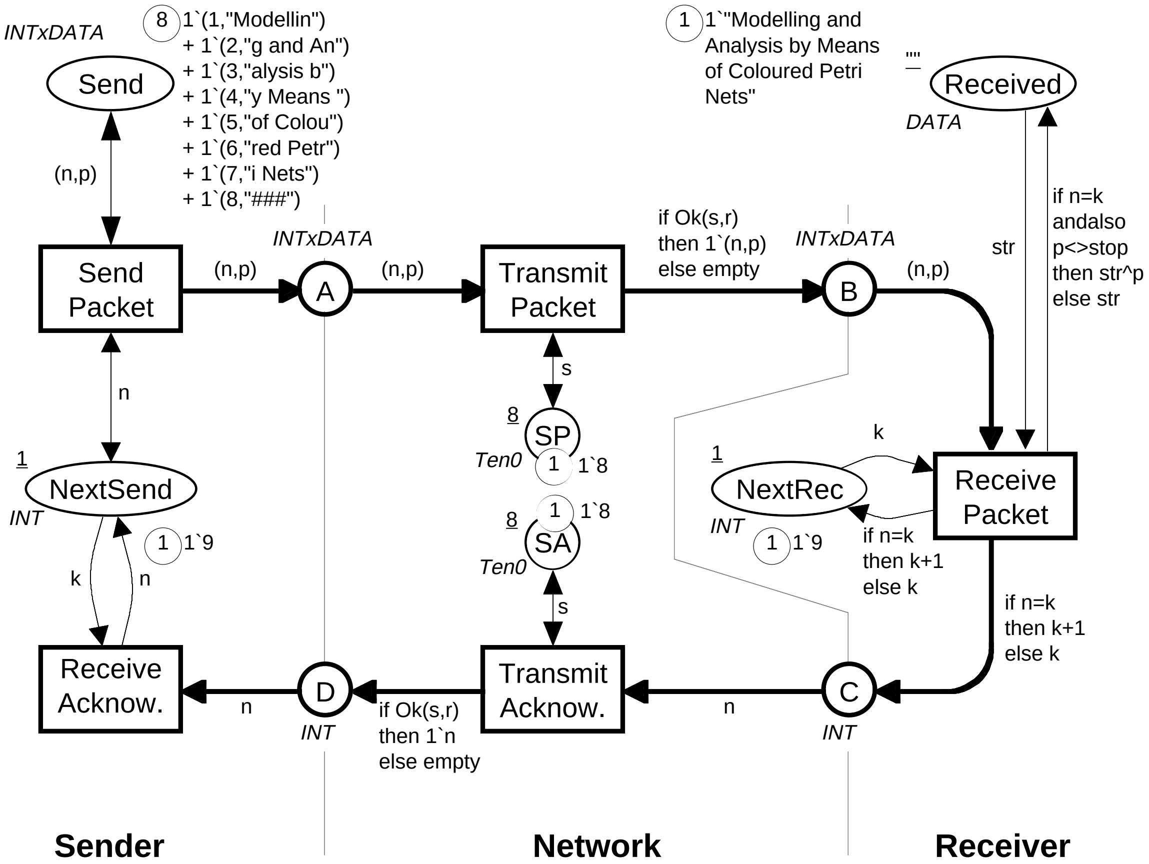


**Fig. 7.** Final marking in which no transitions are enabled

When the last packet (with "###") is successfully received by the *Receiver*, *NextRec* gets the value *9* (one larger than the number of packets). This value will (via an acknowledgement) be communicated to the *Sender*. Then *NextSend* will be updated to nine and sending will stop – since no packet with this number exists. After a few more steps, where the places *A*, *B*, *C*, and *D* are cleared for packets/ acknowledgements, the CP-net will reach the final marking shown in Fig. 7. This marking is **dead**, which means that it has no enabled transitions.

Even though the protocol is rather simple, it is not that easy to see that it actually works correctly. What happens, for instance, if the “last” acknowledgement gets lost? By making a number of simulations the user can greatly increase his confidence in the protocol. He may also prove the correctness by using the state space tool or investigate the performance by means of timed CP-nets. We shall return to this in Sects. 2, 3, and 4, respectively.

In this paper, 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 *NextSend* and *NextRec* are reset to *1*. These details can easily be added, but they are not necessary for the discussion in this paper.

In addition to the arc expressions, it is possible to attach a boolean expression (with variables) to each transition. The boolean expression is called a **guard**. It specifies that we only accept bindings for which the boolean expression evaluates to *true*. As an example, we could add a guard $n \leq 100$ to transition *SendPacket.* This would prevent the sending of messages that have more than one hundred packets.

Above, we have seen that several binding elements may be enabled in a marking. As an example, we saw that the marking in Fig. 4 has two enabled binding elements (one involving transition *SendPacket* and the other involving transition *Receive Packet*). Actually, these two binding elements are **concurrently enabled**, which means that they may **occur concurrently**. The rule for concurrency is very simple. A set of binding elements are concurrently enabled, if there are so many tokens that each binding element can get those tokens that it needs (i.e., those specified by the input arc expressions) – without sharing the tokens with other binding elements. In general, it is possible for a transition to be concurrently enabled with itself (using two different bindings or using the same binding twice). Hence, a **step**, from one marking to the next, may involve a multi-set of binding elements. The multi-set is demanded to be finite and non-empty. In the protocol example, each transition has at least one input place with only one token. Two binding elements involving the same transition will both need this single token. Hence the binding elements will be in **conflict** with each other, and no transition can occur concurrently with itself.

An execution of a CP-net is described by means of an **occurrence sequence**. It lists the markings that are reached and the steps that occur. Above, we have considered an occurrence sequence with five steps. We started in the initial marking and ended in a marking where the value of *NextSend* and *NextRec* had been increased to *2*. To reach this marking we used five steps, which each contained a single binding element. First we used a binding for *SendPacket*, then a binding for *Transmit Packet*, a binding for *ReceivePacket*, a binding for *TransmitAcknowledgement*, and finally a binding for *ReceiveAcknowledgement*.

There are many other occurrence sequences. When a marking has several enabled binding elements, any non-empty and non-conflicting subset of these may be chosen for the next step. This means that the CP-net has a non-deterministic behaviour. As an example, the CP-net for the protocol specifies that retransmissions may take place, but without providing details about when and how often this happens. Transition *SendPacket* is enabled in all reachable markings (except those where the value at *NextSend* has passed the number of the last packet). This means that retransmissions may take place at any time, and with any frequency. Hence, we have occurrence sequences with no retransmissions (as the one we considered above) and we also have occurrence sequences with a lot of retransmissions.

At first glance, it may seem strange that we do not specify the conditions under which retransmissions occur. However, for a lot of purposes this is not necessary. Most CP-nets are used to investigate the logical and functional correctness of a system design. For this purpose it is often sufficient to describe that retransmissions may appear, e.g., because the network is slow. However it is not necessary, or even beneficial, to consider how often this happens – the protocol must be able to cope with all kinds of networks, both those which work so well that there are no retransmissions and those in which retransmissions are frequent. Later on we will see that CP-nets can be extended with a time concept that allows us to describe the duration of the individual actions and states. This will allow us to investigate the performance of the modelled system, i.e., how fast and effectively it operates. Then we will give a much more precise description of retransmissions (e.g., that they occur when no acknowledgement has been received inside two hundred microseconds).

It can be shown that each CP-net can be translated into a behavioural equivalent Place/Transition Net (PT-net) and vice versa. Since the expressive power of the two formalisms are the same, there is no theoretical gain by using CP-nets. However, in practice, CP-nets constitute a more compact, and much more convenient, modelling language than PT-nets – in a similar way as high-level programming languages are much more adequate for practical programming than assembly code and Turing machines.

Attaching a data value to each CP-net token allows us to use much fewer places than needed in a PT-net. In a CP-net we can attach data values to the individual tokens. In a PT-net the only way we can distinguish between tokens is by positioning them at different places. When a CP-net uses complex types (such as integers, reals, products, records, and lists), the equivalent PT-net often has an infinite or astronomical number of places.

The use of variables in arc expressions means that each CP-net transition can occur with different bindings, i.e., in many slightly different ways – in a similar way as a procedure can be executed with different parameters. Hence, we can use a single transition to describe a class of related activities, while in a PT-net we need a transition for each instance of such an activity. As an example, our CPN model of the protocol only has one *SendPacket* transition. This transition is able to handle all packets, even though they have different packet numbers and different data contents. Analogously, there are only one *ReceivePacket* transition. This transition handles all packets, both those where the packet number matches and those where it does not.

The above informal explanation of the enabling and occurrence rules tells us how to understand the behaviour of a CP-net, and it explains the intuition on which CP-nets build. However, it is very difficult (probably impossible) to make an informal explanation which is totally complete and unambiguous. Thus it is extremely important for the soundness of the CPN language and the CPN tools that the intuition is complemented by a more formal definition, which can be found in [1], [2], and [4]. However, it is not necessary for the users to know the formal definition. The correct use of the syntax is enforced by the syntax checker in the CPN editor, while the correct use of the semantics is enforced by CPN simulator and the CPN tool for state space analysis. This is analogous to programming languages, which often are very successfully applied by users who are not familiar with the formal, mathematical definitions of the languages.

In this paper we only consider different versions of the protocol system, and they are all quite simple. However, it is possible to use much more complex types, arc expressions and guards. The CPN tools described in [5] provide computer support for modelling and analysis by means of CP-nets. The CPN tools use the functional language Standard ML [7], [8], and [9] to specify types and net inscriptions. For the protocol example the declarations look as follows (which should be rather self-explanatory):

```
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);
```

Examples of much more complex CPN models can be found in Vol. 3 of [4]. There we give a quite detailed description of nineteen projects in which CP-nets and Design/CPN have been used. Most of the projects have been carried out in an industrial setting, and they have been performed by many different user groups.

Please notice that it is only the CPN tools in [5] that rely on Standard ML. The general definition of CP-nets is independent of a concrete syntax and semantics for net inscriptions, and there exists a number of other tools that rely on other languages. Information about these tools can be found on the Petri Net WWW pages, [10]. Here you can also find a lot of other information about high-level Petri nets and other kinds of Petri nets.

# 2 Simulation of CP-nets

As the individual parts of a CP-net are constructed they are investigated and debugged by means of the CPN simulator – in a similar way as a programmer tests and debugs new parts of his program. As indicated by the examples in Sect. 1, the modeller is able to inspect all details of the reached markings. He can see the set of enabled transitions and he can choose the binding elements that he wants to occur. This is the most manual and **interactive** simulation mode. It is by nature very slow – no human being can investigate more than a few markings per minute. This work mode is similar to single step debugging in an ordinary programming language, and it is often used for the first investigation of a new CPN model (or new parts of a large model). The purpose is to see whether the individual net components work as expected.

Later on it is typical to use other kinds of simulations. Some of the graphical feedback may be turned off and it may be left to the CPN simulator to choose between the enabled binding elements (by means of a random number generator). In this way it is possible to obtain much faster simulations. A totally **automatic** simulation is executed with a speed of several thousand steps per second (depending on the nature of the CPN model and the power of the computer on which the CPN simulator runs). It is obvious that no human being is able to observe the details of

```
1 A SendPack@(1:Top#1)
     { n = 1, p = "Modellin"}
2 A SendPack@(1:Top#1)
     { n = 1, p = "Modellin"}
3 A TranPack@(1:Top#1)
     { n = 1, p = "Modellin", r = 2, s = 8}
4 A TranPack@(1:Top#1)
     { n = 1, p = "Modellin", r = 6, s = 8}
5 A RecPack@(1:Top#1)
     { k = 1, n = 1, p = "Modellin", str = ""}
6 A SendPack@(1:Top#1)
     { n = 1, p = "Modellin"}
7 A TranPack@(1:Top#1)
     { n = 1, p = "Modellin", r = 5, s = 8}
8 A RecPack@(1:Top#1)
     { k = 2, n = 1, p = "Modellin", str = "Modellin"}
9 A SendPack@(1:Top#1)
     { n = 1, p = "Modellin"}
10 A TranAck@(1:Top#1)
     { n = 2, r = 3, s = 8}
11 A TranAck@(1:Top#1)
     { n = 2, r = 4, s = 8}
12 A RecAck@(1:Top#1)
     { k = 1, n = 2}
13 A TranPack@(1:Top#1)
     { n = 1, p = "Modellin", r = 3, s = 8}
14 A RecPack@(1:Top#1)
     { k = 2, n = 1, p = "Modellin", str = "Modellin"}
15 A SendPack@(1:Top#1)
     { n = 2, p = "g and An"}
```

**Fig. 8.** Simulation report

such a simulation by watching the CP-net and its marking. Hence, the simulation results must be shown in some other way. A straightforward possibility is to use the **simulation report**. It is a text file containing detailed information about all the occurred binding elements. For the protocol model a report of the first *15* steps may look as shown in Fig. 8. The "A" following the step number indicates that the binding element was executed in automatic mode. The information after the @-sign specifies the page, i.e., the part of the CP-net to which the occurring transition belongs. For the protocol model this is not very interesting, since the model is so small that it only has one page. However, for a large complex model this information becomes very useful.

Another way to record the results of a simulation is to add a number of **report places** to the CPN model. Such places gather historical information about the simulation runs, without influencing the simulation.

In Fig. 9, we have added three places to the *Sender* part. 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 acknowledgements the *Sender* has received. Each acknowledgement is recorded as a pair, where the first element is a sequence number while the second is the contents of the acknowledge-

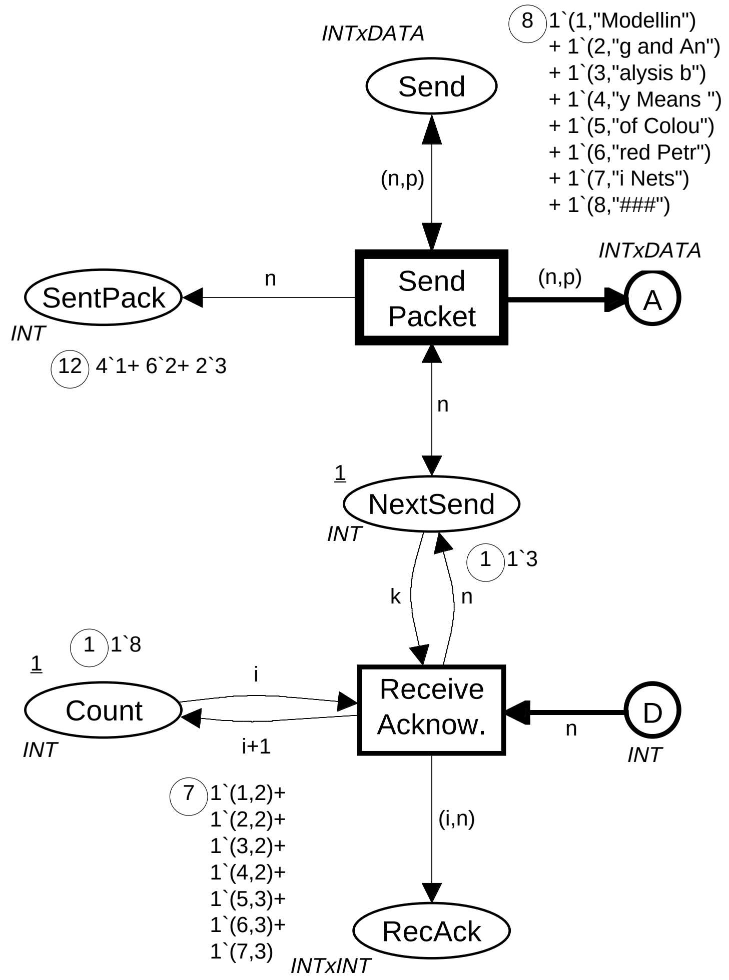


**Fig. 9.** Report places for *Sender*

ment. The sequence number is obtained from the place *Count*. In our example, we have first received four acknowledgements with value *2* and then three acknowledgements with value *3*.

In Fig. 10, we have added two places to the *Network* part. 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 acknowledgements we have transmitted/lost. In our example, we have lost one acknowledgement and successfully transmitted eight.

In Fig. 11, we have added the place *RecPack*. It records the successfully received packets. The type *PackSeq* contains all lists of *INTxDATA* values. The ^^ operator at the arc from *ReceivePacket* to *RecPack* concatenates two lists. In our example, we have received three packets. First we received *(1,"Modellin")*, then *(2,"g and An")*, and finally *(3,"alysis b")*.

A third way to record the results of a simulation is to use a **message sequence chart** (also called an event trace). It provides a graphical overview of the activities in the CP-net and may look as shown in Fig. 12. An occurrence of the *SendPacket* transition is shown by a horizontal arrow between the first and second vertical lines. The arrow is labelled with the packet being transferred to the *Network*. Analogously, a successful occurrence of *TransmitPacket* is indicated by a horizontal arrow between the second and third vertical lines. However, if the packet is lost, we

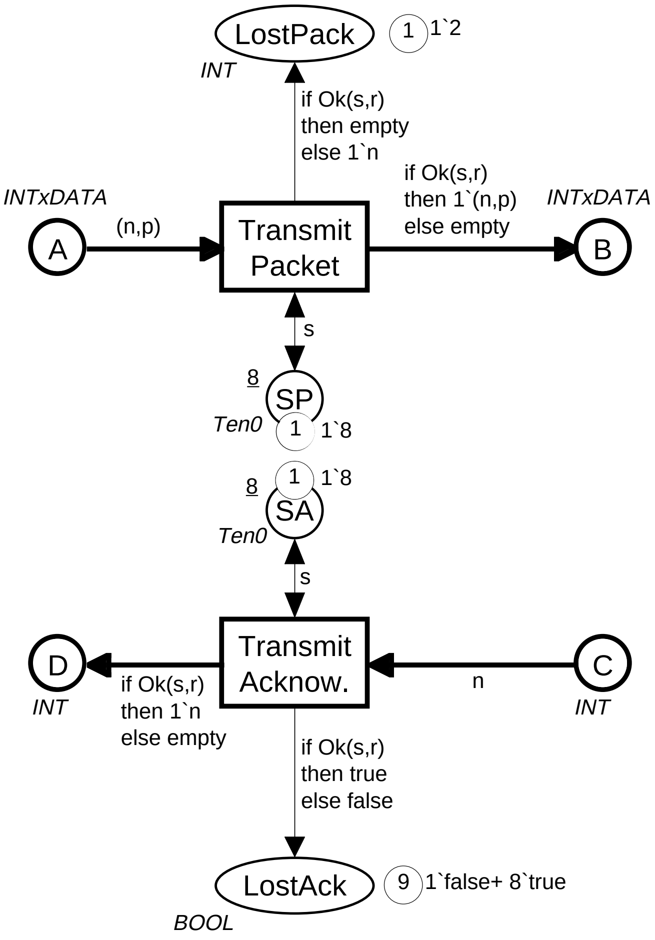


**Fig. 10.** Report places for *Network*

only get a small square at the second line. An occurrence of *ReceivePacket* is indicated by two arcs (one for the packet received and one for the acknowledgement being sent). If the packet is the correct one, we also get a small square between the two arcs. Occurrences of *TransmitAcknowledgement* are indicated in a similar way as occurrences of *TransmitPacket* (but the arrows are now drawn from right to left while the square dots are positioned at the third vertical line). Occurrences of *ReceiveAcknowledgement* are indicated as occurrences of *SendPacket* (with arrows from right to left).

In Fig. 12 we have an arc (or a square) for each step. This means that the message sequence chart contains all the information in the simulation report. However, it is much more common only to record a few key activities, e.g., the transmission of packets and acknowledgements. In this way we can obtain a condensed overview of a lengthy simulation. By extracting the key activities and representing them in a graphical way, it becomes fast to interpret the simulation results. We can see whether the CPN model behaves as expected. If this is not the case, we can see where discrepancies appear. Then we can use interactive simulations or the simulation report to make a closer investigation of these situations.

The message sequence charts are created by means of a standard library provided together with the CPN tools. The calls to the library functions are positioned in **code segments**, which are pieces of sequential Standard ML code attached to the individual transitions. When a transition occurs, the corresponding code segment is executed. It may, e.g., read and write text files, update graphics or even calculate values to be bound to some of the variables of the transition. In this way the code segments provide a very convenient interface between the CPN model and its environment, e.g., the file system.

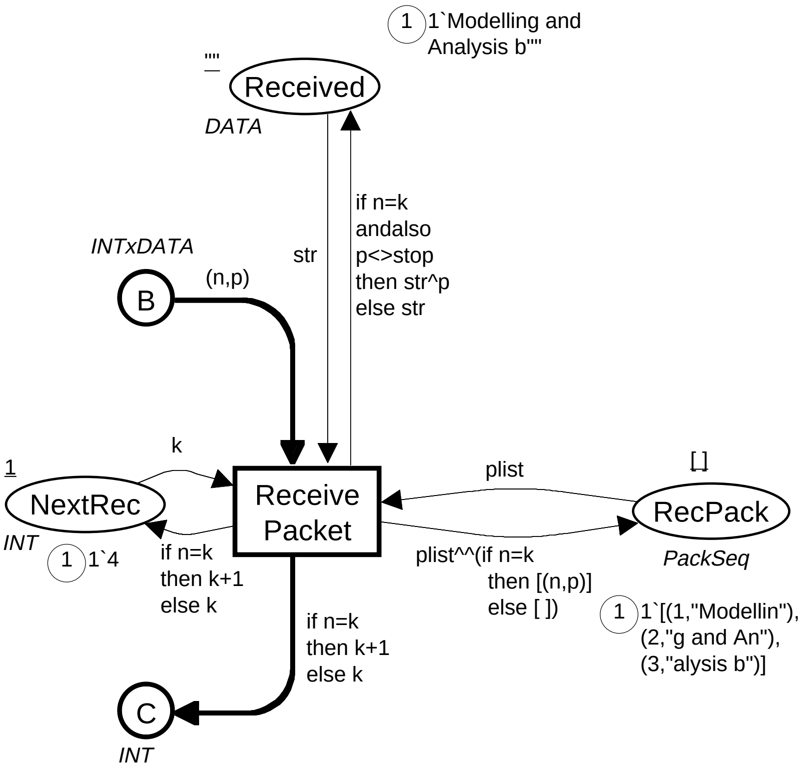


**Fig. 11.** Report place for *Receiver*

To update the message chart in Fig. 12, we give *Send Packet* and *Transmit Packet* the code segments shown in Fig. 13. *MSCdiagram* is a pointer to the message sequence chart, while *mkst_col'INTxDATA* is a predeclared function providing a string representation of the ML value *(n,p)*. The code segments of the remaining three transitions are similar.

Code segments can also be used to update different kinds of **business charts**. For the simple protocol we may use the three charts shown in Fig. 14.

The first chart is a **line chart** showing how fast the individual packets are successfully received (as a function of the step number). From the line chart, we can see that packet number one was received after less than ten steps, packet number two after approximately twenty-five steps, packet number three after approximately

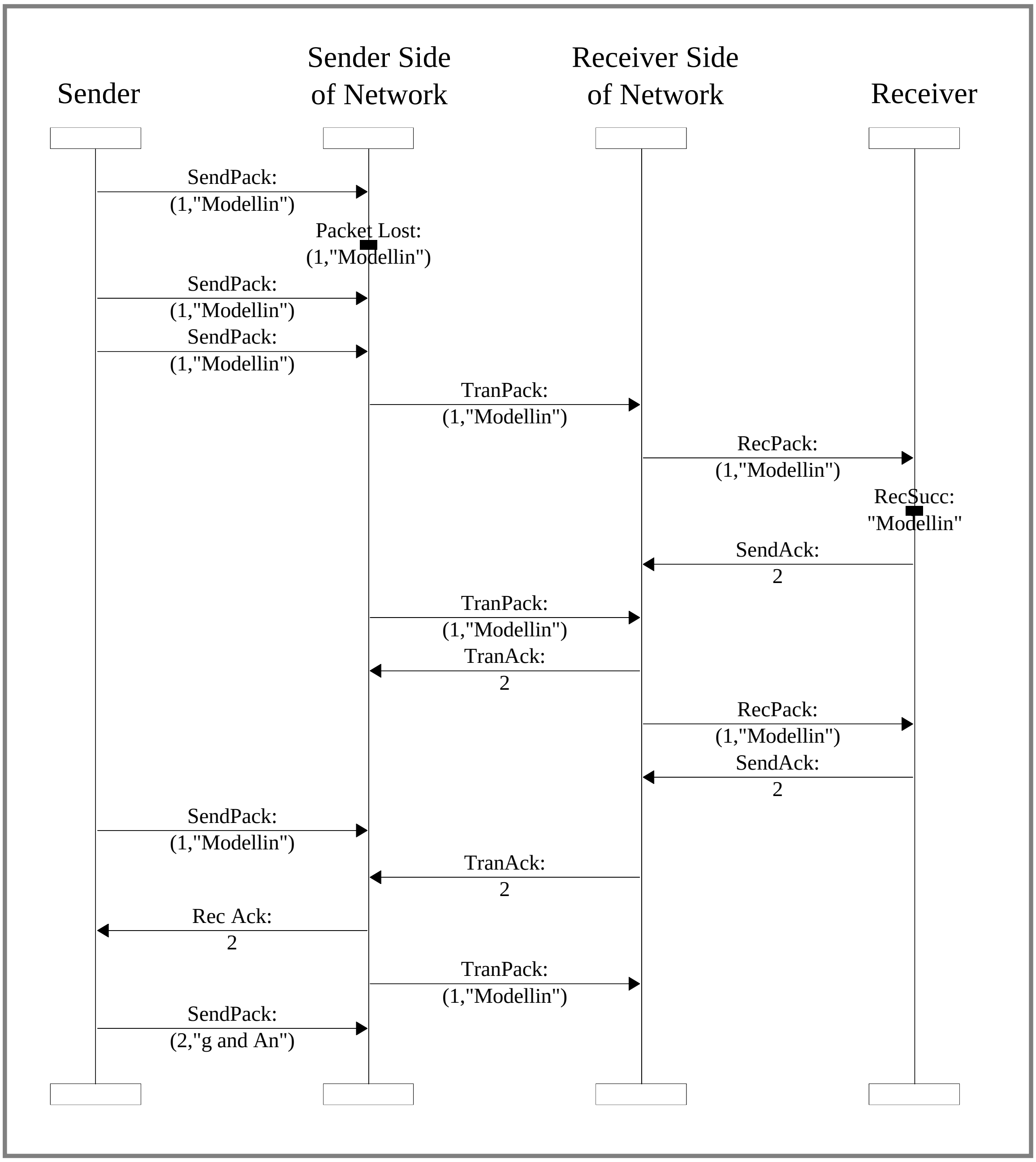


**Fig. 12.** Message sequence chart

forty-five steps, and so on. The line chart is updated each time a new packet is successfully received. This is done by a few lines in the code segment of *Receive Packet.*

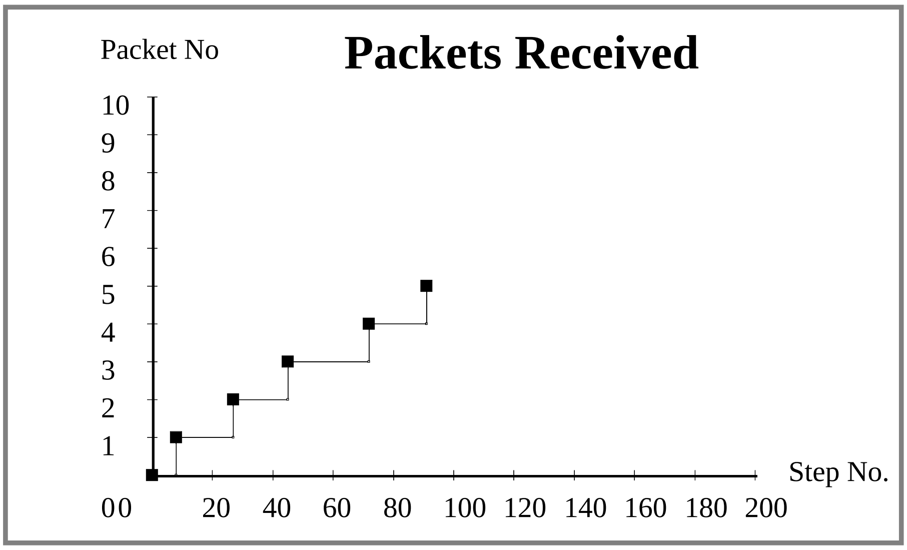
The second chart is a **bar chart**. It tells us how many times each of the packets has been sent and with which result. From the bar chart we see that packet number one has been sent six times. One of these was lost, four were received as failures (i.e., out of order) and the last was successfully received. Analogously, we can see that packet number two has been sent six times, while packets number three and four have been sent five times each. Finally, we see that packet number five has been sent twice, and that both of these are en route (i.e., on one of the *Network* places *A* and *B*). The third chart is similar to the second, but shows the progress of acknowledgements. The two bar charts are updated periodically, with intervals specified by the modeller, e.g., for each fifty steps.

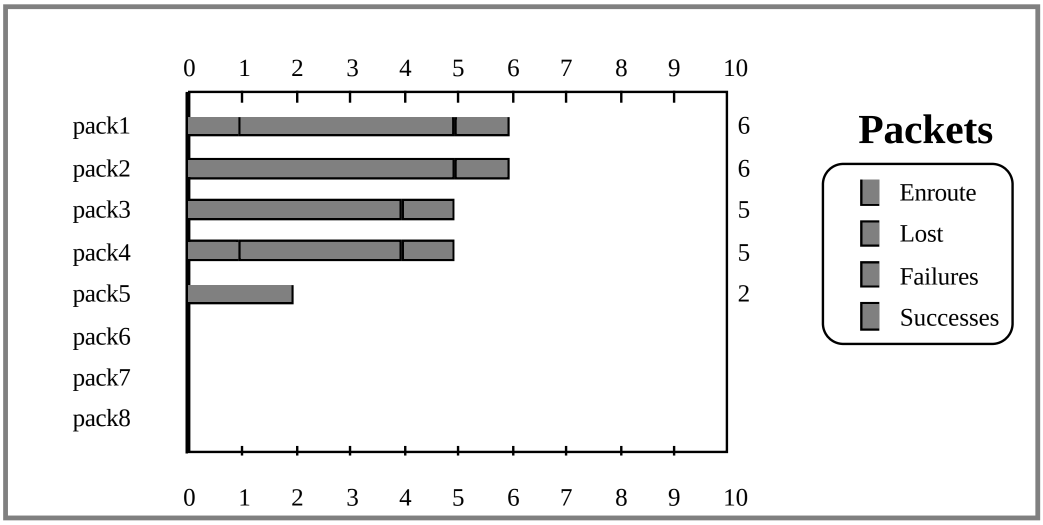
The three charts in Fig. 14 give us a lot of valuable information about the behaviour of the protocol. As an example, it is straightforward to see that failures (i.e., overtaking) often cause more retransmissions than lost packets. It is also easy to see that we need more than ninety steps to successfully transmit the first five packets, while it with a perfect network (and no overtaking) should be possible to do this in twenty-five steps.

```
input (n,p);
action
    MSC.Message (!MSCdiagram)
           {sender = "sender",
            receiver = "sendernet",
            label = "SendPack:"^NEWLINE^
                       (mkst_col'INTxDATA(n,p))};
```

```
input (n,p,s,r);
action
    if Ok(s,r) then
        MSC.Message (!MSCdiagram)
              {sender = "sendernet",
               receiver = "receivernet",
               label = "TranPack:"^NEWLINE^
                          (mkst_col'INTxDATA(n,p))}
    else
        MSC.Processmark (!MSCdiagram)
              {process = "sendernet",
               label = "Packet Lost:"^NEWLINE^
                          (mkst_col'INTxDATA(n,p))};
```

**Fig. 13.** Code segments to produce the message sequence chart in Fig. 12





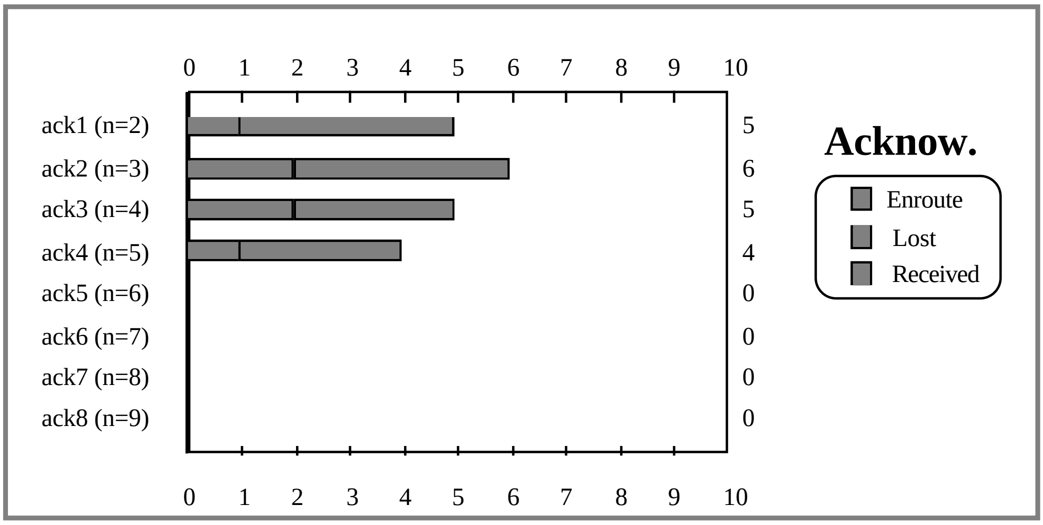


**Fig. 14.** A line chart and two bar charts

Code segments can also be used to create and update more model specific representations of the marking (i.e., the state of the modelled system). As an example, Fig. 15 represents the state of a simple telephone protocol. It contains ten different phones. For each phone, we see the state, e.g., *Inactive* for *u(9)*, *Ringing* for *u(6)* and *Connected* for *u(7)* and *u(8)*. Moreover, we see the relationships between the phones. A thin dashed arrow indicates that a connection has been requested, i.e., that a number has been dialled. The calling phone *u(10)* has *NoTone*, while the called phone *u(2)* may be in any state. A thicker non-dashed arrow indicates that the request has been accepted and that the connection is being attempted. The calling phone *u(2)* has a tone with *Long* intervals between beeps, while the called phone *u(6)* is *Ringing*. Finally, connections can be established, as indicated by the very thick arrow from *u(8)* to *u(7)*. Both phones are *Connected*.

As mentioned above, code segments can be used to write text files. In this way selected results can be recorded, e.g., on a form which can be directly used as input to a standard spreadsheet/charting program. This is a very efficient way of obtaining customised high-quality representations of complex simulation results.

Code segments can also be used to read text files. This is, e.g., used to initialise a model. In this way it becomes possible to change the initial marking without modifying the CPN model itself.

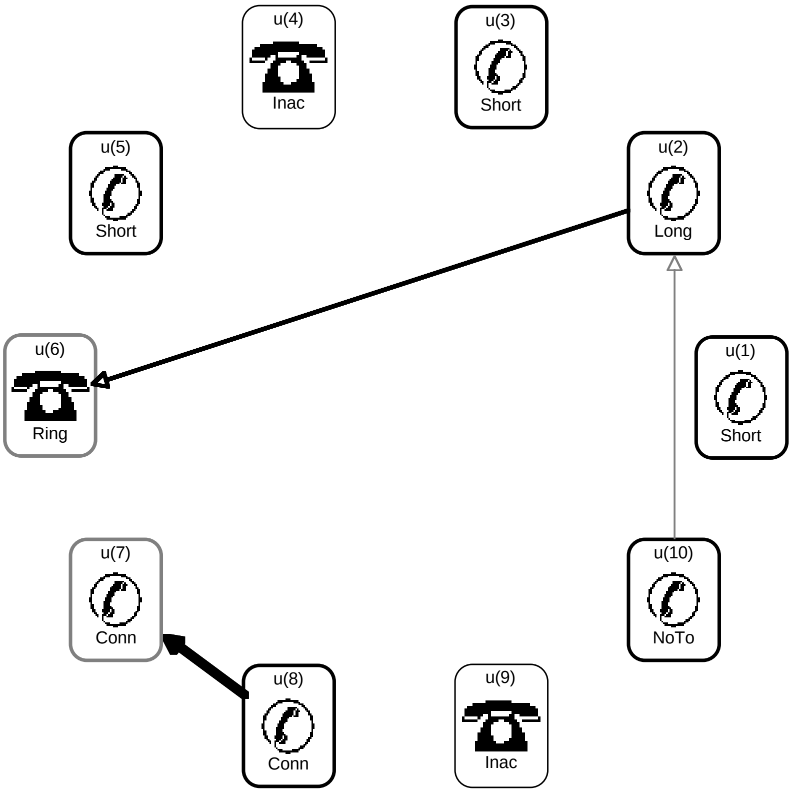


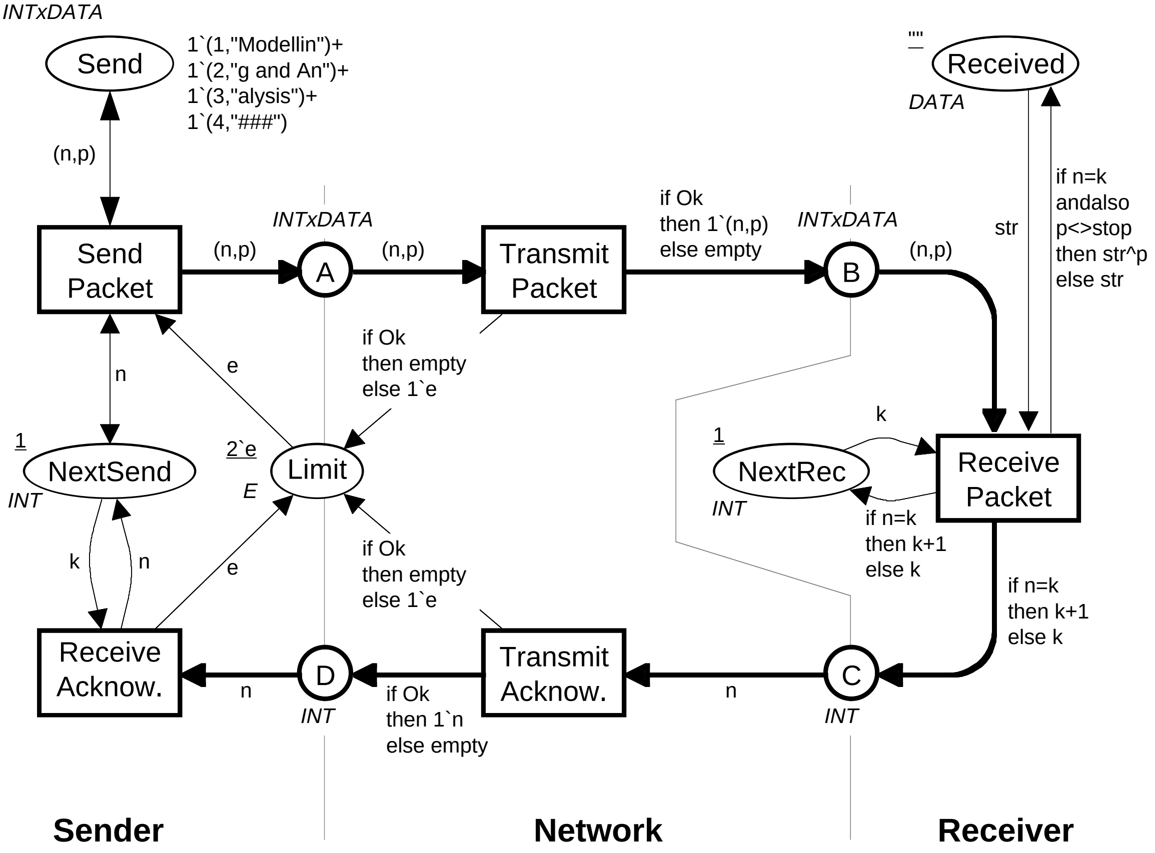
**Fig. 15.** A more system specific representation of the system state

# 3 State Space Analysis of CP-nets

As explained in Sect. 2 it is customary to debug and investigate a CPN model by means of simulation. This works in a similar way as program testing and hence it can never prove the correctness of a model (unless it is trivial). Hence, we often complement simulation with the construction of one or more state spaces.

The basic idea behind a **state space** is to construct a graph which has a node for each reachable marking and an arc for each occurring binding element. State spaces are also called **occurrence graphs** or **reachability graphs/trees**. The first of these names reflects the fact that a state space contains all the possible occurrence sequences, while the two latter names reflect that the state space contains all reachable markings.

To be able to construct a state space of reasonable size, we often have to modify the CP-net. For our protocol system, we make three modifications. Firstly, we re-



```
color INT = int;
color DATA = string;
color INTxDATA = product INT * DATA;
var n, k : INT;
var p,str : DATA;
val stop = "###";

color E = with e;
color BOOL = bool;
var Ok : BOOL;
```

**Fig. 16.** Modified CP-net suitable for state space analysis

duce the number of packets from eight to four. Secondly, we introduce a new place to *Limit* the number of packets/acknowledgements which can be present simultaneously at the *Network*, i.e., at one of the places *A, B, C,* and *D*. The new place has a type *E* with only one possible value *e*. Intuitively, this means that tokens of this type carry no data. Finally, we simplify the decision mechanism for transmitting/losing packets and acknowledgements. For state spaces it does not make sense that packets are transmitted/lost with a certain probability. Hence, we replace the *Ok* function with a boolean variable *Ok*. The modified CPN model for the protocol can be seen in Fig. 16.

Having made the modifications described above, we are ready to construct state spaces. Let us start with the situation where the initial marking of place *Limit* is *1`e*. This means that the *Network* (i.e., the places *A, B, C,* and *D*) contains at most one packet/acknowledgement at a time. Hence overtaking is impossible. The state space is small. It has *33* nodes and *44* arcs. The initial and final parts of it are shown in Figs. 17 and 18.

The rounded boxes are the nodes of the state space. Each of them represents a reachable marking, and the content of this marking is described in the dashed box next to the node – places with an empty marking are omitted and we also omit the markings of *Send* and *Limit* (the first one never changes, the second is only added to limit the size of the state space). At the top of Fig. 17, we have a node with a thicker borderline. This node represents the initial marking. The text inside the node tells us that this is node number one and that it has one predecessor and one successor (the latter information may be useful when we have drawn only a part of a state space). Analogously, we see that node number two has one predecessor and two successors. By convention we use $M_n$ to denote the marking of node number *n*.

Each arc represents the occurrence of the binding element listed in the dashed box on top of the arc. In $M_1$ the only enabled binding element is transition *SendPacket* with binding *< n = 1, p = "Modellin">*. When this binding element occurs, we reach marking $M_2$, in which there are two enabled binding elements. An occurrence of *TransmitPacket* with the variable *Ok* bound to *true* will lead to marking $M_3$, while an occurrence of *TransmitPacket* with *Ok* bound to *false* will lead back to the initial marking $M_1$. By convention we do not include arcs that correspond to steps with more than one binding element. Such arcs would give us information about the concurrency between binding elements, but they are not necessary for the verification of standard behavioural properties.

From Figs. 17 and 18, we can see that the state space has a regular structure, in the sense that some patterns are repeated. The subgraph of nodes *{4, 5, 7, 9}* has the same form as the subgraph of nodes *{28, 29, 31, 33}*. The only difference is that the latter is “three packets ahead” of the former. If we draw the middle part of the state space, we will find two additional copies of the pattern.

Now let us investigate the more complex situation in which overtaking is possible. To do this, we construct a state space for the situation where the initial marking of place *Limit* is *2`e*. The new state space is considerably larger than the first one, and hence we do not make any attempt to draw it. Instead we ask the state space tool to make a **state space report** providing some key information about the behaviour of the CP-net. The state space report has four parts.

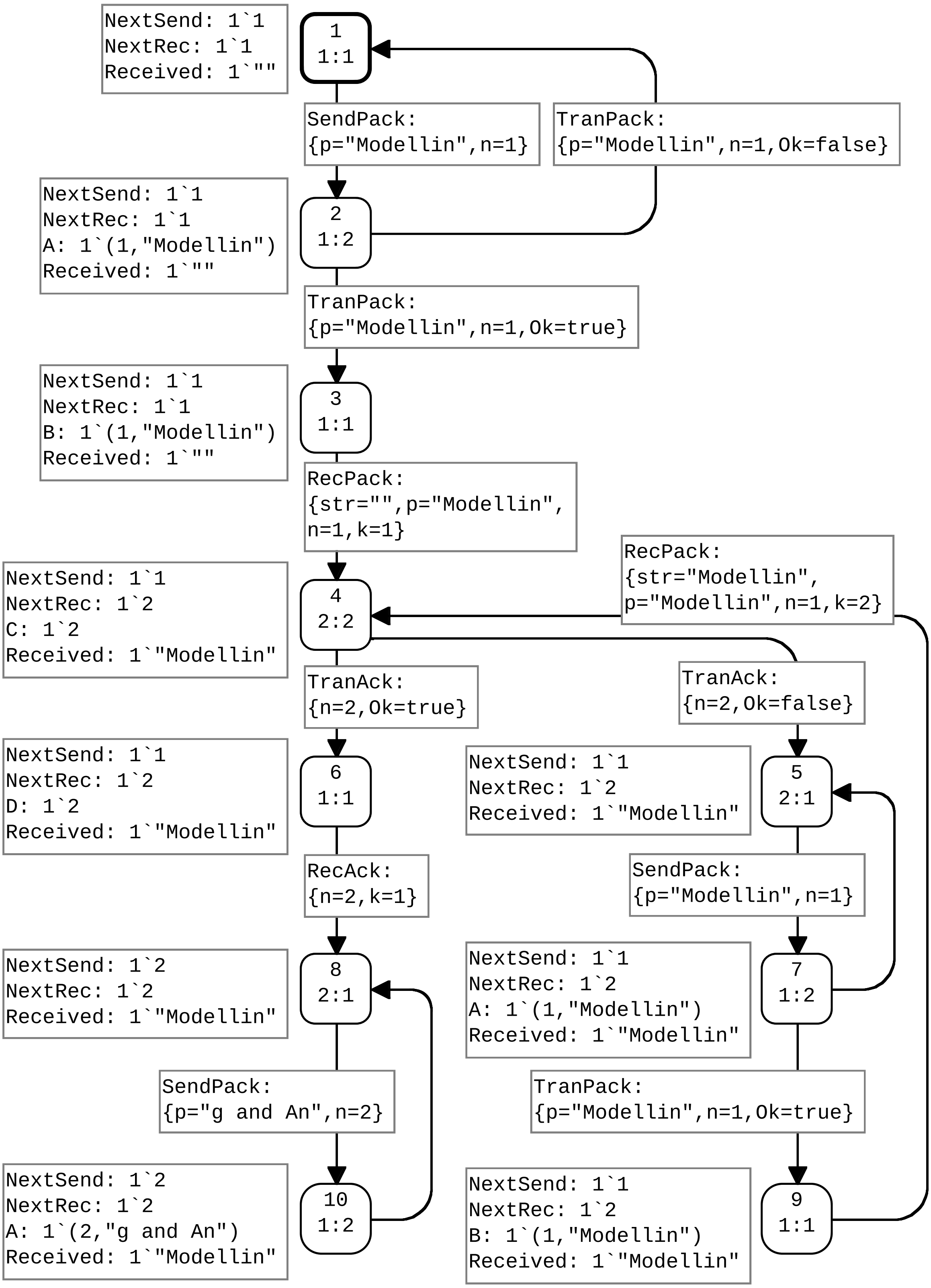


**Fig. 17.** Initial part of state space

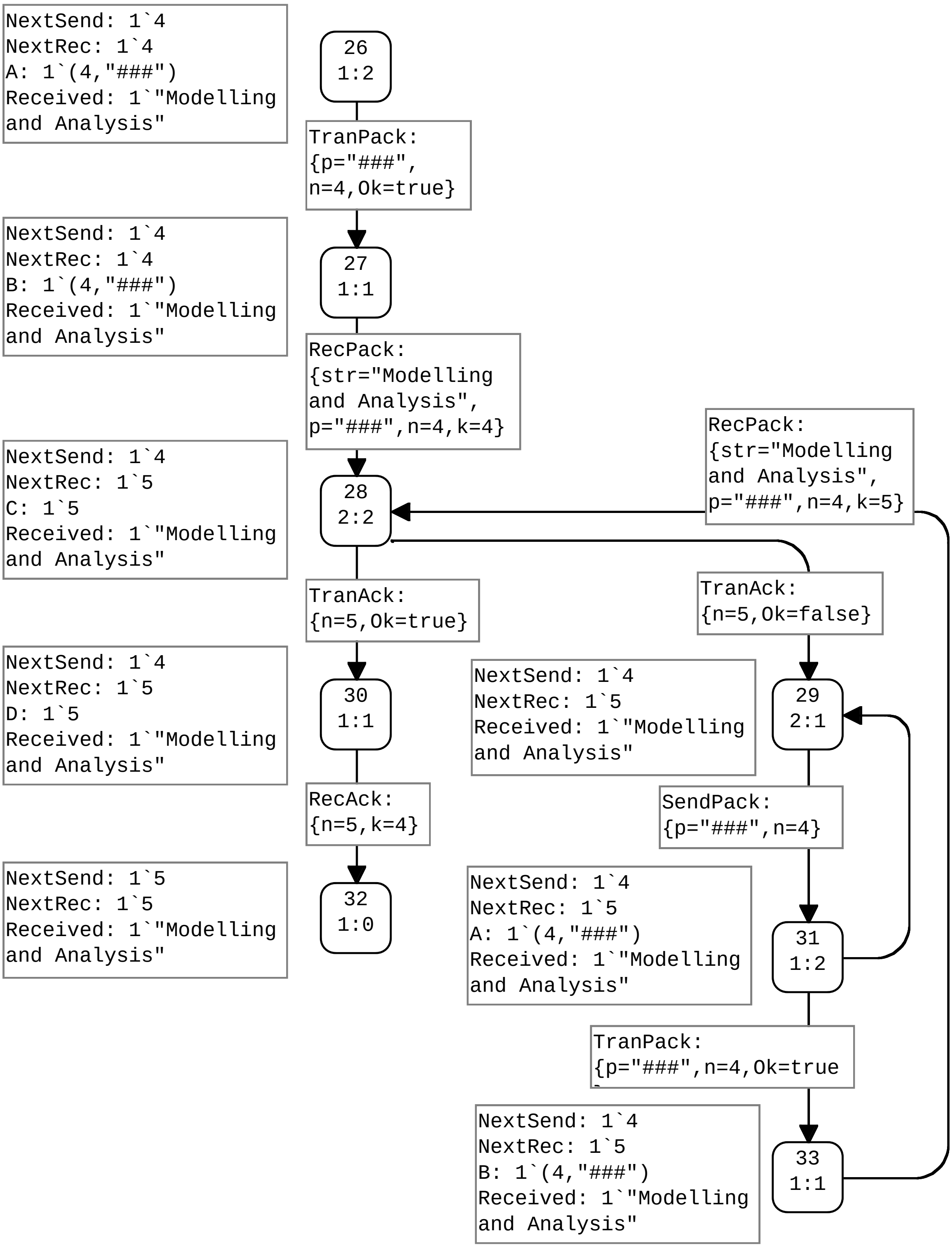

NextSend: 1`4
NextRec: 1`4
A: 1`(4,"###")
Received: 1`"Modelling and Analysis"
26
1:2
TranPack: {p="###", n=4,Ok=true}
NextSend: 1`4
NextRec: 1`4
B: 1`(4,"###")
Received: 1`"Modelling and Analysis"
27
1:1
RecPack: {str="Modelling and Analysis", p="###",n=4,k=4}
RecPack: {str="Modelling and Analysis", p="###",n=4,k=5}
NextSend: 1`4
NextRec: 1`5
C: 1`5
Received: 1`"Modelling and Analysis"
28
2:2
TranAck: {n=5,Ok=true}
TranAck: {n=5,Ok=false}
NextSend: 1`4
NextRec: 1`5
D: 1`5
Received: 1`"Modelling and Analysis"
30
1:1
NextSend: 1`4
NextRec: 1`5
Received: 1`"Modelling and Analysis"
29
2:1
RecAck: {n=5,k=4}
SendPack: {p="###",n=4}
NextSend: 1`5
NextRec: 1`5
Received: 1`"Modelling and Analysis"
32
1:0
NextSend: 1`4
NextRec: 1`5
A: 1`(4,"###")
Received: 1`"Modelling and Analysis"
31
1:2
TranPack: {p="###",n=4,Ok=true
NextSend: 1`4
NextRec: 1`5
B: 1`(4,"###")
Received: 1`"Modelling and Analysis"
33
1:1

**Fig. 18.** Final part of state space

The first part looks as shown in Fig. 19. It contains **statistical** information about the size of the state space. We see that the state space has *428* nodes and *1130* arcs. We have calculated the *full* state space, and this took only *1* second. The statistical part also contains information about the SCC-graph of the state space, i.e., the number of **strongly connected components** and the number of arcs that start in one component and end in another. A strongly connected component is a maximal subgraph in which it is possible to find a path from any node to any other node. Strongly connected components are very useful to determine certain kinds of behavioural properties, and they can be calculated by standard algorithms (which are linear in time and space). From Fig. 19 we see that there are *182* strongly connected components and *673* arcs that start in one component and end in another. Hence there are less strongly connected components than state space nodes. This implies that the system has at least one strongly connected component with more than one node, and hence infinite occurrence sequences exist. In other words, we cannot be sure that the protocol terminates – to achieve termination one usually limits the number of retransmissions.

The second part of the state space report contains information about the integer and multi-set bounds. The upper part of Fig. 20 shows the upper and lower **integer bounds**, i.e., the maximal and minimal number of tokens which the individual places may have. We see that each of the places *A*, *B*, *C*, *D*, and *Limit* always has between zero and two tokens. We also see that *NextRec*, *NextSend,* and *Received* always have exactly one token each. Finally, we see that *Send* always has exactly four tokens. None of this is surprising, but it is reassuring, because it indicates that the system is working as expected.

The lower parts of Fig. 20 show the **multi-set bounds**. By definition, the upper multi-set bound of a place is the smallest multi-set which is larger than all reachable markings of the place. Analogously, the lower multi-set bound is the largest multi-set which is smaller than all reachable markings of the place. The integer bounds give us information about the number of tokens, while the multi-set bounds give us information about the values which the tokens may carry. From the multi-set bounds we see that places *A* and *B* may contain all four different packets,

```
Statistics

 State Space
   Nodes:  428
   Arcs:   1130
   Secs:   1
   Status: Full

 Scc Graph
   Nodes:  182
   Arcs:   673
   Secs:   1
```

**Fig. 19.** Size of state space and the SCC-graph

```
 Integer Bounds

           Upper      Lower
A            2          0
B            2          0
C            2          0
D            2          0
Limit        2          0
NextRec      1          1
NextSend     1          1
Received     1          1
Send         4          4

Upper Multi-set Bounds

A            2`(1,"Modellin")+ 2`(2,"g and An")+
             2`(3,"alysis")+ 2`(4,"###")

B            2`(1,"Modellin")+ 2`(2,"g and An")+
             2`(3,"alysis")+ 2`(4,"###")

C            2`2+ 2`3+ 2`4+ 2`5
D            2`2+ 2`3+ 2`4+ 2`5
Limit        2`e
NextRec      1`1+ 1`2+ 1`3+ 1`4+ 1`5
NextSend     1`1+ 1`2+ 1`3+ 1`4+ 1`5

Received     1`""+ 1`"Modellin"+ 1`"Modelling and An"+
             1`"Modelling and Analysis"

Send         1`(1,"Modellin")+ 1`(2,"g and An")+
             1`(3,"alysis")+ 1`(4,"###")

Lower Multi-set Bounds

A            empty
B            empty
C            empty
D            empty
Limit        empty
NextRec      empty
NextSend     empty
Received     empty

Send         1`(1,"Modellin")+ 1`(2,"g and An")+
             1`(3,"alysis")+ 1`(4,"###")
```

**Fig. 20.** Integer and multi-set bounds

while places *C* and *D* may contain all four possible acknowledgements. Remember that an acknowledgement always specifies the number of the next packet to be sent (hence we never have an acknowledgement with value *1*). We also see that no other token values are possible at these four places.

Notice that the upper multi-set bound of *A* is a multi-set with eight elements, although the upper integer bound tells us that there never can be more than two tokens on *A* at a time. The two tokens can take the following four token values: (1,"Modellin"), (2,"g and An"), (3,"alysis"), and (4,"###"). Hence, we can, e.g., have the markings:

1\`(1,"Modellin")
1\`(1,"Modellin") + 1\`(3,"alysis")
2\`(3,"alysis").

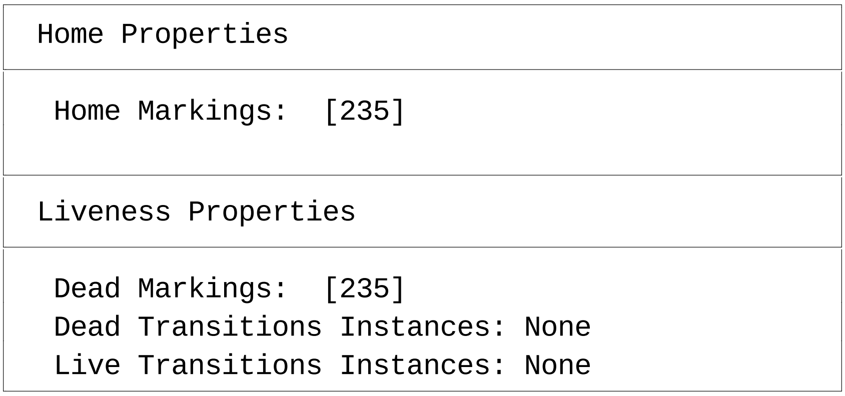
By this kind of argument, it is easy to see that the smallest multi-set which is larger than all possible markings of *A* is the multi-set:

2\`(1,"Modellin") + 2\`(2,"g and An") + 2\`(3,"alysis") + 2\`(4,"###").

A similar remark applies to the multi-set bounds of *B, C, D, NextRec, NextSend,* and *Received*.

The two counter places *NextSend* and *NextRec* can take all values between *1* and *5*. Place *Received* may contain four different values – corresponding to the situations where we have received data from zero, one, two or three packets (packet number four contains "###" which we never copy to *Received*). Finally, place *Send* has identical upper and lower multi-set bounds. This means that the marking of this place never changes. Also the multi-set bounds are as expected, and this indicates that the protocol works as intended. However, the multi-set bounds also give us some new knowledge. As an example, there are probably many readers who did not notice, until now, that acknowledgements never carry the value *1*.

The third part of the state space report is shown in Fig. 21. It provides information about home and liveness properties. A **home marking** is a marking which is reachable from all reachable markings, i.e., a marking which always can be reached – independently of what has happened up to now. We see that the protocol has a single home marking, $M_{235}$. A **dead marking** is a marking with no enabled transitions. We see that the protocol has a single dead marking, and that the dead marking

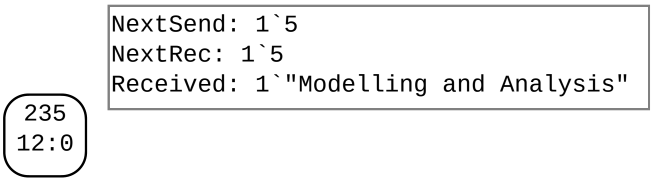


```
Home Properties

 Home Markings:  [235]

Liveness Properties

 Dead Markings:  [235]
 Dead Transitions Instances: None
 Live Transitions Instances: None
```

**Fig. 21.** Home and liveness properties

is identical to the home marking. Hence, let us take a closer look at this marking. To do this we ask the state space tool to draw the node *235*. It looks as follows:

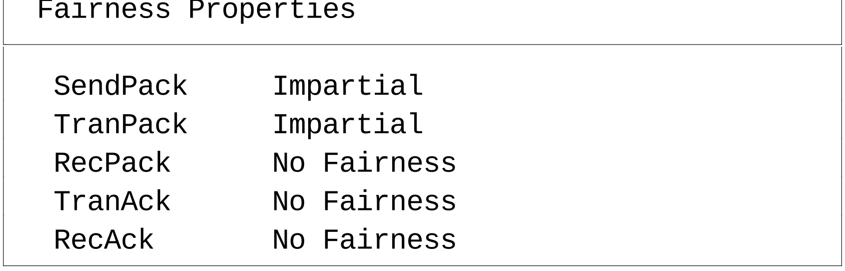
**Dead Marking / Home Marking**



From this we see that the marking $M_{235}$ corresponds to the state where all four packets have been successfully received (and no tokens are left at places *A*, *B*, *C*, and *D*). The fact that $M_{235}$ is dead tells us that the protocol is **partial correct**, i.e., if it terminates, we have the correct result. The fact that $M_{235}$ is a home marking tells us that the protocol has the nice property that it never can reach a state from which it is impossible to terminate with the correct result.

From the liveness properties we also see that there are no **dead transitions**. This means that each transition is enabled in at least one reachable marking (which is a rather weak property). We also see that there are no **live transitions**. A live transition is a transition that always, no matter what happens, can become enabled once more. When there are dead markings (as in our protocol), there cannot be any live transitions. At first glance, one might think that the existence of dead markings would prevent the existence of home markings. However, by definition, a marking is always reachable from itself, and hence a dead marking may be a home marking (provided that it is the only dead marking). This is the case for the protocol system.

The fourth and final part of the state space report is shown in Fig. 22. It provides information about the fairness properties, i.e., how often the individual transitions occur. We see that *SendPacket* and *TransmitPacket* are **impartial**. This means that each of them occurs infinitely often in any infinite occurrence sequence. In other words, if one of these transitions ceases to occur, then the protocol must terminate (after some final number of additional steps). There are two other kinds of fairness properties, called **fair** and **just**. They are weaker properties than impartiality, and hence they are automatically fulfilled by *SendPacket* and *Transmit Packet*. From Fig. 22 we see that none of the remaining three transitions are impartial, fair or just.



```
Fairness Properties

 SendPack     Impartial
 TranPack     Impartial
 RecPack      No Fairness
 TranAck      No Fairness
 RecAck       No Fairness
```

**Fig. 22.** Fairness properties

The state space report is produced in a few seconds – totally automatic. It contains a lot of highly useful information about the behaviour of the CPN model. Hence it is usually the first thing that the modeller asks for, when he has constructed a state space. By studying the state space report the modeller gets a first rough idea, whether his model works as expected. If the system contains errors they are often reflected in the state space report. As an example, one may be designing a system in which the initial marking is expected to be a home marking. However, one may forget to return a resource after its use. Then it is no longer possible to return to the initial marking, and this will be evident from the state space report. To make a closer investigation of the problem, the user can ask the state space tool to find one of those markings from which it is impossible to return to the initial marking. The modeller can also ask the system to find a path from the initial marking to the marking which has the problem. In this way a counter example has been provided. Then it is, usually, easy to spot and correct the error.

Above, we have seen that $M_{235}$ is the desired final marking, and we have also seen that it can be reached from any reachable system state. Now let us investigate how fast it can be reached from the initial marking. To do this we ask the system to construct a path from the initial marking $M_1$ to $M_{235}$. This is done by means of the following query – the question is in the rectangular box, while the result is in the rounded box.

**Length of Shortest Path**

```
length(ArcsInPath(1,235));
```

```
> 20 : int
```

*ArcsInPath* is a predeclared Standard ML function provided by the state space tool. It returns a list of arcs constituting a shortest path between the two nodes specified as arguments to the function. From the result, we see that at least twenty transitions must occur, in order to reach $M_{235}$ from $M_1$. This is not surprising. The shortest path must be one in which we have no retransmissions and no overtaking. We have four packets and to process a packet (plus the corresponding acknowledgement) we need one occurrence of each of the five transitions.

Next let us investigate the way in which we update the *NextSend* counter. One might expect that this counter is always increased (or left unchanged). However, the following query tells us that there are a number of transition occurrences that actually decrease the value of *NextSend*.

**Is NextSend Ever Decreased?**

```
let
  fun Value_NextSend(n) =
       ms_to_col(Mark.Top'NextSend 1 n)
in
  PredAllArcs(fn a =>
       Value_NextSend(DestNode(a)) <
       Value_NextSend(SourceNode(a)))
end;
```

```
>
[973,951,934,921,920,895
,894,845,844,818,817,753
,729,663,648,587,573,567
,517,499,497,429,428,360
,310,271,233] : Arc list
```

The result of the query is a list containing all those arcs that fulfil the predicate specified as the argument to the function *PredAllArcs*. For an arc *a*, the predicate evaluates to *true*, if and only if the token value at *NextSend* in the destination node of *a* is strictly less than the token value in the source node of *a*. The token value is found by means of the local function *Value_NextSend*. It uses the predeclared ML function *Mark.Top'NextSend* to find the marking of *NextSend* (in the specified node *n*). The marking is a multi-set. It is converted into an integer by means of the function *ms_to_col* (which converts a multi-set with one element into that element, e.g., *1`3* into *3*).

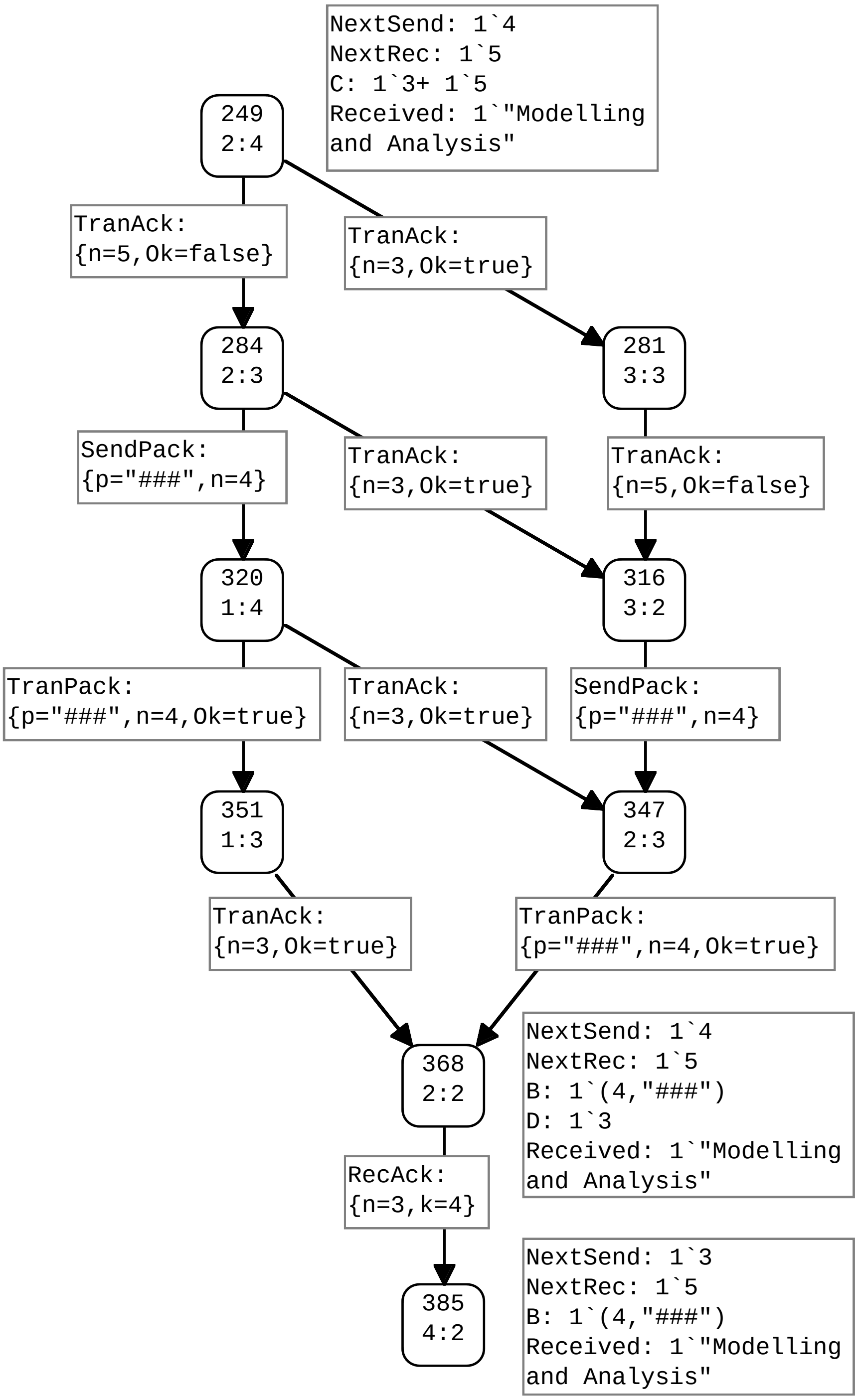


**Fig. 23.** A small interesting part of the state space

To investigate why *Next Send* is decreased, we make the drawing in Fig. 23. We ask the system to display the first arc in the result of the above query, i.e., arc number *973* (from node *368* to node *385*). Moreover, we use the *Display Predecessors* command to draw some of the nearest predecessors of node *368*. After a few "backwards" steps we find marking $M_{249}$, which is of interest. In this marking *Next Send* has the value *4* while *Next Rec* has the value *5*. There is an acknowledgement with value *5* positioned at place *C*. However, C has also an "old" acknowledgement with value *3*. This acknowledgement has existed for quite a while. It was created when packet number three was expected. The old acknowledgement has been overtaken by several "younger" acknowledgements. However, it may still proceed and cause *Next Send* to be decreased to *3*.

Another way to investigate the possible decrease of *Next Send*, is to ask how much *Next Send* can differ from the *Next Rec*. This is done by means of the following query, which tells us that the difference can be *3, 2, 1,* and *0*. This result is consistent with our analysis above. *Next Rec* can be at most five while *Next Send* is at least one, but *Next Send* can never be reset to less than two – because we never have acknowledgements with value *1*.

**Difference Between Counters**

```
let
  fun Value_NextSend(n) =
       ms_to_col(Mark.Top'NextSend 1 n);
  fun Value_NextRec(n) =
       ms_to_col(Mark.Top'NextRec 1 n)
in
  remdupl(
  EvalAllNodes(fn n =>
       Value_NextRec(n) - Value_NextSend(n)))
end;
```

```
> [3,2,1,0] : INT list
```

From our analysis above, it is quite obvious that an easy way to improve our protocol is to avoid decreasing *Next Send*. This can be achieved by modifying the arc expression of the arc from *Rec Acknowledgement* to *Next Send* – so that it becomes *max(n,k)* instead of *n*.

The algorithms used to generate the state space report and to answer the more system dependent queries build upon a number of proof rules. Each proof rule states a relationship between a behavioural property of a CP-net and a property of its state space. As an example, it is straightforward to see that a marking is dead if and only if the corresponding state space node has no outgoing arcs. As a more complex example, it can be seen that a marking is a home marking if and only if the strongly connected component to which it belongs is the only one with no outgoing arcs. A detailed description of the proof rules and the mathematical proofs of them can be found in Vol. 2 of [4].

In this section, we have demonstrated that state spaces is a very efficient way to investigate the dynamic behaviour of a system. The construction and analysis of state spaces are totally automatic. From the state space report the modeller gets a lot of knowledge about the dynamic behaviour of the system. However, he can also formulate his own queries. This can be done in two different ways. Either by using a large number of predeclared query functions written in Standard ML (as those used above) or by using a library that support queries in a CTL-like temporal logic. The first approach is the easiest for most users, while the second is the most general. For complex systems the state space may be quite large, and the construction may take several hours. This is, however, not a big problem since it is totally automatic and hence can be done overnight or in the lunch break. It is often faster to make a state space analysis than it is to make a thorough investigation by means of simulations.

One of the main drawbacks of the state space method is the so-called state explosion problem. For many systems, the state space becomes so large that it cannot be fully constructed. Our present state space tool supports state spaces with up to half a million nodes and one million arcs (when the tool runs a machine with three hundred Megabytes of memory). One of the ways to be able to handle larger state spaces is a more condensed storage of the multi-sets encountered during the state space construction. Our present representation is identical to the representation used in the CPN simulator. This is far from being optimal, and we expect to be able to improve it with as much as a factor one hundred.

A number of methods exist to alleviate the state explosion problem, i.e., to construct smaller state spaces without losing too much analytic power. Some of these methods take a modular approach, while other avoids construction of all the sequences in which concurrent binding elements can be interleaved. There are also methods that exploit the inherent symmetries found in many systems. These methods are described in Sect. 6.

Another drawback of the state space method is the fact that a state space is always constructed for a particular initial marking, which often corresponds to only one out of many different possible system configurations. Above, we have shown that the protocol works when we have at most two simultaneous packets on the *Network.* With the present capacity of the state space tool, we can perform a similar analysis for state spaces with up to five simultaneous packets. These state spaces have the sizes shown in Fig. 24. They are constructed on a Sun Ultra Sparc Enterprise 3000. This machine has also been used for all the other simulations and state spaces reported in this paper.

However, how can we know that the protocol still works when we allow more than five simultaneous packets? Unfortunately the general answer is discouraging. In theory, we cannot know this for sure. However, in practice the situation is not that bad. If the protocol works for five simultaneous packets it is very likely that it also works when more packets are allowed. Other analysis methods are more gen-

| Limit | Nodes | Arcs | Time |
|---|---:|---:|---:|
| 1 | 33 | 44 | « 1 sec. |
| 2 | 428 | 1,130 | 1 sec. |
| 3 | 3,329 | 12,825 | 14 secs. |
| 4 | 18,520 | 91,220 | 3 mins. |
| 5 | 82,260 | 483,562 | 47 mins. |

**Fig. 24.** Sizes of state spaces for the simple protocol

eral and provide a full proof that a system works as expected – for all configurations. This is e.g., the case for place invariant analysis. However, such analysis methods are often much less automatic. They usually involve a good deal of human reasoning in the form of one or more mathematical proofs. This implies that the methods are time-consuming and difficult to use by engineers. Moreover, they are error-prone. What do you prefer? A sequence of automatically constructed and automatically analysed state spaces showing that the protocol works for all cases where there are five or less simultaneous packets – or a five to ten page manual proof based on rather complex mathematical arguments.

With our present knowledge and technology, we cannot hope to verify large systems by means of state spaces. However, we can use state spaces on selected subnets. This is a very effective way to locate errors. A small mistake will often imply that we do not get, e.g., an expected marking bound or an expected home marking. It is also possible to investigate rather complex CP-nets by means of partial state spaces, where we only develop, e.g., a fixed number of outgoing arcs for each node. Such a method will very often catch design errors – although it cannot count as a full proof of the desired system properties. A partial state space corresponds to making a large number of simulation runs – the state space represents the results in a systematic way.

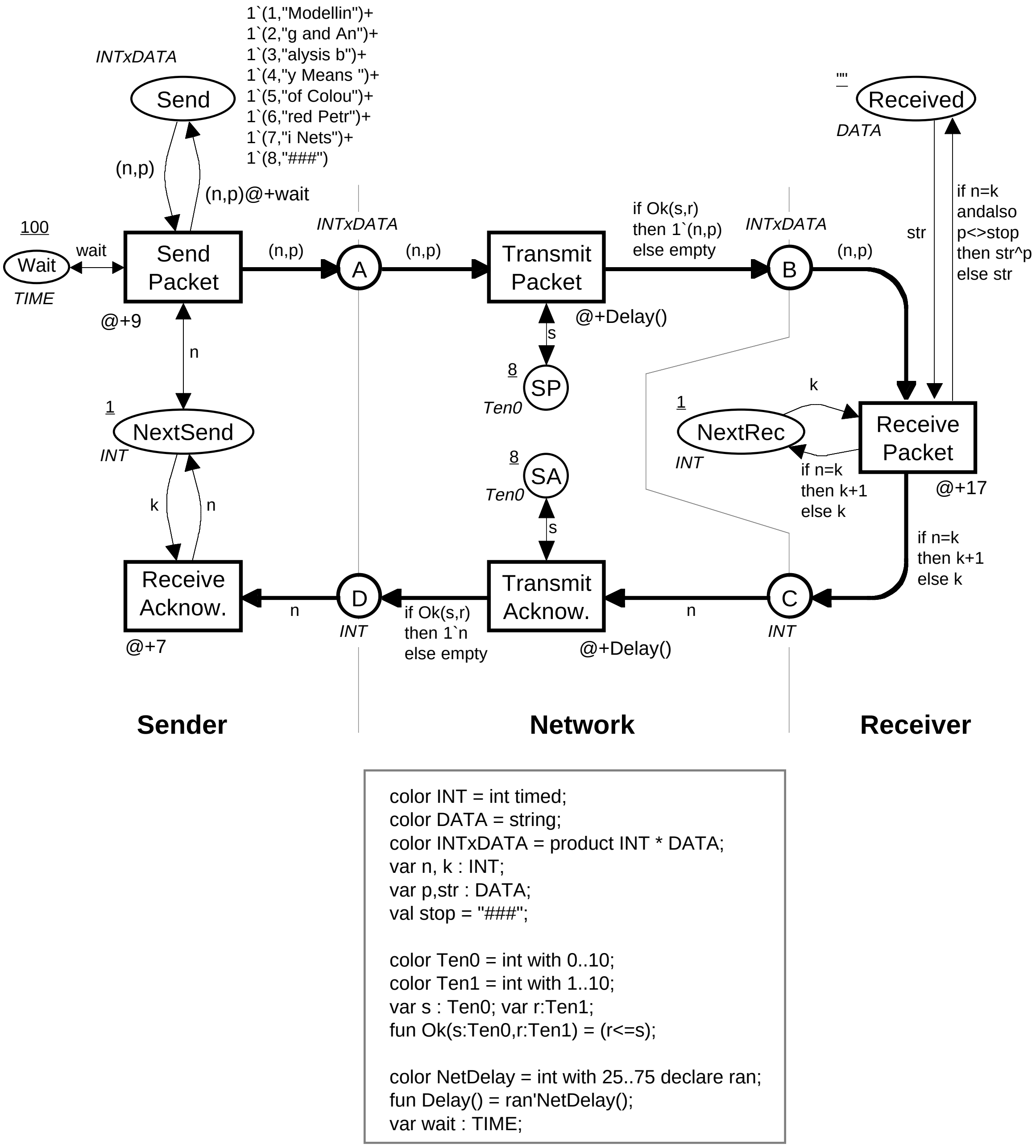
# 4 Performance Analysis of CP-nets

To investigate the performance of systems, i.e., the speed at which they operate, it is convenient to extend CP-nets with a time concept. To do this, we introduce a **global clock**. The clock values represent **model time**, and they may either be integers (i.e., discrete) or reals (i.e., continuous). In addition to the token value, we allow each token to carry a **time value**, also called a **time stamp**. Intuitively, the time stamp describes the earliest model time at which the token can be used, i.e., removed by a binding element.

In a **timed CP-net** a binding element is said to be **colour enabled** when it satisfies the requirements of the enabling rule for untimed CP-nets (i.e., when there are the required tokens at the input places and the guard evaluates to *true*). However, to be **enabled**, the binding element must also be **ready**. This means that the time stamps of the tokens to be removed must be less than or equal to the current model time.

To model that an activity/operation takes *r* time units, we let the corresponding transition *t* create time stamps for its output tokens that are *r* time units larger than the clock value at which *t* occurs. This implies that the tokens produced by *t* are unavailable for *r* time units. It can be argued that it would be more natural to delay the creation of the output tokens, so that they did not come into existence until *r* time units after the occurrence of *t* had begun. However, such an approach would mean that a timed CP-net would get “intermediate” markings which do not correspond to markings in the corresponding untimed CP-net, because there would be markings in which input tokens have been removed but output tokens not yet generated. Hence we would get a more complex relationship between the behaviour of timed and untimed nets.

The execution of a timed CP-net is time driven, and it works in a similar way to that of the event queues found in many other languages for discrete event simulation. The system remains at a given model time as long as there are colour enabled binding elements that are ready for execution. When no more binding elements can be executed, at the current model time, the system advances the clock to the next model time at which binding elements can be executed. Each marking exists in a closed interval of model time (which may be a point, i.e., a single moment). The occurrence of a binding element is instantaneous.



```
color INT = int timed;
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);

color NetDelay = int with 25..75 declare ran;
fun Delay() = ran'NetDelay();
var wait : TIME;
```

**Fig. 25.** Timed CP-net for the simple protocol

A timed CP-net for the protocol system is shown in Fig. 25. The timed net has the same net structure as the untimed net – except that a new place *Wait* has been added (in the left-hand side). This place is used to specify how long time transition *SendPacket* should wait before retransmitting a packet.

From the declarations it can be seen that type *INT* is timed. This means that the corresponding tokens carry time stamps. The types *DATA*, *Ten0* and *Ten1* are not timed. Tokens of these types do not carry time stamps, and hence they are always available. By convention, the structured type *INTxDATA* is timed – because it contains a component *INT* which is timed. Hence, we conclude that the four places *Received*, *SP*, *SA*, and *Wait* have tokens without time stamps, while the remaining seven places have tokens with time stamps. For this system we use an integer clock, starting at zero. Hence, all time stamps in the initial marking are equal to zero.

We have added a **time inscription** (starting with @+) to each of the five transitions. Intuitively, the time inscription describes how long time the corresponding operation takes. Now let us take a closer look at the five different transitions in the protocol system.

*Send Packet* has a time inscription: @+ *9*. This implies that the tokens created at *A* and *NextSend* get time stamps which are *9* time units larger than the time *r** at which the transition occurs. The output arc to place *Send* specifies an additional time delay to be used for the tokens added to *Send*. This token will get a time stamp which is *r* + 9 + 100* (since the variable *wait* is bound to *100*). Intuitively, *9* represents the time used to send a packet, while *100* is the time that has to elapse before a retransmission is done, i.e., before *SendPacket* occurs once more for the same packet. Hence, a retransmission will only happen if the number in *NextSend* remains unaltered for *9 + 100* time units, i.e., if no acknowledgement for the packet is received inside this time period.

*TransmitPacket* has a time inscription: @+ *Delay()*, where *Delay* is declared to be an ML function returning a random element from the type *NetDelay* (i.e., a random integer in the interval between *25* and *75*). This implies that the duration of a transmit operation may vary inside this interval.

*ReceivePacket* and *ReceiveAcknowledgement* have time inscriptions specifying a fixed duration (*17* and *7* time units respectively), while *Transmit Acknowledgement* has a variable duration time, between *25* and *75*. All time delays are specified by means of Standard ML expressions. Hence, it is easy to use statistical functions specifying more complex types of delays (e.g., exponential distributions).

Note that the token in *NextSend* carries a time stamp. Intuitively, this means that the *Sender* cannot start a new *SendPacket* or a new *ReceiveAcknowledgement* as long as one of these operations is already ongoing. If the *Sender* has multiple threads, allowing an unlimited number of *Sender* operations to be performed at the same time, we simply make the type of *NextSend* untimed. A similar remark applies for the operations of the *Receiver* and the type of *NextRec*.

In general, the time delays may depend upon the binding in question, i.e., upon the values of the input and output tokens. As an example it might, on some networks, be faster to lose a packet than it is to transmit it.

For a timed CP-net we require that each step consists of binding elements which are both colour enabled and ready. Hence the possible occurrence sequences of a timed CP-net always form a subset of the possible occurrence sequences of the corresponding untimed CP-net. This means that we have a well-defined and easy-to-understand relationship between the behaviour of a timed CP-net and the behaviour of the corresponding untimed CP-net.

In the timed CP-net for the protocol, we have only illustrated one of the simplest ways in which time stamps can be used. All removed tokens for a binding element were required either to be without time stamps or to have time stamps which were less than or equal to the time value *r** at which the binding element occurs. At a given place, all added tokens either got no time stamps or got identical time stamps which were equal to *r** plus a delay *r*. In general, the situation can be considerably more complex. For details see Vol. 2 of [4].

After a number of simulation steps the timed CP-net may reach a dead final marking with the contents shown in Fig. 26. In the markings displayed next to the places, we separate the time stamps from the token values by an @ sign (which usually is read as “at”). From the time stamp of *NextRec* it can be seen that the last packet was received at time *1790*. Analogously, the time stamp at *NextSend* tells us that the last operation of the *Sender* was finished at time *1832*. The time stamps at place *Send* tell us the times at which the individual packets would have been retransmitted (had this become necessary). As an example, we can see that the first packet would have been retransmitted at time *109*, the second at time *331*, the third at *359*, and so on.

By means of our timed CPN model we can investigate the performance of the protocol, e.g., experiment with different values for the retransmission delay specified by *Wait*. A short delay increases the chance of making unnecessary retransmis-

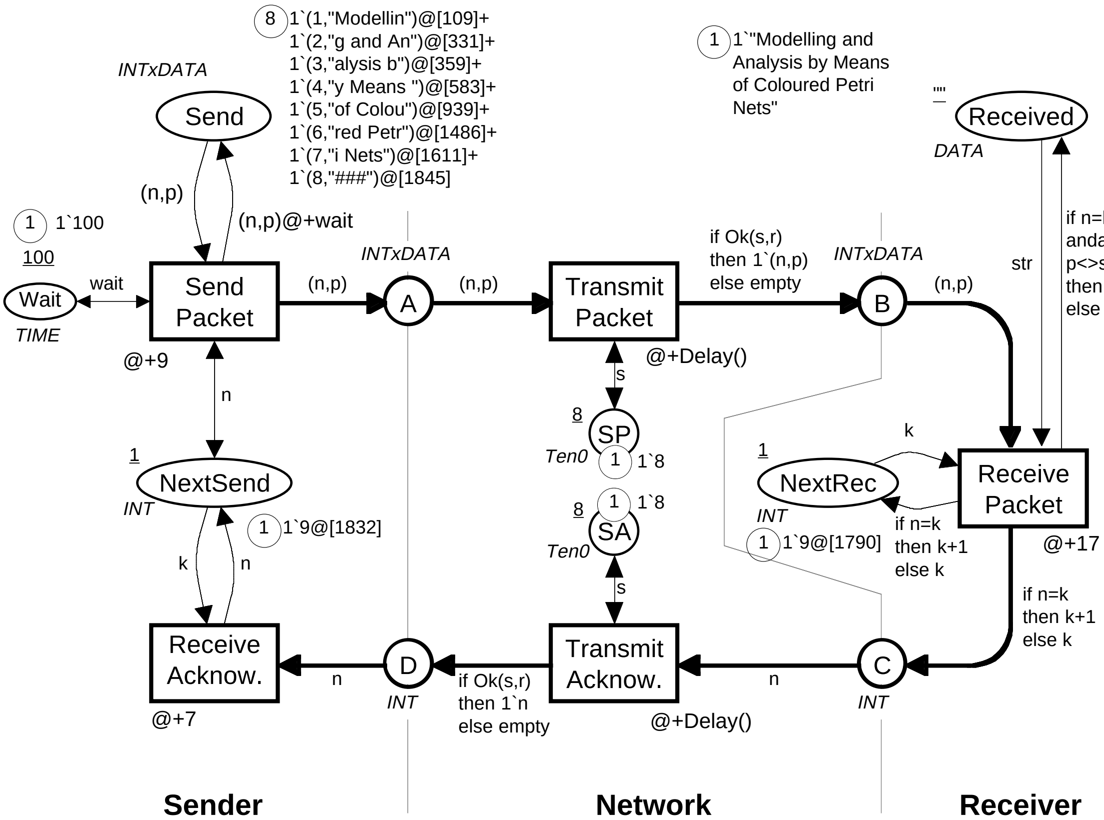


**Fig. 26.** Final marking of the timed CP-net

sions. It also increases the chance of overtaking and the chance that a *Receive Acknowledgement* operation is postponed, because the *Sender* process is engaged in a retransmission. A long delay means it may take too long before the *Sender* recognises that a packet or an acknowledgement has been lost. By making a number of simulations, with different token values at *Wait*, we can determine the optimal value for the retransmission delay.

To obtain reliable results we increase the number of packets from eight to one hundred. We also want to be able to describe more realistic success rates, and hence we change the types *Ten0* and *Ten1* to *Hun0* and *Hun1* (containing the integers in the intervals *0..100* and *1..100*, respectively). With this change, we would have one hundred different enabled bindings for transition *TransmitPacket* – for each token on place *A* with a ready time stamp. Instead of calculating all these bindings and then choosing one of them, we now use a predeclared function *ran'Hun1()* to draw a random element from the type *Hun1*. The new randomisation method gives the same result as the old one, but it is more effective, since we do not calculate bindings which we do not use.

With these modifications, we get the simulation results shown in Fig. 27. Each simulation took *2-10* seconds, and it was repeated ten times to obtain the mean value and the standard deviation of the time and steps used to transfer one hundred packets. The number of steps is a measure of the computational resources used by the protocol. It is proportional to the number of occurrences of *TransmitPacket* and *TransmitAcknowledgement* and hence proportional to the use of bandwidth on the network.

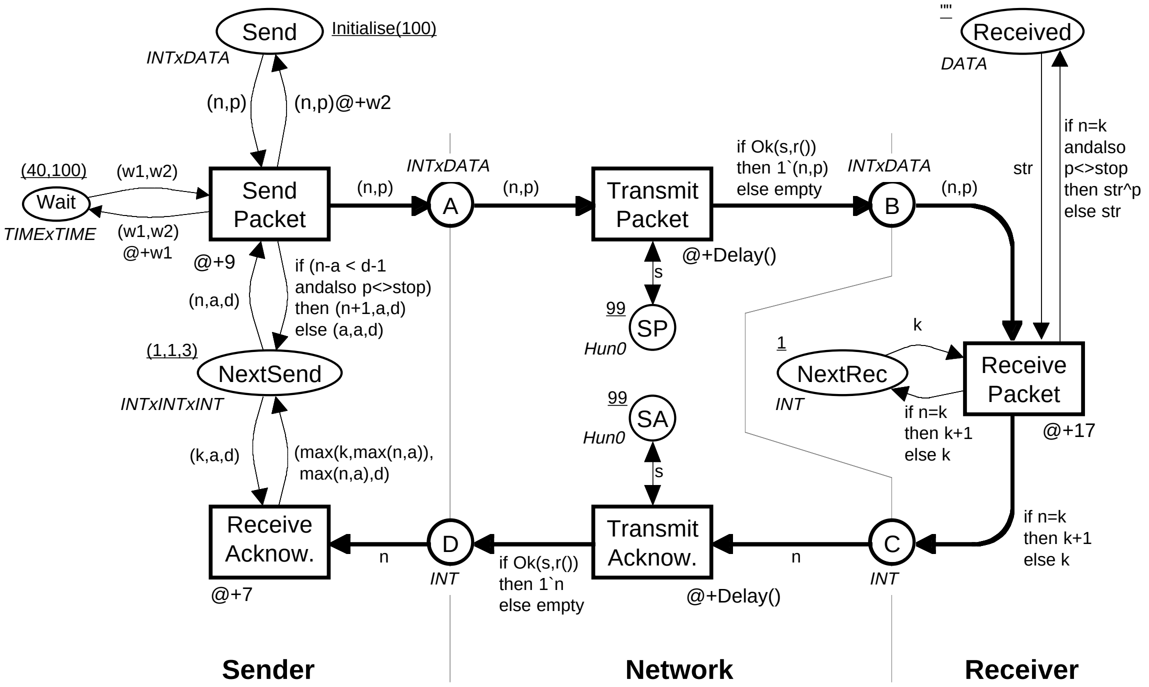
| Success Rate: 90 % | | |
|---|---|---|
| Wait | Time used | Number of steps |
| 20 | 13,092 ± 345 | 2,154 ± 54 |
| 50 | 13,931 ±275 | 1,260 ±22 |
| 100 | 14,850 ±510 | 871 ±18 |
| 200 | 16,983 ±1,275 | 565 ±17 |

| Success Rate: 99 % | | |
|---|---|---|
| Wait | Time used | Number of steps |
| 20 | 13,750 ±177 | 2,405 ±38 |
| 50 | 13,576 ±317 | 1,341 ±26 |
| 100 | 13,867 ±246 | 916 ±22 |
| 200 | 13,416 ±308 | 504 ±4 |

**Fig. 27.** Simulation results for the simple protocol

From the simulation results it can be seen that a poor network (with *10%* losses) has a clear trade-off between use of bandwidth and transmission time. If we make more frequent retransmissions, we use more bandwidth, but we also achieve faster transmission times. For a better network (with *1%* losses) the situation is different. Here, we do not obtain any gain in transmission time by making frequent retransmissions.

In the simple protocol, analysed above, we use a pessimistic strategy. This means that we keep transmitting a packet until we receive an acknowledgement for it. Now let us consider the more optimistic protocol shown in Fig. 28. In this protocol we assume that most packets will arrive without problems, and hence we transmit the



```
color INT = int timed;
color DATA = string;
color INTxDATA = product INT * DATA;
color INTxINTxINT = product INT * INT * INT;
var n, k, a, d : INT;
var p,str : DATA;
val stop = "###";

color Hun0 = int with 0..100;
color Hun1 = int with 1..100 declare ran;
var s : Hun0;
fun Ok(s:Hun0,r:Hun1) = (r<=s);
fun r()= ran'Hun1();

color NetDelay = int with 25..75 declare ran;
color TIMExTIME = product TIME * TIME timed;
fun Delay() = ran'NetDelay();
var w1,w2 : TIME;

local
   fun InitMark 0 = empty
     | InitMark n = 1`(n,"."^makestring(n)) + (InitMark (n-1));
in
   fun Initialise(n) = 1`(n,stop) + (InitMark (n-1))
end;
```

**Fig. 28.** A second and slightly more complex protocol

next packet without waiting for an acknowledgement of the packet which we have just sent. Now *NextSend* contains a triple of integers. The first element is the number *n* of the next packet to be sent. The second element is the number of the last acknowledgement *a* which the *Sender* has received. The third element is the window size, i.e., the maximal distance *d* which we allow between *n* and *a*. During a simulation the first two values will change, while the third remains constant.

Now let us consider the arc expression on the arc from *SendPacket* to *Next Send*. It determines the number of the next packet to be sent. As long as the distance between *n* and *a* is less than *d–1* and we have not reached the last packet (with *p = stop*) we increase *n* by one. Otherwise, *n* is set to *a* which is the first packet not yet known to be received by the *Receiver*. Whenever a new acknowledgement *n* is received the value of *a* is updated to *max(n,a)*. Simultaneously, we update the first element in *NextSend* from *k* to *max(k,max(n,a))*.

For the new protocol, place *Wait* has the type *TIMExTIME*. This means that the place now specifies two different time values. The first time value determines the delay between the sending of two different packets, while the second determines the delay between retransmissions of a packet.

The performance of the optimistic protocol is investigated in a similar way as the performance of the simple protocol. This gives the simulation results shown in Fig. 29. A comparison of Figs. 27 and 29 shows us that the performance of the optimistic protocol is much better than the performance of the pessimistic protocol. Hence, we could start a more detailed investigation of the optimistic protocol, e.g., to determine a good balance between the two delays specified by *Wait*. We could also try to optimise the window size specified by the third element in the *NextSend* token. However, such investigations are outside the scope of this paper.

| Success Rate: 90 %   Window Size: 3 | | |
|---|---|---|
| Wait | Time used | Number of steps |
| (20,20) | 7,192 ± 256 | 1,102 ±37 |
| (40,50) | 8.960 ±556 | 831 ±42 |
| (40,100) | 9,288 ±489 | 804 ±33 |
| (40,200) | 10,270 ±790 | 615 ±42 |

| Success Rate: 99 %   Window Size: 3 | | |
|---|---|---|
| Wait | Time used | Number of steps |
| (20,20) | 6,668 ±130 | 1,088 ±23 |
| (40,50) | 8,127 ±144 | 806 ±12 |
| (40,100) | 8,335 ±165 | 766 ±18 |
| (40,200) | 7,904 ±230 | 510 ±10 |

**Fig. 29.** Simulation results for the optimistic protocol

# 5 Hierarchical CP-nets

The basic idea behind hierarchical CP-nets is to allow the modeller to construct a large model by using a number of small CP-nets which are related to each other in a well-defined way. This is similar to the situation in which a programmer constructs a large program by means of a set of modules. Many CPN models consist of more than one hundred individual CP-nets with a total of many hundred places and transitions. Without hierarchical structuring facilities, such a model would have to be drawn as a single (very large) CP-net, and it would become totally incomprehensible.

In a hierarchical CP-net it is possible to relate a transition (and its surrounding arcs and places) to a separate CP-net – providing a more precise and detailed description of the activity represented by the transition. The idea is analogous to the hierarchy constructs found in many graphical description languages (e.g., data flow diagrams). It is also, in some respects, analogous to the module concepts found in many modern programming languages. At one level, we want to give a simple description of the modelled activity without having to consider internal details about how it is carried out. At another level, we want to specify the more detailed behaviour. Moreover, we want to be able to integrate the detailed specification with the more crude description – and this integration must be done in such a way that it becomes meaningful to speak about the behaviour of the combined system.

Now let us consider a hierarchical version of our protocol. The most *Abstract* description of the protocol is shown in Fig. 30. As before we have a *Sender* and a *Network*, but now we have two different *Receivers: RecNo1* and *RecNo2*. The CP-net in Fig. 30 has eight places and four transitions. Each of the transitions is marked with an HS-tag indicating that it is a **substitution transition** (HS ≈ Hierarchy + Substitution). The dashed boxes next to the HS-tags are called **hierarchy inscriptions** and they define the details of the substitutions.

The first line of each hierarchy inscription specifies the **subpage**, i.e., the CP-net that contains the detailed description of the activity represented by the corresponding substitution transition. In our example, we see that transition *Sender* has a subpage with the same name as itself, and so has transition *Network*. Transitions *RecNo1* and *RecNo2* both have a subpage called *Receiver*. During an execution of the CP-net, there will be two separate instances of the *Receiver* page, one for each substitution transition. Each of these **page instances** will have its own marking which is totally independent of the marking of the other page instance (in a similar way that procedure calls have private copies of local variables).

Now let us consider the subpages for our substitution transitions. They are shown in Fig. 31, and it can be seen that they are similar to the *Sender*, *Network* and *Receiver* parts of the simple protocol in Fig. 1. Each subpage has a number of places which are marked with an In-tag, Out-tag or I/O-tag. These places are called **port places** and they constitute the interface through which the subpage communicates with its surroundings. Through the input ports the subpage receives tokens from the surroundings. Analogously, the subpage delivers tokens to the surroundings through the output ports. A place with an I/O-tag is both an input port and an output port at the same time.

Substitution transition *Sender* in Fig. 30 has a single input place *D* and a single output place *A*. These places are called **socket places**. More precisely, *D* is an input socket for the *Sender* transition while *A* is an output socket. To specify the relationship between a substitution transition and its subpage, we must describe how the port places of the subpage are related to the socket places of the substitution transition. This is done by providing a **port assignment**. For the *Sender* subpage, we relate the input port *D* in Fig. 31 to the input socket *D* in Fig. 30. Analogously, we relate the output port *A* in Fig. 31 to the output socket *A* in Fig. 30. The relationship between ports and sockets are listed in the hierarchy inscriptions. However, to increase readability and brevity, we omit port assignments where the port and the socket have identical names. Hence we do not list the assignments *A–>A* and *D–>D* in the hierarchy inscription of *Sender*.

Next let us consider the *Network* subpage. Here, *A*, *C1*, and *C2* are input sockets, while *B1*, *B2*, and D are output sockets. The subpage has six port places, which each has the same name as the socket to which it is assigned (and hence we do not list the assignments in the hierarchy inscription). For the remaining two substitution transitions the situation is a bit more interesting. For *RecNo1* we have the port assignment *B1–>B*, *C1–>C*, and *Received1–>Received*. For *RecNo2* we have the port assignment *B2–>B*, *C2–>C*, and *Received2–>Received*.

When a port place is assigned to a socket place, the two places become identical. The port place and the socket place are just two different representations of a single conceptual place. In particular this means that the port and the socket places always have identical markings. When an input socket receives a token from the surroundings of the substitution transition that token also becomes available at the input port of the subpage, and hence the token can be used by the transitions on the subpage. Analogously, the subpage may produce tokens on an output port. Such tokens are also available at the corresponding output socket and hence they can be used by the

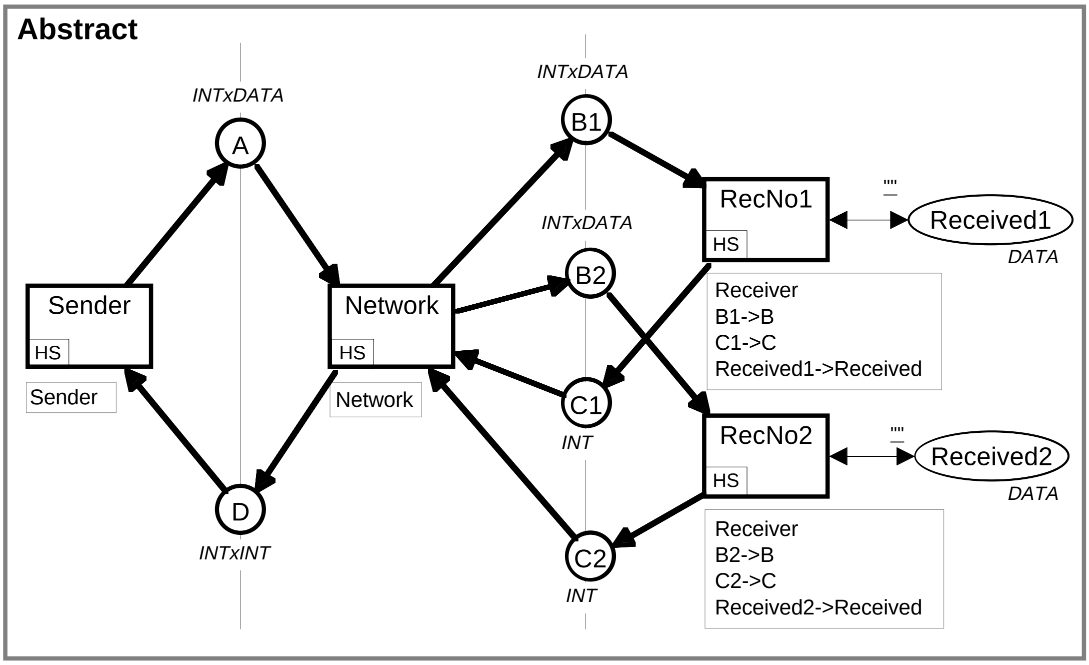


**Fig. 30.** Most abstract page in a hierarchical version of the simple protocol

surroundings of the substitution transition. In our example, we have three different representations of place *A*: one in the most *Abstract* net, one in the *Sender* subpage and one in the *Network* subpage. A similar remark applies to *B1, B2, C1, C2,* and *D*, while *Received1* and *Received2* only have two representations each.

Now let us consider the three subpages in Fig. 31 in some more detail. The basic idea is that the *Sender* sends messages which the *Network* broadcasts to the two *Receivers*. Analogously, the *Receivers* send acknowledgements which the *Network* transmits to the *Sender*.

The *Sender* subpage is similar to the *Sender* part in Fig. 1. The main difference is that *NextSend* now models two counters – one for each *Receiver*. Each acknowledgement is a pair where the first element specifies whether the acknowledgement came from *RecNo1* or *RecNo*2, while the second element contains the number of the next packet which the *Receiver* wants to get. When an acknowledgement *(rec,n)* is received, we update the counter for the corresponding *Receiver*. Based on our discussion in Sect. 3, we replace the old counter value *k* with the value *max(n,k)*. Since packets are sent by means of broadcasts, the *Sender* has to send the same packet to both *Receivers* – even in the case where the two counters at *NextSend* have different values. An obvious solution is to demand the *Sender* to use the minimum of the two counter values, i.e., to follow the acknowledgements of the most unlucky *Receiver*. However, instead we shall allow the *Sender* to use an arbitrary of the two counter values. For a *Network* with many losses this is less efficient. However, it allows one of the *Receivers* to get the entire message, even if the other *Receiver* stops to work, e.g., due to a crash.

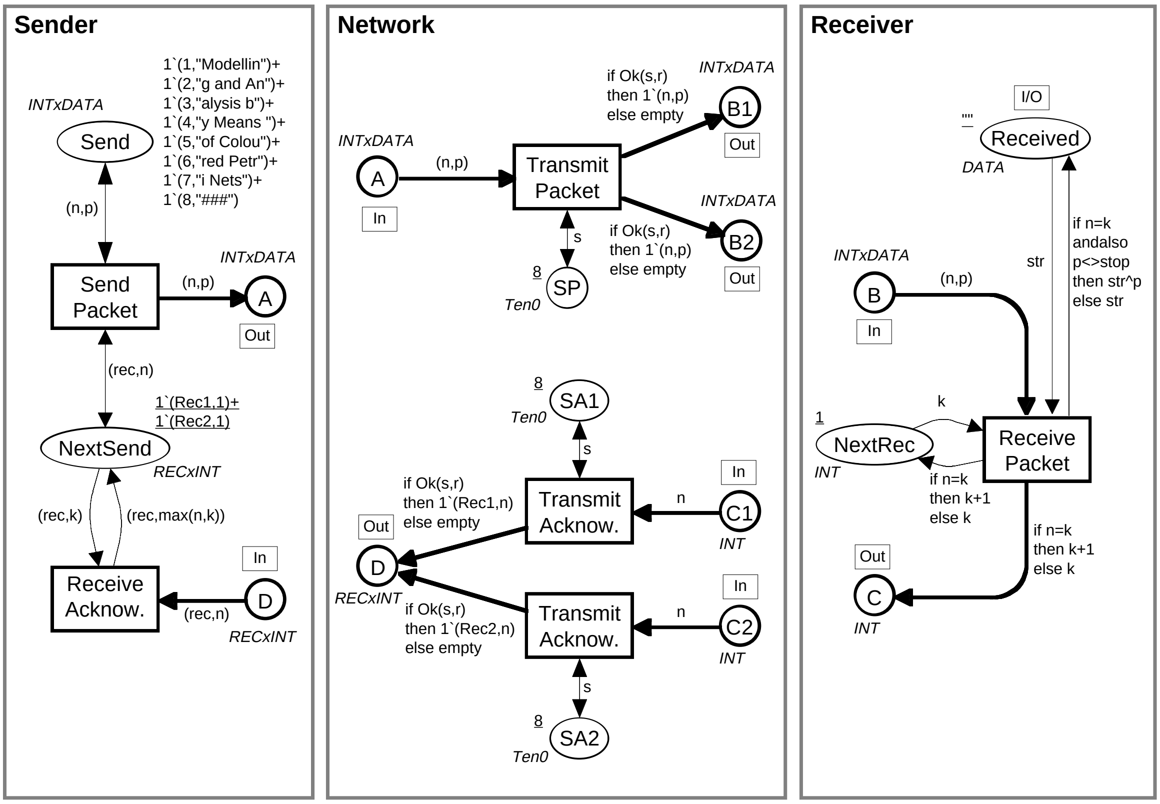


**Fig. 31.** Three subpages used by the substitution transitions in Fig. 30

The *Network* subpage is similar to the *Network* part in Fig. 1. However, again there are a few differences. *TransmitPacket* produces packets at two different output places *B1* and *B2*. The packets at *B1* are for *RecNo1*, while the packets at *B2* are for *RecNo2*. It should be noted that we use the same variable *r* to determine whether the packets for *B1* and *B2* are lost or not. This means that we model a broadcast in which it is guaranteed that all *Receivers* get the same packets. If we replace *r* with two different variables *r1* and *r2*, we get a broadcast where one *Receiver* may get a packet while the other does not. Transition *Transmit Acknowledgement* has now been split into two. The upper transition handles acknowledgements from *RecNo1* while the lower handles those from *RecNo2*. Both transitions add information specifying where the acknowledgement came from.

The *Receiver* subpage is totally identical to the *Receiver* part of Fig. 1. However, it should be noted that the *Receiver* page is used by two substitution transitions, *RecNo1* and *RecNo2*. As explained above, this means that we will have two instances of the subnet – during an execution. The two instances may have different markings and different enabling. Otherwise they will be identical.

In the protocol example, we only have two levels in the **page hierarchy** – the *Abstract* page in Fig. 30 and the three subpages in Fig. 31. However, in practice there are often up to ten different hierarchical levels. A subpage may contain substitution transitions and thus have its own subpages. It is often the case that a page both has ordinary transitions and substitution transitions, i.e., that some activities are described in full detail, while other activities are described in a more coarse way – deferring the detailed description to a subpage.

To give an overview of the relationship between the different pages in a CPN model, we use a page hierarchy graph as the one shown in Fig. 32. It contains a node for each page. An arc between two pages indicates that the latter is a subpage of the former, i.e., that the source page contains a substitution transition that uses the destination page as subpage. Each node is inscribed with a text that specifies the page name and the page number. Analogously, each arc may have a text that specifies the name of the substitution transition in question. As an example, Fig. 32 shows us that page *User_Top#2* has thirteen substitution transitions. One of these *U4* use *Call_Del#16* as subpage.

The page hierarchy graph shows the different pages and their hierarchical relationship. However, in the CPN tools it is also an active device by which the user can manipulate the pages. As an example, he can open a page by double-clicking the corresponding page node. He can delete a page by deleting the page node and he can remove the relationship between a substitution transition and its subpage by deleting the arc representing the relationship.

The page hierarchy graph in Fig. 32 is taken from a CPN model that describes a protocol for ISDN telephone networks. The big bracket indicates that each of the five pages to the right of the bracket is a subpage of all (or nearly all) of the twelve pages to the left of the bracket. Hence there are nearly sixty page instances, only for these five pages. We also see that some of the pages in the rightmost part of the page hierarchy graph have multiple instances. One of them has eight page instances, while two other pages have three instances each. Altogether the CPN model has forty-two pages with a total of approximately one hundred page instances. Page *ISDN#1* (in the upper left corner) has a small *Prime* next to it. This indicates that

*ISDN#1* is a **prime page**, i.e., a page on the most abstract level. A CPN model has a page instance for each prime page. For each substitution transition on a prime page we get a page instance of the corresponding subpage. If these page instances have substitution transitions, we get page instances for these, and so on – until we reach the bottom of the page hierarchy (which is demanded to be acyclic).

It can be shown that each hierarchical CP-net has a behavioural equivalent non-hierarchical CP-net. To obtain the non-hierarchical net, we simply replace each substitution transition (and its surrounding arcs) by a copy of its subpage – "gluing" each port place to the socket place to which it is assigned.

It should be noted that substitution transitions never become enabled and never occur. Substitution transitions work as a macro mechanism. They allow subpages to be conceptually inserted at the position of the substitution transitions – without doing an explicit insertion in the model. In Fig. 30 we have not provided any arc expressions for the arcs that surround the substitution transitions. These are not necessary for the simulation and state space analysis, since the substitution transitions never become enabled or occur. However, nevertheless they can be very useful. They can give the reader of a model a first impression of the functionality of the subpage. Moreover, the CPN simulator allows the modeller to specify that the subpage of one or more substitution transition shall be temporarily ignored. Then the transition behaves as an ordinary transition – and the guard and arc expressions become significant. By using this facility, and by changing the set of prime pages, it is easy to debug selected pages of a large hierarchical CPN model without having to "cut them out" of the model.

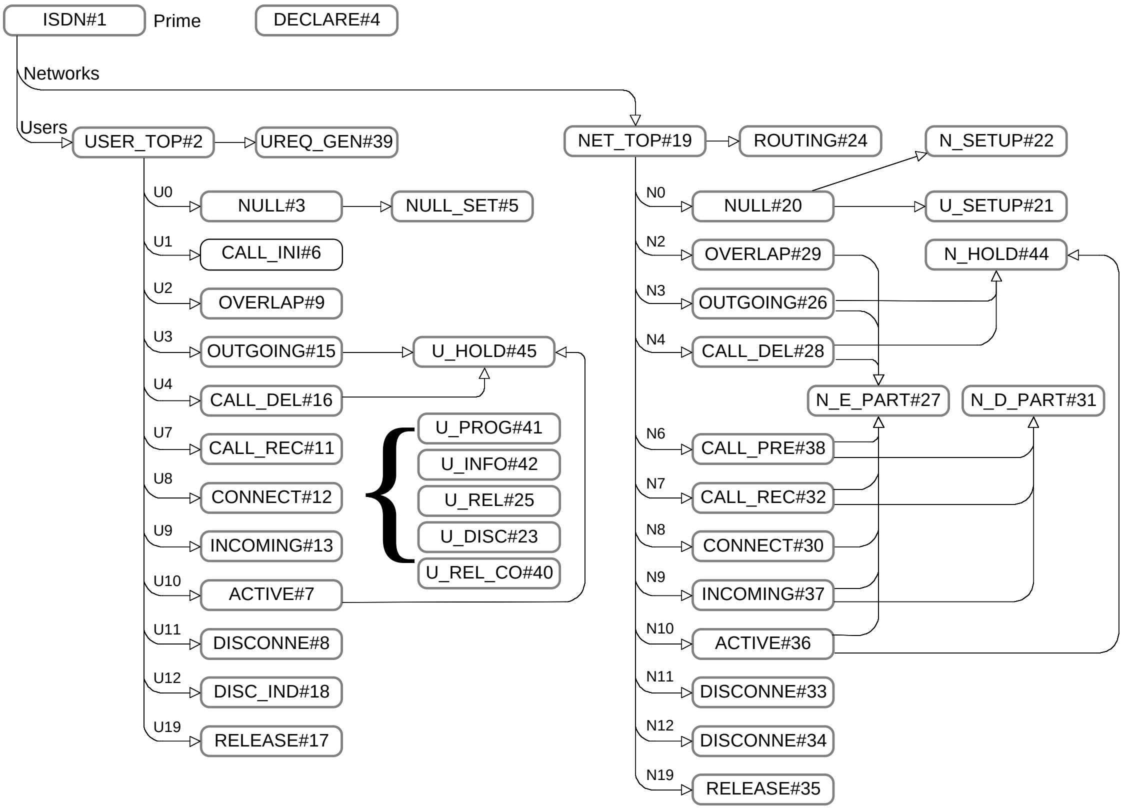


**Fig. 32.** Example of a more complex page hierarchy

It may be argued that it is adequate to force the modeller to create a one-to-one correspondence between the arc expressions of a substitution transition and the behaviour of the corresponding subpage – because this will make it easier to develop methods allowing modular analysis. We think it is very important to develop such incremental analysis methods, built upon behavioural equivalence between the different levels of description. However, we do not think that strict behavioural equivalence is the only interesting relationship between the different abstraction levels. As an example, there are protocol models in which the arc expressions and guards of a typical substitution transition specify the normal behaviour of an activity, while the subpage specifies the more complex behaviour which is necessary when time-outs, retransmissions and special services are added.

The protocol system in Figs. 30 and 31 was described in a top-down manner, but this does not necessarily mean that it was constructed in this way. It could just as well have been constructed bottom-up or (more likely) by mixing the two strategies.

The CPN editor supports the creation of hierarchical nets, and it is very easy to add new subpages – or rearrange the page hierarchy in other ways. When a page gets too many places and transitions, we can move some of them to a new subpage. This is done by a single editor operation. The user selects the nodes to be moved and invokes the Move to Subpage command. Then the editor:

- checks the legality of the selection (it must form a subnet bounded by transitions),
- creates the new page,
- moves the subnet to the new page,
- creates the port places by copying those places which were next to the selected subnet,
- calculates the port types (In, Out, or I/O),
- creates the corresponding port tags,
- constructs the necessary arcs between the port nodes and the selected subnet,
- prompts the user to create a new transition which becomes the substitution transition for the new subpage,
- draws the arcs surrounding the new transition,
- creates a hierarchy inscription for the new transition,
- updates the page hierarchy.

As may be seen, a lot of rather complex checks, calculations and manipulations are involved in the Move to Subpage command. However, almost all of these are automatically performed by the CPN editor. The user only selects the subnet, invokes the command and creates the new substitution transition. The rest of the work is done by the CPN editor. This is of course only possible because the CPN editor recognises a CPN diagram as a hierarchical CP-net, and not just as a mathematical graph or as a set of unrelated objects. Without this property the user would have to do all the work by means of the ordinary editing operations (which allow him to copy, move and create the necessary objects). This would be possible – but it would be much slower and much more error-prone.

There is also an editor command which turns an existing transition into a substitution transition – by relating it to an existing page. Again, most of the work is done by the editor. The user selects the transition and invokes the command. Then the editor:

- makes the hierarchy page active,
- prompts the user to select the desired subpage; when the mouse is moved over a page node it blinks, unless it is illegal (because selecting it would make the page hierarchy cyclic),
- waits until a blinking page node has been selected,
- tries to deduce the port assignment by means of a set of rules which looks at the port/socket names and the port/socket types (In, Out, or I/O),
- creates the hierarchy inscription with the name and number of the subpage and with those parts of the port assignment which could be automatically deduced,
- updates the page hierarchy.

Finally, there is an editor command that replaces a substitution transition by the entire content of its subpage. Also, this operation involves a lot of complex calculations and manipulations, but again all of them are done by the CPN editor. The user simply selects the substitution transition, invokes the command and uses a simple dialogue box to specify the details of the operation (e.g., whether the subpage shall be deleted when no other substitution transition uses it).

The three hierarchy commands described above can be invoked in any order. A user with a top-down approach would typically start by creating a page where each transition represents a rather complex activity. Then a subpage is created for each activity. The easiest way to do this is to use the Move to Subpage command. Then the subpage automatically gets the correct port places, i.e., the correct interface to the substitution transition. As the new subpages are modified, by adding places and transitions, the subpages may become so detailed that additional levels of subpages must be added. This is done in exactly the same way as the first level was created.

Hierarchical CP-nets also offer a concept known as **fusion places**. This allows the modeller to specify that a set of places are considered to be identical, i.e., they all represent a single conceptual place even though they are drawn as a number of individual places. When a token is added/removed at one of the places, an identical token will be added/removed at all the other places in the fusion set. From this description, it is easy to see that the relationship between the members of a fusion set is (in some respects) similar to the relationship between two places which are assigned to each other by a port assignment.

When all members of a fusion set belong to a single page and that page only has one page instance, place fusion is nothing other than a drawing convenience that allows the user to avoid too many crossing arcs. However, things become much more interesting when the members of a fusion set belong to several different pages or to a page that has several page instances. In that case, fusion sets allow the user to specify a behaviour which it may be cumbersome to describe without fusion.

There are three different kinds of fusion sets: **global** fusion sets are allowed to have members from many different pages, while **page** fusion sets and **instance** fusion sets only have members from a single page. The difference between the last two is the following. A page fusion unifies all the instances of its places (independently of the page instance at which they appear), and this means that the fusion set only has one “resulting place” which is “shared” by all instances of the corresponding page. In contrast, an instance fusion set only identifies place instances that belong to the *same* page instance, and this means that the fusion set has a "resulting place" for each page instance. The semantics of a global fusion set is analogous to that of a page fusion set – in the sense that there is only one "resulting place" (which is common for all instances of all the participating pages). To allow modular analysis of hierarchical CP-nets, global fusion sets should be used with care.

It is important to understand that the basic idea behind hierarchical CP-nets is to allow the modeller to construct a large model by combining a number of small CP-nets into a single model. This is similar to the situation in which a programmer constructs a large program from a set of modules and subroutines. However, the idea is different from those approaches that relate two or more separate subnets to each other – in order to compare their behaviour – but *without* combining them into a single model. Such approaches are analogous to program transformations, and the individual subnets are alternative descriptions of the same system.

As mentioned above, it is always possible to translate a hierarchical CP-net into a non-hierarchical CP-net – which in turn can be translated into a PT-net. This means that the theoretical modelling powers of these three classes of nets are the same. However, from a practical point of view, the three net classes have very different properties. To cope with large systems we need to develop strong structuring and abstraction concepts. The first very substantial step on this path was to replace low-level Petri nets with high-level nets. The second step is to introduce hierarchical nets. In terms of programming languages, the first step can be compared to the introduction of types – allowing the programmer to work with structured data elements instead of single bits. The second step may then be compared to the development of programming languages with subroutines and modules – allowing the programmer to construct a large model as a set of smaller models which are related to each other in a well-defined way. From a theoretical point of view, machine languages (or even Turing machines) are equivalent to the most powerful modern programming languages. From a practical point of view, this is of course not the case. One of the most important limitations that system developers face today is their own inability to cope with many details at the same time. In order to develop and analyse complex systems, they need structuring and abstraction concepts that allow them to work with a selected part of the model without being distracted by the low-level details of the remaining parts. Hierarchical nets provide the Petri net modeller with such abstraction mechanisms.

The concept of hierarchical nets is much younger than the concept of high-level nets, and this means that the hierarchy concepts are likely to undergo many improvements and refinements (in the same way that the first very simple concept of subroutines has undergone dramatic changes to become the procedure concept of modern programming languages). In other words, we do not claim that our current proposal will be the "final solution". However, we do think that it constitutes a good starting point for further research and practical experiences in the area of hierarchical nets.

The intention has been to make a set of hierarchy constructs which is general enough to be used with many different development methods and with many different analysis techniques. When new methods are developed, they will influence the definition of the hierarchy constructs in the same way that modern programming languages have been influenced by the progress in the areas of programming methodology and verification techniques.

To evaluate the strength of the existing hierarchy constructs, the reader is encouraged to consult some of the industrial CPN models described in Vol. 3 of [4]. They illustrate that substitution transitions and fusion places can be used in many different ways, and that they are quite efficient mechanisms to structure a large and complex CPN model.

# 6 Condensed State Spaces

In this section we illustrate how the symmetries inherent in many systems can be exploited to obtain a more succinct state space analysis.

To illustrate the basic idea, let us again consider the hierarchical protocol from Sect. 5 – the one with two *Receivers*. It should be obvious that the two receivers behave in a similar way. Hence, we can interchange them without influencing the behaviour of the system. To make this a bit more explicit, consider the two markings shown in Fig. 33. We do not list the markings of the places *Send*, *SP*, *SA1* and *SA2*,

| Marking $M_1$ | | |
|---|---|---|
| Sender | | |
| Next Send: | 1`(Rec1,2) + 1`(Rec2,3) | |
| A: | 1`(3,"alysis b") | |
| D: | 1`(Rec2,3) | |
| Receiver | Rec No 1 | Rec No 2 |
| Next Rec: | 1`2 | 1`3 |
| Received: | 1`"Modellin" | 1`"Modelling and An" |
| B: | 2`(2,"g and An") | empty |
| C: | empty | empty |

| Marking $M_2$ | | |
|---|---|---|
| Sender | | |
| Next Send: | 1`(Rec2,2) + 1`(Rec1,3) | |
| A: | 1`(3,"alysis b") | |
| D: | 1`(Rec1,3) | |
| Receiver | Rec No 1 | Rec No 2 |
| Next Rec: | 1`3 | 1`2 |
| Received: | 1`"Modelling and An" | 1`"Modellin " |
| B: | empty | 2`(2,"g and An") |
| C: | empty | empty |

**Fig. 33.** Two symmetrical markings

since they never change. Neither do we list the markings of the *Network* places *A*, *B1, B2, C1 C2, and D*. Due to the port assignments these places have markings that are identical to the markings of the corresponding places in the *Sender* and *Receiver* parts. An analogous remark applies to all the places in the *Abstract* part.

It is easy to see that marking $M_1$ can be mapped into $M_2$ (and vice versa) by performing a systematic interchange of the two receivers – including the token values at places *NextSend*, *A*, and *D*. Hence, we say that $M_1$ and $M_2$ are symmetric. However, the two markings not only look symmetric – they also behave in a symmetric way. To illustrate this let us consider the enabled binding elements which are shown in Fig. 34.

For each binding element $b_1$ which is enabled in $M_1$, we can find a symmetric binding element $b_2$ which is enabled for $M_2$. Again this is done by a systematic interchange of the two receivers. Moreover, the binding element $b_1$ will lead to a new marking $M_1'$ which is symmetric to the marking $M_2'$ to which $b_2$ will lead. By repeating this argument, we can see that each occurrence sequence starting in $M_1$ determines a symmetric occurrence sequence starting in $M_2$ (and vice versa). This

Enabled binding elements in $M_1$

Sender
(SendPacket, <rec = Rec1, n = 2, p = "g and An">)
(SendPacket, <rec = Rec2, n = 3, p = "alysis b">)
(ReceiveAcknowledgement, <rec = Rec2, k = 3, n = 3>)

Network
(TransmitPacket, <n = 3, p = "alysis b", s = 8, r = …>)

RecNo1
(ReceivePacket, <n = 2, p = "g and An", k = 2, str = " Modellin ">)

RecNo2
none

Enabled binding elements in $M_2$

Sender
(SendPacket, <rec = Rec2, n = 2, p = "g and An">)
(SendPacket, <rec = Rec1, n = 3, p = "alysis b">)
(ReceiveAcknowledgement, <rec = Rec1, k = 3, n = 3>)

Network
(TransmitPacket, <n = 3, p = "alysis b", s = 8, r = …>)

RecNo1
none

RecNo2
(ReceivePacket, <n = 2, p = "g and An", k = 2, str = " Modellin ">)

**Fig. 34.** Two symmetrical sets of enabled binding elements

means that $M_1$ and $M_2$ have symmetrical behaviours. If we know what can happen from one of these markings, we also know what can happen from the other, and hence it is sufficient to investigate one of the two markings.

Now let us consider the number of symmetry mappings, i.e., the number of ways in which a marking/binding element can be mapped into symmetrical markings/binding elements. There is a symmetry mapping for each possible permutation of the receivers, i.e., two mappings for two receivers, six mappings for three receivers, *24* mappings for four receivers, *120* mappings for five receivers, and so on. This means that a marking/binding element may have many symmetrical markings/binding elements. Hence, we may obtain a significant gain if it is sufficient to consider one of these.

The symmetries determine equivalence classes of states (markings) and equivalence classes of state changes (binding elements). They make it possible to construct a condensed state space where each node represents an equivalence class of states while each arc represents an equivalence class of state changes. Such a condensed state space is often much smaller than the ordinary state space and it is usually also faster to construct.

To illustrate the strength and limitations of the symmetry method, we construct state spaces for the hierarchical protocol with different numbers of receivers. To do this, we modify the CPN model as described in the beginning of Sect. 3. We reduce the number of packets to be sent, we limit the number of simultaneous packets on the *Network*, and we replace the boolean function *Ok* with a boolean variable. Fig. 35 shows the sizes of the different state spaces and the time used to construct them. In the lower part, we also show the reduction factor for nodes and arcs (i.e., how many times the condensed state space is smaller than the ordinary state space). The reduction factor should be compared to the number of symmetry mappings shown in the rightmost column.

For some of the ordinary state spaces the construction time is unknown. These state spaces are so big that it is impossible to construct them with the present version of our state space tool. However, it is possible to calculate their size from the condensed state spaces. The construction of condensed state spaces is still on a quite experimental level, and hence there is still plenty of room for improvement of the construction algorithm, in particular the efficiency of the ML function that determines whether two markings are equivalent to each other.

Although the condensed state spaces are smaller than the ordinary state spaces, they contain almost the same information – represented in a more condensed way. Hence, we do not lose analytical power. Condensed state spaces can be used to prove the same kind of behavioural properties as ordinary state spaces. The proof rules of condensed state spaces are similar to the proof rules of ordinary state spaces, but a bit more complicated since they have to deal with equivalence classes of markings and equivalence classes of binding elements.

An introduction to the theory of condensed state spaces can be found in [3], while a more detailed description can be found in Vol. 2 of [4]. The latter defines three different kinds of condensed state spaces. In state spaces with **equivalence classes**, the modeller directly specifies an equivalence relation for the set of markings and an equivalence relation for the set of binding elements. In state spaces with **symmetries** the equivalence relations are defined implicitly, by specifying a set of symmetry mappings, similar to those used for our protocol example. The symmetry mappings constitute an algebraic group, and this is sufficient to guarantee that they induce equivalence relations on the set of markings and on the set of binding elements. In state spaces with **permutation symmetries** the symmetry mappings are implicitly defined. This is done by specifying how the values of the individual types used in the CPN model can be permuted. The permutations used for a given type must be a subgroup of all permutations on that type. This guarantees that the permutations induce symmetry mappings which form an algebraic group. Each of the three kinds of condensed state spaces has its own set of proof rules and its own set of soundness criteria. A detailed description of these can be found in Vol. 2 of [4].

| Ordinary State Spaces | | | | | |
|---|---|---|---|---|---|
| Recs. | Limit | Packets | Nodes | Arcs | Time |
| 2 | 2 | 4 | 921 | 1,832 | 2 secs. |
| 2 | 3 | 3 | 14,025 | 44,826 | 2 mins. |
| 2 | 3 | 4 | 35,909 | 115,676 | 9 mins. |
| 3 | 3 | 4 | 22,317 | 64,684 | 4 mins. |
| 3 | 4 | 2 | 104,258 | 427,696 | 77 mins. |
| 4 | 4 | 2 | 39,617 | 154,752 | 14 mins. |
| 4 | 4 | 3 | 172,581 | 671,948 | 3 hours |
| 5 | 5 | 2 | 486,767 | 2,392,458 | --- |
| 6 | 6 | 2 | 5,917,145 | 35,068,448 | --- |
| 7 | 7 | 2 | 71,479,607 | 495,935,350 | --- |

| Condensed State Spaces | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Recs. | Limit | Packets | Nodes | | Arcs | | Time | Recs! |
| 2 | 2 | 4 | 477 | 1.9 | 924 | 2.0 | 3 secs. | 2 |
| 2 | 3 | 3 | 7,037 | 2.0 | 22,360 | 2.0 | 4 mins. | 2 |
| 2 | 3 | 4 | 17,991 | 2.0 | 57,743 | 2.0 | 23 mins. | 2 |
| 3 | 3 | 4 | 4,195 | 5.3 | 11,280 | 5.7 | 2 mins. | 6 |
| 3 | 4 | 2 | 18,253 | 5.7 | 72,929 | 5.9 | 31 mins. | 6 |
| 4 | 4 | 2 | 2,559 | 15.5 | 8,085 | 19.1 | 1 mins. | 24 |
| 4 | 4 | 3 | 9,888 | 17.5 | 32,963 | 20.4 | 8 mins. | 24 |
| 5 | 5 | 2 | 8,387 | 58.0 | 31,110 | 76.9 | 8 mins. | 120 |
| 6 | 6 | 2 | 24,122 | 245.3 | 101,240 | 346.4 | 1 hour | 720 |
| 7 | 7 | 2 | 62,625 | 1,141 | 290,018 | 1,710 | 10 hours | 5,040 |

**Fig. 35.** Sizes of state spaces and condensed state spaces (using interchanging of receivers)

Above, we have illustrated how state spaces for our protocol can be condensed by interchanging the different receivers. However, as shown in [6], the state spaces for our protocol may also be condensed using a different equivalence relation. Here, we consider two markings to be equivalent, if they are identical when we ignore the values of those packets and acknowledgements that are "old". A packet is considered to be old if it has a packet number which is less than the value at *NextRec*. It is easy to see that two different old packets have the same effect, since none of them match the current value of *NextRec*. Analogously, an acknowledgement is considered old if it contains a number which is less than or equal to the value at *NextSend*. This means that the acknowledgement will have no effect because we never decrease the value of *NextSend*. To illustrate the strength of this equivalence relation, Fig. 36 shows the sizes of some state spaces and the time used to construct them. All state spaces are for a single receiver. The two condensation techniques illustrated in Figs. 35 and 36 can be combined with each other, and with other condensation techniques. Theoretical and practical work with this integration is in progress, but outside the scope of this paper.

| Ordinary State Spaces | | | |
|---|---|---|---|
| Limit | Nodes | Arcs | Time |
| 1 | 33 | 44 | « 1 sec. |
| 2 | 293 | 764 | 1 sec. |
| 3 | 1,829 | 6,860 | 6 secs. |
| 4 | 9,025 | 43,124 | 56 secs. |
| 5 | 37,477 | 213,902 | 11 mins. |
| 6 | 136,107 | 891,830 | 2 hours |

| Condensed State Spaces | | | | | |
|---|---|---|---|---|---|
| Limit | Nodes | | Arcs | | Time |
| 1 | 33 | 1.0 | 44 | 1.0 | « 1 sec. |
| 2 | 155 | 1.9 | 383 | 2.0 | 1 sec. |
| 3 | 492 | 3.7 | 1,632 | 4.2 | 7 secs. |
| 4 | 1,260 | 7.1 | 5,019 | 8.6 | 36 secs. |
| 5 | 2,803 | 11.2 | 12,685 | 16.9 | 3 mins. |
| 6 | 5,635 | 24.2 | 28,044 | 31.8 | 9 mins. |
| 7 | 10,488 | --- | 56,203 | --- | 29 mins. |
| 8 | 18,366 | --- | 104,442 | --- | 81 mins. |
| 9 | 30,605 | --- | 182,754 | --- | 3 hours |
| 10 | 48,939 | --- | 304,445 | --- | 8 hours |

**Fig. 36.** Sizes of state spaces and condensed state spaces
(using equivalence relation for "old" packets and acknowledgements)

# 7 Conclusions

Below, we list a number of reasons for using CP-nets. We do not claim that CP-nets are superior to all other modelling languages. Such claims are, in our opinion, made far too often – and they nearly always turn out to be ridiculous. However, we do think that for some purposes CP-nets are extremely useful, and that, together with some of the other modelling languages, they should be a standard part of the repertoire of advanced system designers and system analysts.

*1. CP-nets have a graphical representation.* The graphical form is intuitively very appealing. It is very easy to understand and grasp – even for people who are not familiar with the details of CP-nets. This is due to the fact that CPN diagrams resemble many of the informal drawings that designers and engineers make while they construct and analyse a system.

*2. CP-nets have a well-defined semantics which unambiguously defines the behaviour of each CP-net.* The presence of the semantics makes it possible to implement simulators for CP-nets, and it also forms the foundation for the formal analysis methods.

*3. CP-nets are very general and can be used to describe a large variety of different systems.* The applications of CP-nets range from informal systems (such as the description of work processes) to formal systems (such as communication protocols). They also range from software systems (such as distributed algorithms) to hardware systems (such as VLSI chips). Excerpts from a number of industrial CPN models can be found in Vol. 3 of [4]. They cover a wide range of application areas.

*4. CP-nets have very few, but powerful, primitives.* The definition of CP-nets is rather short and it builds upon standard concepts which many system modellers already know from simple mathematics and programming languages. This means that it is relatively easy to learn to use CP-nets. However, the small number of primitives also means that it is possible to develop strong analysis methods.

*5. CP-nets have an explicit description of both states and actions.* This is in contrast to most system description languages which describe either the states or the actions – but not both. Using CP-nets, the reader may easily change the point of focus from states to actions, or vice versa.

*6. CP-nets have a semantics which builds upon true concurrency instead of interleaving.* In an interleaving semantics it is impossible to have two actions in the same step, and thus concurrency only means that the actions can occur after each other, in any order. A true-concurrency semantics is easier to work with – because it is closer to the way human beings think about concurrent actions.

*7. CP-nets offer hierarchical descriptions.* This means that we can construct a large CP-net by relating a number of small CP-nets to each other, in a well-defined way. The hierarchy constructs of CP-nets play a role similar to that of subroutines, procedures and modules of programming languages. The existence of hierarchical CP-nets makes it possible to model large systems in a manageable and modular way.

*8. CP-nets integrate the description of control and synchronisation with the description of data manipulation.* This means that on a single sheet of paper it can be seen what the environment, enabling conditions and effects of an action are. Many other graphical description languages work with graphs that only describe the environment of an action – while the detailed behaviour is specified separately (often by means of unstructured prose).

*9. CP-nets can be extended with a time concept.* This means that it is possible to use the same modelling language for the specification/validation of functional/ logical properties (such as absence of deadlock) and performance properties (such as throughput, bottlenecks and waiting times).

*10. CP-nets are stable towards minor changes of the modelled system.* This is proved by many practical experiences and it means that small modifications of the modelled system do not completely change the structure of the CP-net. In particular, it is possible to add a new sequential process without changing the net structure representing existing processes.

*11. CP-nets offer interactive simulations where the results are presented directly on the CPN diagram.* The simulation makes it possible to debug a large model while it is being constructed – analogously to a good programmer debugging the individual parts of a program as he finishes them. The data values of the moving tokens can be inspected.

*12. CP-nets have a number of formal analysis methods by which properties of CP-nets can be proved.* The two most important analysis methods are known as state spaces and place invariants. The first of these is described in this paper. The second is very similar to the use of invariants in program verification.

*13. CP-nets have an elaborated set of computer tools supporting their drawing, simulation and formal analysis.* This makes it possible to handle even large nets without drowning in details and without making trivial calculation errors. The existence of such computer tools is very important for the practical use of CP-nets.

## Acknowledgements

Many students and colleagues – in particular at Aarhus University and Meta Software – have influenced the development of CP-nets, their analysis methods and their tool support. The development has been supported by several grants from the Danish Natural Science Research Council. A more detailed description of individual contributions can be found in the prefaces of [4].

# References

[1] K. Jensen: *Coloured Petri Nets: A High-level Language for System Design and Analysis*. In: G. Rozenberg (ed.): Advances in Petri Nets 1990, Lecture Notes in Computer Science Vol. 483, Springer-Verlag 1991, 342– 416. Also in: K. Jensen and G. Rozenberg (eds.): High-level Petri Nets. Theory and Application, Springer-Verlag, 1991, 44 –122.

[2] K. Jensen: *An Introduction to the Theoretical Aspects of Coloured Petri Nets*. In: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): A Decade of Concurrency, Lecture Notes in Computer Science Vol. 803, Springer-Verlag 1994, 230–272.

[3] K. Jensen: *Condensed State Spaces for Symmetrical Coloured Petri Nets*. Formal Methods in System Design 9 (1996), Kluwer Academic Publishers, 7–40.

[4] K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Vol. 1: Basic Concepts, 1992. Vol. 2: Analysis Methods, 1994. Vol. 3: Practical Use, 1997. Monographs in Theoretical Computer Science, Springer-Verlag.

[5] K. Jensen, et al: *Design/CPN Manuals*. Meta Software Corporation and Department of Computer Science, University of Aarhus, Denmark. On-line version: http://www. daimi.aau.dk/designCPN/.

[6] J.B. Jørgensen, L.M. Kristensen: *Verification of Coloured Petri Nets Using State Spaces with Equivalence Classes*. In: B. Farwer, D. Moldt and M-O. Stehr (eds.): Proceedings of Workshop on Petri Nets in System Engineering (PNSE'97) Modelling, Verification, and Validation, Hamburg, Germany, Publication No. 205, Universität Hamburg, Fachberich Informatik, 1997, 20 –31.

[7] R. Milner, R. Harper, M. Tofte: *The Definition of Standard ML*. MIT Press, 1990.

[8] R. Milner, M. Tofte: *Commentary on Standard ML*. MIT Press, 1991.

[9] L. Paulson: *ML for the Working Programmer*. Cambridge University Press, 1991.

[10] *Petri Net WWW pages*. URL: http://www.daimi.aau.dk/PetriNets/.