# Chapter 5
# Hierarchical Coloured Petri Nets

This chapter shows how a CPN model can be organised as a set of modules, in a way similar to that in which programs are organised into modules. There are several reasons why modules are needed. Firstly, it is impractical to draw a CPN model of a large system as a single net, since it would become very large and inconvenient. Although the net can be printed on a set of separate sheets and glued together, it would be difficult to get an overview and it would be time-consuming to produce a nice layout. Secondly, the human modeller needs abstractions that make it possible to concentrate on only a few details at a time. CPN modules can be seen as black boxes, where modellers, when they desire, can forget about the details within modules. This makes it possible to work at different abstraction levels, and hence we shall also refer to CPN models with modules as *hierarchical CPN models*. Thirdly, there are often system components that are used repeatedly. It would be inefficient to model these components several times. Instead, a module can be defined once and used repeatedly. In this way there is only one description to read, and one description to modify when changes are necessary.

Section 5.1 introduces the concept of modules and their interfaces, and explains how to compose modules using substitution transitions. Section 5.2 introduces module instances, and Sect. 5.3 shows how modules can be parameterised. Section 5.4 shows how to parameterise a CPN model to make it easy to consider different configurations of the modelled system. Section 5.5 introduces the concept of fusion sets, and Sect. 5.6 shows how a hierarchical CPN model can be unfolded into a non-hierarchical CPN model.

## 5.1 Modules and Interfaces

To illustrate the use of modules, we revisit the CPN model of the protocol given in Sect. 2.4 and develop a hierarchical CPN model for this example protocol. A straightforward idea is to create a *module* for the sender, a module for the network, and a module for the receiver. These three modules could look as shown in Figs 5.1–

K. Jensen, L.M. Kristensen, *Coloured Petri Nets*, DOI 10.1007/b95112_5,

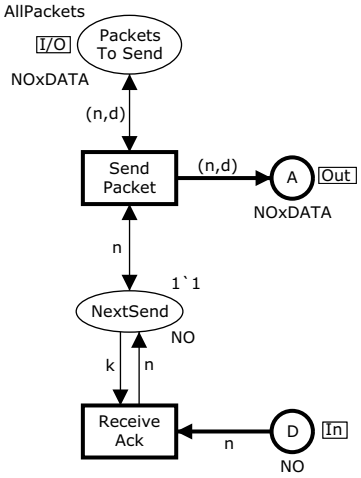


**Fig. 5.1** Sender module

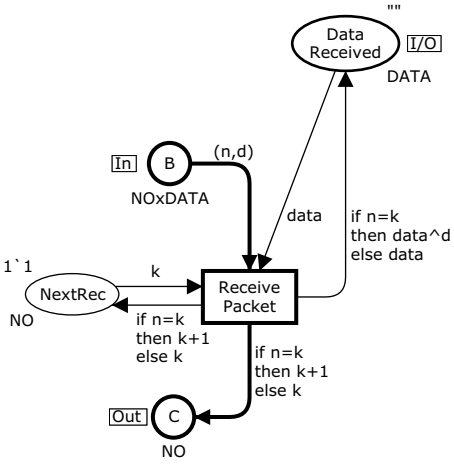


**Fig. 5.2** Receiver module

5.3. Intuitively, the protocol has been cut into three separate parts, where each part is identical to a subnet of Fig. 2.10.

The Sender module contains two transitions and four places. Place D is an *input port*, place A is an *output port*, and the place PacketsToSend is an *input/output port*. This means that A, D, and PacketsToSend constitute the *interface* through which the Sender module exchanges tokens with its environment (i.e., the other modules). The Sender module will import tokens via the input port D and export tokens via the output port A. An input/output port is a port through which a module can both import and export tokens. Port places can be recognised by rectangular port tags

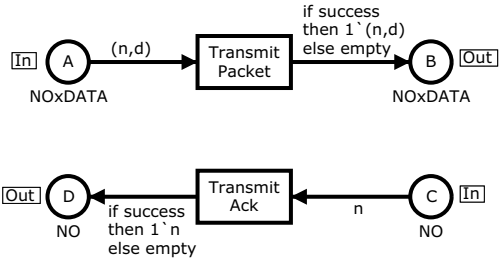


**Fig. 5.3** Network module

positioned next to them specifying whether the port place is an input, output, or input/output port. The place NextSend is an internal place, which is relevant only to the Sender module itself.

The Receiver module has an input port B, an output port C, an input/output port DataReceived, and an internal place NextRec. The Network module has two input ports, A and C, together with two output ports, B and D. The Network module has no internal places.

To tie the three modules together, we create the Protocol module, shown in Fig. 5.4. This represents a more abstract view of the entire protocol system. In the Protocol module, we can see that the Sender, Network, and Receiver exchange tokens with each other, via the places A, B, C, and D – but we cannot see the details of what the Sender, Network, and Receiver do.

The rectangular boxes with double-line borders in the Protocol module are *substitution transitions*. Each of them has a rectangular *substitution tag* positioned next to it. The substitution tag contains the name of a *submodule* which is related to the substitution transition. Intuitively, this means that the submodule presents a more detailed view of the behaviour represented by the substitution transition, in a way

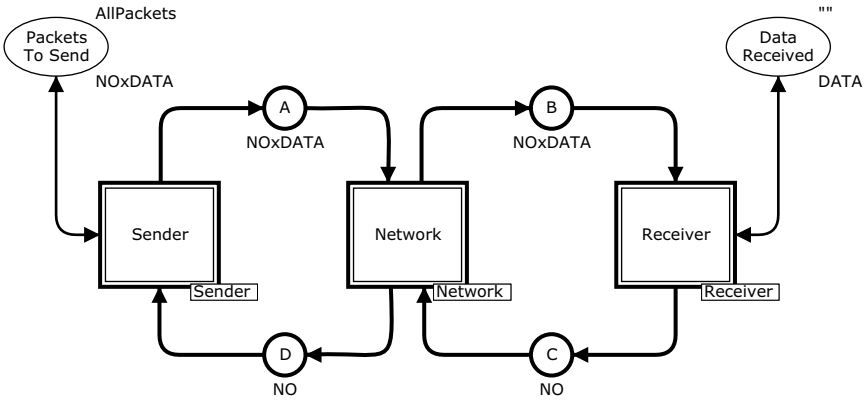


**Fig. 5.4** Protocol module: top-level module of the hierarchical protocol model

similar to that in which the implementation of a procedure provides a more detailed view of the effect of a procedure call. In Fig. 5.4, each substitution transition has the same name as its submodule, but this is not required in general.

The input places of substitution transitions are called *input sockets*, and the output places are called *output sockets*. This means that A is an output socket for the substitution transition Sender, and an input socket for the substitution transition Network. The place PacketsToSend is an *input/output socket* for the substitution transition Sender.

The socket places of a substitution transition constitute the interface of the substitution transition. To obtain a complete hierarchical model, it must be specified how the interface of each submodule is related to the interface of its substitution transition. This is done by means of a *port–socket relation*, which relates the port places of the submodule to the socket places of the substitution transition. Input ports are related to input sockets, output ports to output sockets, and input/output ports to input/output sockets. In Figs 5.1–5.4, each port has the same name as the socket to which it is related, but this is not required in general.

When a port and a socket are related, the two places constitute two different views of a single place. This means that related port and socket places always share the same marking and hence conceptually become a single *compound place*. Figures 5.5–5.7 show the marking of the Protocol, Sender, and Network modules after an occurrence of the transition SendPacket in the initial marking.

When the transition SendPacket occurs, it creates a token at the output port A in the Sender module (see Fig. 5.6). This port place is related to the output socket A of the substitution transition Sender in the Protocol module (see Fig. 5.5). Hence, the new token will also appear at place A in the Protocol module. This place is also

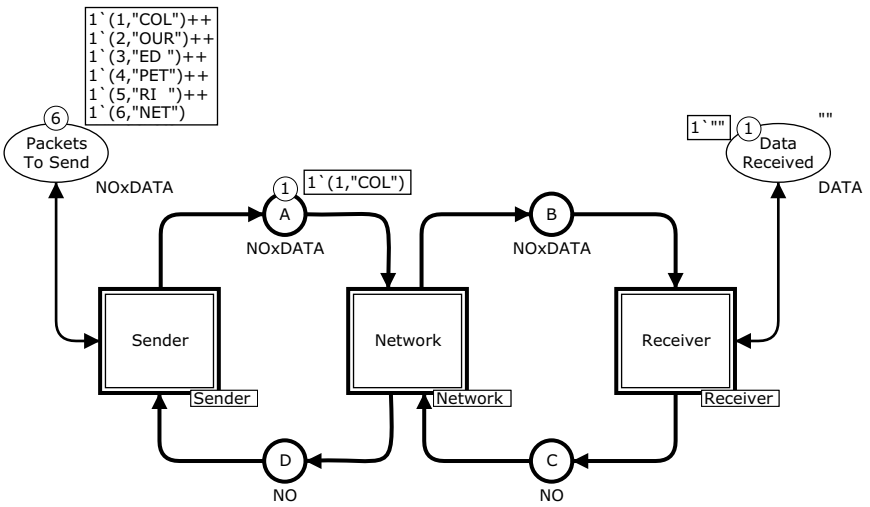


**Fig. 5.5** Marking of Protocol module, after occurrence of SendPacket

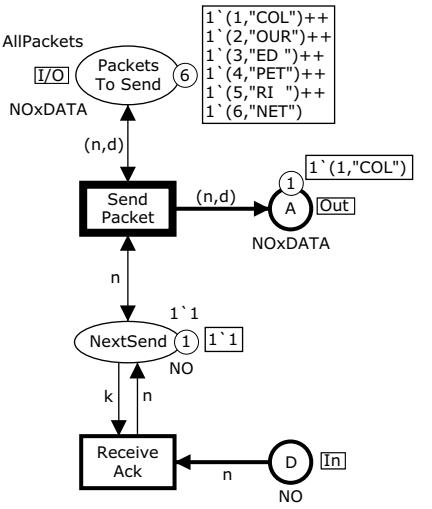


**Fig. 5.6** Marking of Sender module, after occurrence of SendPacket

an input socket for the substitution transition Network and has the port place A in the Network module (see Fig. 5.7) related to it. Hence, the new token also becomes available at the port place A of the Network module. In other words, the three places A in the Protocol, Sender, and Network modules are three different views of a single *compound place*, through which the modules can interchange tokens with each other. Similar remarks can be made about the places B, C, and D. The place D appears in the Protocol, Sender, and Network modules, while B and C appear in the Protocol, Network, and Receiver modules.

We have seen above that two related port and socket places constitute different views of a single place, and that this means that they always have the same marking. Obviously, this implies that they also need to have identical colour sets, and their initial marking expressions must evaluate to the same multiset of tokens. The only exception is that if a port place does not have an initial marking expression, then it

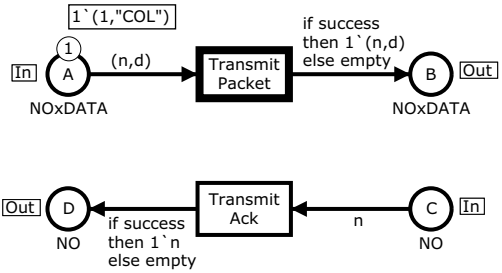


**Fig. 5.7** Marking of Network module, after occurrence of SendPacket

obtains its initial marking from the related socket place. We shall show later how this can be used to parameterise modules. It should be noted that substitution transitions do not have arc expressions or guards. It does not make sense to talk about the enabling and occurrence of a substitution transition. Instead, the substitution transition represents the compound behaviour of its submodule.

In the hierarchical model presented above there are only two levels of abstraction. The highest abstraction level is the Protocol module, and the lowest abstraction level is the Sender, Network, and Receiver modules. In general, there can be an arbitrary number of abstraction levels. As an example, a more detailed model of the protocol could be obtained by turning the transition SendPacket into a substitution transition having a submodule where the send operation is defined by a number of separate transitions, for example, one for the ordinary send operation and another for the resend operation. A larger system could also be envisioned in which the Protocol module is a submodule of one or more substitution transitions. CPN models of larger systems typically have up to 10 abstraction levels.

## 5.2 Module Instances and Hierarchy

Next let us take a closer look at the Network module in Fig. 5.3. It contains two transitions that have a very similar behaviour. However, the token colours involved are slightly different. The transition TransmitPacket deals with data packets of type `NOxDATA`, whereas the transition TransmitAck deals with acknowledgements of type `NO`. This means that we cannot immediately use the same submodule to represent the behaviour of TransmitPacket and TransmitAck, because a socket and its related port must have the same colour set. To overcome this problem, we use a union colour set in a way similar to that in Sect. 3.2. It can contain values from `NOxDATA` and values from `NO`, and is defined as follows:

```
colset PACKET = union Data:NOxDATA + Ack:NO;
```

This colour set is a union, and it uses two constructors `Data` and `Ack` to tell whether a data value of this colour set represents a data packet (such as `Data (1,"COL")`) or an acknowledgement packet (such as `Ack(2)`). Using the `PACKET` colour set, we can construct a modified version of the hierarchical protocol model consisting of the five modules shown in Figs 5.8–5.12.

As before, there are modules called Protocol, Sender, Network, and Receiver. For the Protocol, Sender, and Receiver modules no changes are made, except for those implied by the use of the colour set `PACKET` instead of `NOxDATA` and `NO`. The Network module now has two substitution transitions, each related to the new Transmit module shown in Fig. 5.12. The transition Transmit of the Transmit module transmits packets of type `PACKET`, i.e., both data packets and acknowledgements. The variable `p` is a variable of the colour set `PACKET`.

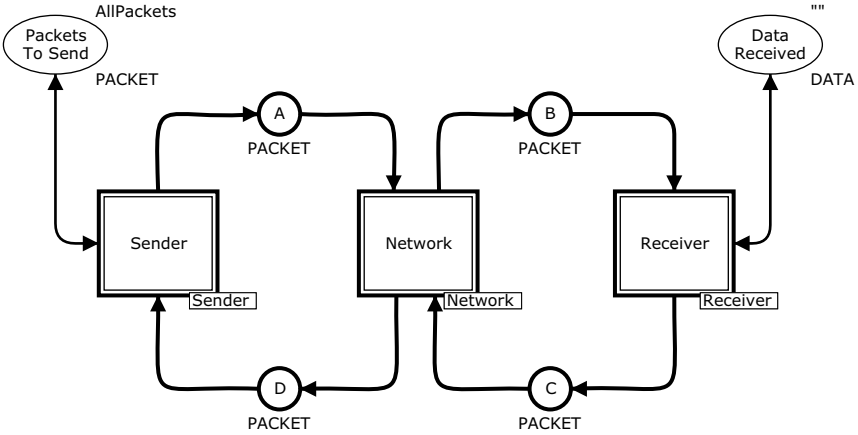


**Fig. 5.8** Modified Protocol module

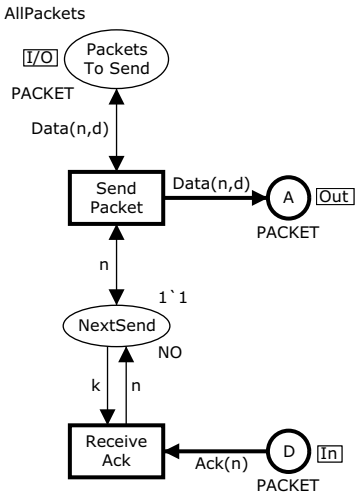


**Fig. 5.9** Modified Sender module

The Transmit module is used as a submodule of the substitution transitions TransmitData and TransmitAck in the Network module. This means that there will be two separate *instances* of the Transmit module – one for each of the two substitution transitions. For the instance of the Transmit module which is a submodule of the substitution transition TransmitData in Fig. 5.11 the port place IN is related to the socket place A, and the port place OUT is related to the socket place B. For the instance of the Transmit module which is a submodule of the substitution transition TransmitAck in Fig. 5.11, the port place IN is related to the socket place C, and

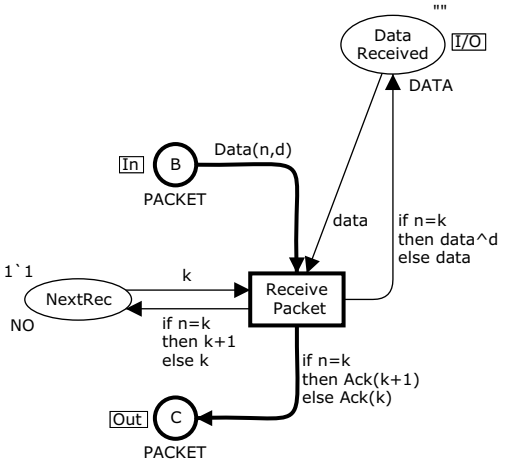


**Fig. 5.10** Modified Receiver module

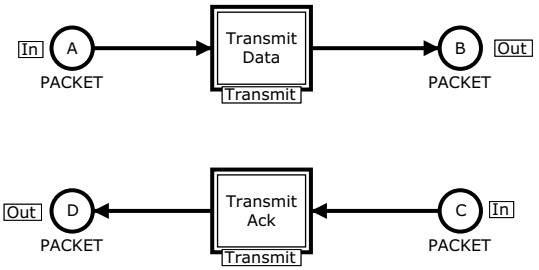


**Fig. 5.11** Modified Network module

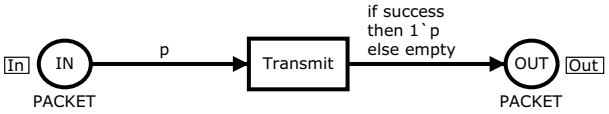


**Fig. 5.12** New Transmit module

the port place OUT is related to the socket place D. The places and transitions in a module instance are referred to as *place instances* and *transition instances*.

Each instance of a module has its own marking. This means that the marking of the instance of the Transmit module corresponding to the substitution transition TransmitData is independent of the marking of the instance corresponding to the substitution transition TransmitAck. Figure 5.13 shows a marking of the Protocol module with data packets on each of the socket places A and B, and acknowledgements on each of the places C and D. Figures 5.14 and 5.15 show the markings of the two instances of the Transmit module. It can be seen that each instance has its

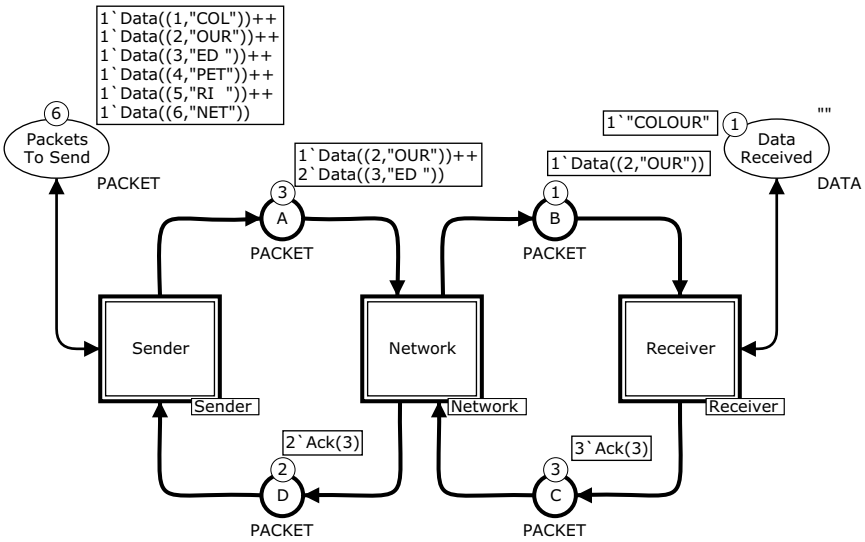


**Fig. 5.13** Example marking of the modified Protocol module

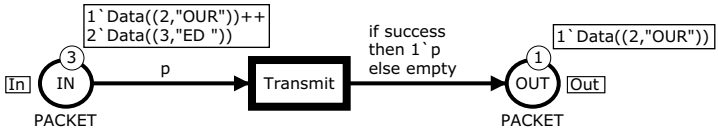


**Fig. 5.14** Marking of the Transmit module instance corresponding to TransmitData

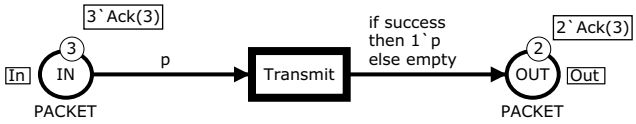


**Fig. 5.15** Marking of the Transmit module instance corresponding to TransmitAck

private marking, matching the tokens present on the socket places of the associated substitution transition.

The relationship between modules in a hierarchical model can be represented as a directed graph which has a node for each module and an arc for each substitution transition. For the CPN model in Figs 5.8–5.12, the *module hierarchy* looks as shown in Fig. 5.16. The names of the modules have been written inside the nodes, and the arcs have been labelled with the names of the substitution transitions. The node representing the Protocol module has no incoming arcs, it is a root of the module hierarchy and is called a *prime module*. This node has three outgoing arcs, corresponding to the three substitution transitions in the Protocol module (see Fig. 5.8). The arc from Protocol to Sender, labelled Sender, specifies that the substitution

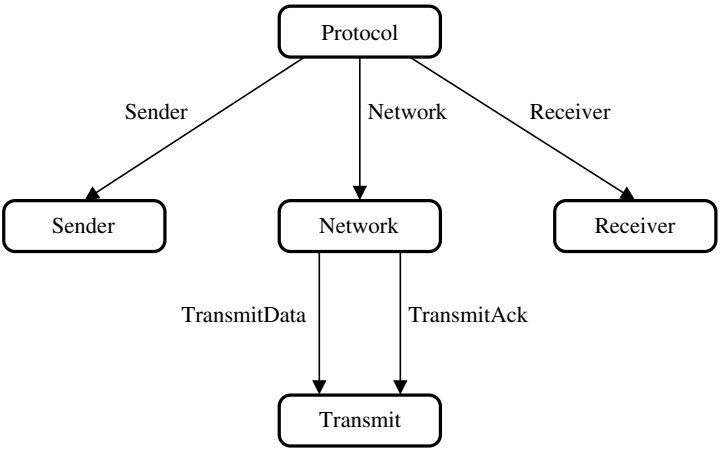


**Fig. 5.16** Module hierarchy for the hierarchical protocol model in Figs 5.8–5.12

transition Sender in the Protocol module has the Sender module as its related module. The modules that can be reached by following the arcs starting from a given module are said to be submodules of the latter module. The module hierarchy is required to be acyclic and hence it is not possible for a module to be a submodule of itself. This is required to ensure that there are only finitely many instances of each module when the modules are instantiated.

Before simulation of a hierarchical model is possible, the appropriate number of instances of each module must be instantiated and associated with substitution transitions. This means that the module hierarchy is unfolded into a directed tree called the *instance hierarchy*, where each node represents an instance of a module and the arcs represent substitution transitions. A tree is a directed graph where each node has at most one predecessor. Figure 5.17 shows the instance hierarchy obtained from the module hierarchy in Fig. 5.16. For the Transmit module which is the only module with more than one instance, we have written the instance number in parentheses following the module name. The first instance of the Transmit module is associated with the substitution transition TransmitData and the second instance is associated with the substitution transition TransmitAck. Instantiation of modules is handled fully automatically by CPN Tools, and the user is able to access the instance hierarchy via the index. Figure 5.18 shows how the module instances are organised in the index for the CPN model shown in Figs 5.8–5.12. A small triangle to the left of a module name indicates that it has submodules, and the submodules of the module are listed below it, and indented to the right. Each indentation level hence corresponds to a level in the instance hierarchy. A number in parentheses after a module name indicates that there are multiple instances, whereas a missing number indicates that there is only one instance of that module. The user can hide/show the submodules of a module in the index by clicking on the small triangle, and hence for large models it is possible to show only parts of the instance hierarchy.

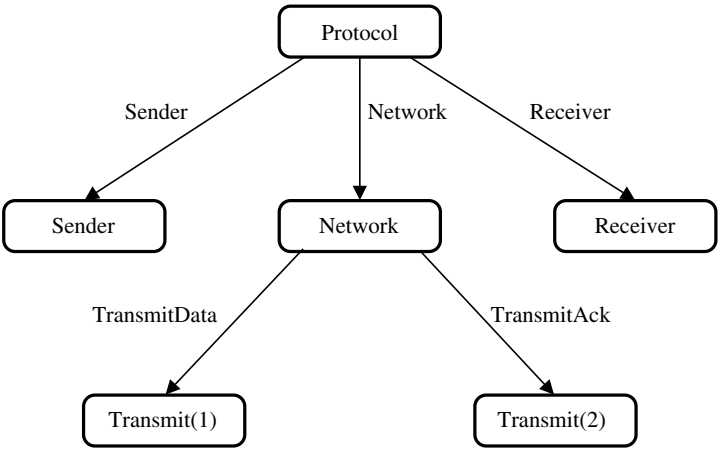


**Fig. 5.17** Instance hierarchy for the hierarchical protocol model in Figs 5.8–5.12

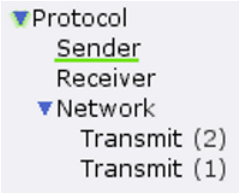


**Fig. 5.18** Index in CPN Tools for accessing module instances

It should be noted that instantiation of modules is done as the model is being constructed, i.e., prior to simulation of the CPN model. Hence, the number of instances of modules is fixed throughout the simulation of a hierarchical model, and it is not possible to dynamically instantiate new modules during the simulation.

## 5.3 Instance Folding and Module Parameterisation

As another example of a hierarchical model we shall consider a variant of the protocol model, with two receivers. This will be used to illustrate two modelling techniques that are often used in practice: a technique that allows us to achieve parameterisation of modules, and a technique that allows multiple instances of a module to be folded into a single instance of a module.

Figures 5.19–5.23 show a first hierarchical model (and a representative marking) of the protocol with two receivers; these receivers will be referred to as Receiver1 and Receiver2. The Transmit module is not shown, since it is identical to the one shown in Fig. 5.12. The model with two receivers was obtained by splitting the network places A, B, C, and D of the original model into eight places using A1, B1,

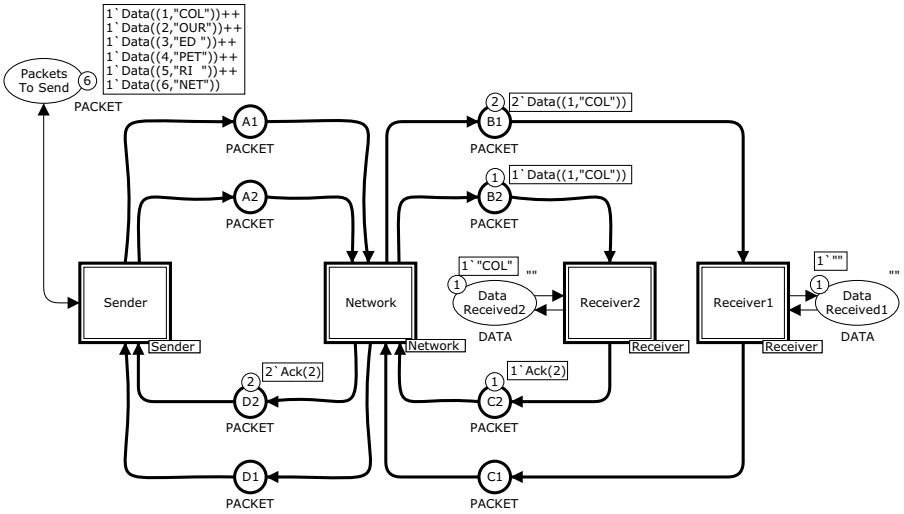


**Fig. 5.19** Protocol module for protocol with two receivers

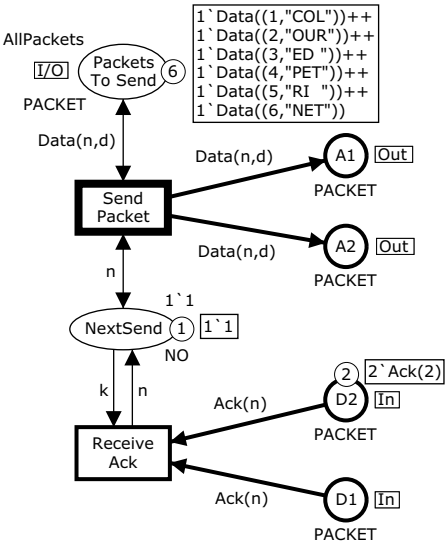


**Fig. 5.20** Sender module for protocol with two receivers

C1, and D1 for communication with Receiver1 and A2, B2, C2, and D2 for communication with Receiver2 (see Fig. 5.19). Furthermore, the substitution transitions Receiver1 and Receiver2 have been introduced, representing the two receivers and linked accordingly to the network places. The data received by Receiver1 is put on the place DataReceived1, and the data received by Receiver2 is put on place DataReceived2. The Network module (see Fig. 5.21) has been modified to take the

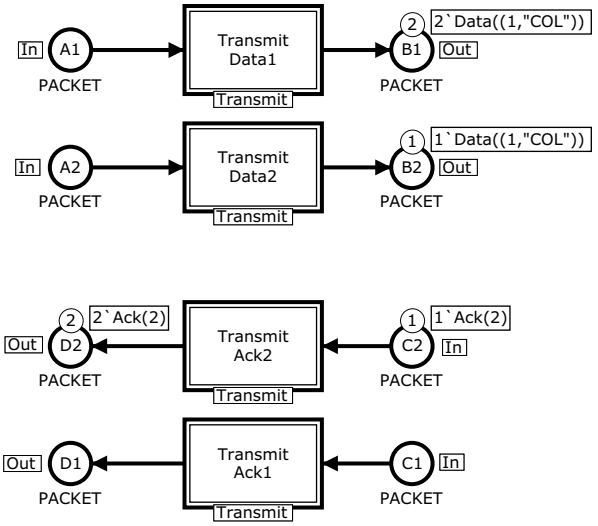


**Fig. 5.21** Network module for protocol with two receivers

eight network places into account. The Sender module (see Fig. 5.20) has been modified so that it sends (broadcasts) each data packet to both of the receivers. The transition ReceiveAck can only occur when the input places contain two identical acknowledgements – one from each of the two receivers. The Receiver module has not been modified. In the first instance of the Receiver module (corresponding to Receiver1), the port place B is related to the socket place B1, and in the second instance of the Receiver module (corresponding to Receiver2), the port place B is related to the socket place B2. The port place C is related to C1 and C2 in a similar fashion. Finally, the port place DataReceived is related to the socket places DataReceived1 and DataReceived2. This time there will be one instance of the Protocol, Sender, and Network modules, four instances of the Transmit module, and two instances of the Receiver module.

In the CPN model above, we have chosen to split the network places A, B, C, and D of the original model into eight places, using A1, B1, C1, and D1 for communication with Receiver1 and the places A2, B2, C2, and D2 for communication with Receiver2. This was done to be able to send data packets to both of the receivers and receive acknowledgements from both of the receivers. A similar effect can also be achieved by not splitting the network places, but instead revising the colour set used for packets to include a component specifying the intended receiver of the data packet and the receiver from which the acknowledgement originated. The modified colour set definitions are

```
colset RECV        = index Recv with 1..2;
colset RECVxPACKET = product RECV * PACKET;
```

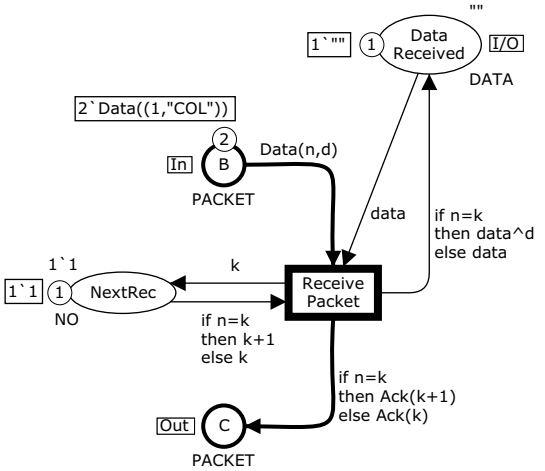


**Fig. 5.22** Receiver module instance for substitution transition Receiver1

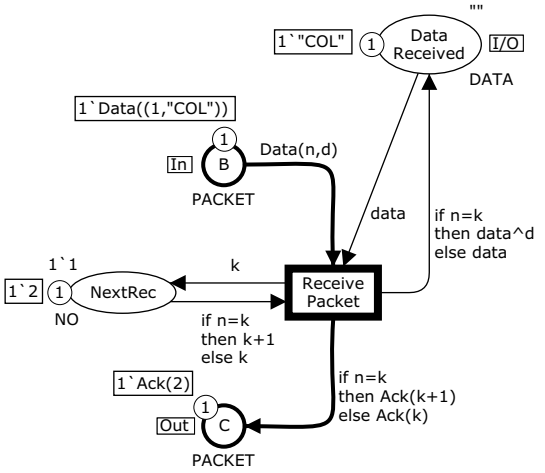


**Fig. 5.23** Receiver module instance for substitution transition Receiver2

The *index colour set* `RECV` is used for modelling the identity of the two receivers. This colour set contains two colours: `Recv(1)`, identifying the first receiver, and `Recv(2)` identifying the second receiver. The colour set `RECVxPACKET` is used to model the packets on the network. An example of a colour in this colour set is `(Recv(1),Data(1,"COL"))`, specifying a data packet `Data(1,"COL")` intended for the first receiver. Another example is `(Recv(2),Ack(2))` representing an acknowledgement `Ack(2)` originating from the second receiver. The modified Protocol module is shown in Fig. 5.24 with a representative marking.

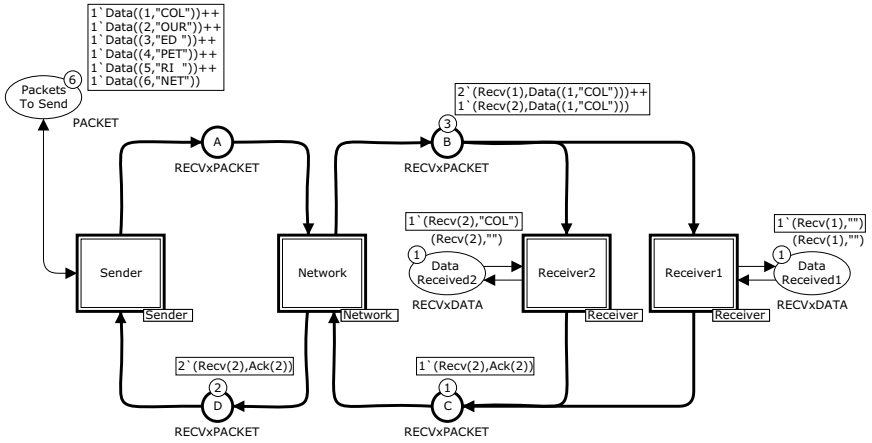


**Fig. 5.24** Protocol module for modified protocol with two receivers

The places A and B are used for sending data packets to the two receivers, and the places C and D are used for sending acknowledgement from the receivers. The place B has the marking

```
2‘(Recv(1),Data((1,"COL"))) ++
1‘(Recv(2),Data((1,"COL")))
```

which represents two identical data packets `Data(1,"COL")` in transit to the first receiver and one data packet `Data(1,"COL")` in transit to the second receiver. The first component of the pair specifies the receiver of the data packet. The identity of the receiver sending an acknowledgement is represented in a similar way in the tokens on the places C and D. Hence, by adding an extra component to the colour set of the network places, we have effectively *folded* the network places A1 and A2 into a single place A, and similarly for the other network places. Furthermore, we have modified the colour set of the places DataReceived1 and DataReceived2 to `RECVxDATA`, which is defined as

```
colset RECVxDATA = product RECV * DATA;
```

The idea is that the first component will specify the receiver identity and the second component will specify the data received. The initial marking of the two places has also been modified such that the initial marking of DataReceived1 is `(Recv(1),"")` and the initial marking of DataReceived2 is `(Recv(2),"")`. The purpose of this modification will be explained when the Receiver module is presented.

Figure 5.25 shows the modified Sender module. The expression on the arc from SendPacket to A now produces two tokens whenever the transition SendPacket occurs – one copy for each receiver. The expression on the arc from D to ReceiveAck requires two tokens to be present on D for ReceiveAck to be enabled – one from each receiver.

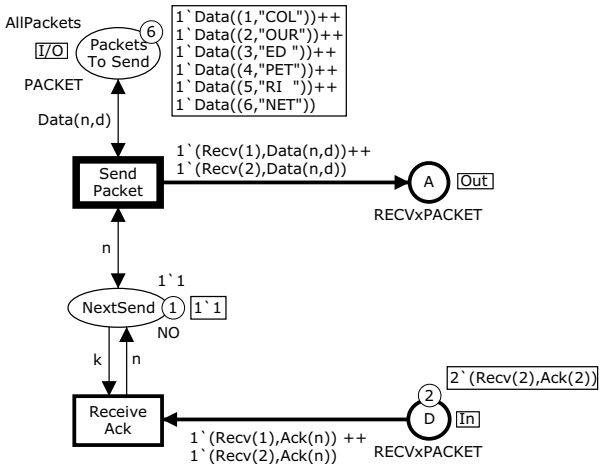


**Fig. 5.25** Sender module for modified protocol with two receivers

Figure 5.26 shows the modified Network module, where there are now only two substitution transitions: TransmitData, representing transmission of data packets to the two receivers, and TransmitAck, representing transmission of acknowledgements from both receivers. Both substitution transitions have the Transmit module shown in Fig 5.27 as an associated submodule. The Transmit module has been modified to take into account the modified colour sets of the port places IN and OUT. When the transition TransmitPacket occurs, the variable `pack`, of colour set `PACKET`, is bound to the packet (data packet or acknowledgement), and the variable `recv`, of colour set `RECV`, is bound to the identity of the receiver. Before, we had two instances of the Transmit module for transmission of data packets – now, there is just a single instance. The two instances have effectively been folded into a single instance of Transmit, and it is now the value bound to the variable `recv` that specifies whether the transmission is concerned with the first or the second receiver.

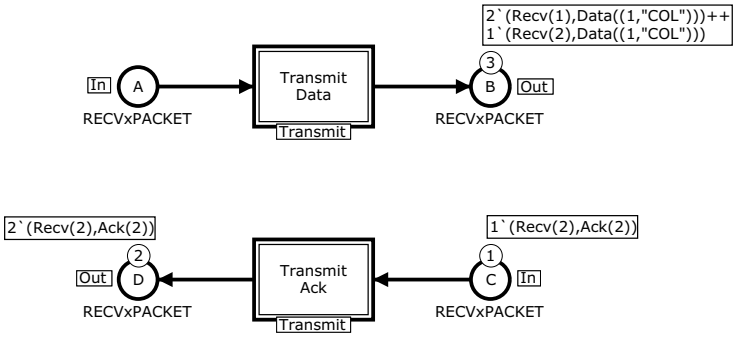


**Fig. 5.26** Network module for modified protocol with two receivers

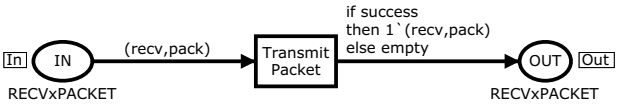


**Fig. 5.27** Transmit module for modified protocol with two receivers

Figures 5.28 and 5.29 show the two instances of the new Receiver module, where we have modified the colour set of the place DataReceived to `RECVxDATA`. For the substitution transition Receiver1 we relate the port place DataReceived to the socket place DataReceived1, and for the substitution transition Receiver2 we relate this port place to the socket place DataReceived2. The port place DataReceived does not have an initial marking expression, and therefore obtains its initial marking from the related socket place. The initial marking of the compound place consisting of the socket place DataReceived1 (see Fig. 5.24) and the port place DataReceived (see Fig. 5.28) is therefore determined by the initial marking expression of DataReceived1. Analogously, the initial marking of the compound place consisting of the socket place DataReceived2 (see Fig. 5.24) and the port place DataReceived (see Fig. 5.29) is determined by the initial marking expression of DataReceived2. This means that when the modules are instantiated, the module instance corresponding to the substitution transition Receiver1 will have a `(Recv(1),"")` token on the place DataReceived and the instance corresponding to the substitution transition Receiver2 will have a `(Recv(2),"")` token on the place DataReceived.

By using the variable `recv` on the arc from DataReceived to ReceivePacket, it is ensured that the first component in the token consumed from B matches the identity of the receiver. This ensures that it is only the ReceivePacket transition in the instance corresponding to Receiver1 that can consume the tokens with colour `(Recv(1),Data((1,"COL")))` and, similarly, it is only the ReceivePacket transition in the instance corresponding to Receiver2 that is able to consume the token `(Recv(2),Data((1,"COL")))`. The variable `recv` is also used on the output arc to C. This ensures that the acknowledgement is labelled with the correct receiver.

The above example demonstrates how a degree of *parameterisation* can be achieved by using port and socket places and then using the initial marking of the socket places to transfer parameters to the submodule (in this case the identity of the receiver). The example above has also demonstrated that it is possible to fold places and transitions in a CPN model and obtain a more compact model with fewer places and transitions. It should be noted that the two models presented in this section are behaviourally equivalent. The additional component in the tokens on the network places specifying the receiver in the second model effectively tells us whether the token was present on, for example, the place A1 or A2 in the original model. A similar observation applies to the other network places. It is now the binding of the variable `recv` of the transition TransmitPacket that specifies which earlier instance the token corresponds to.

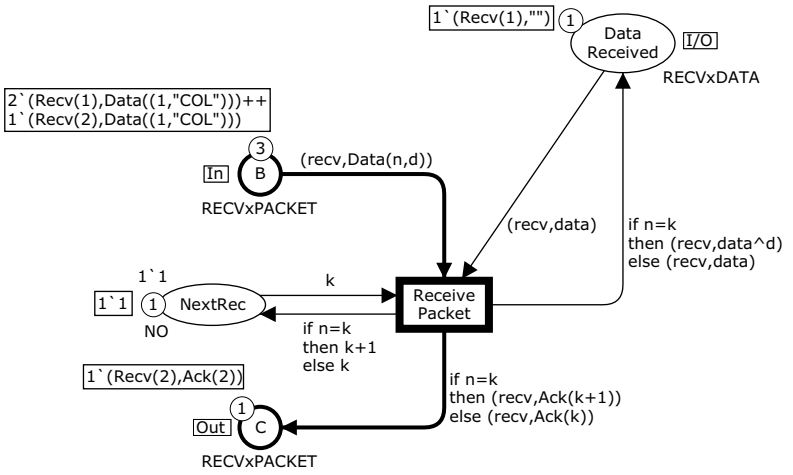


**Fig. 5.28** Receiver module instance for Receiver1 in modified protocol with two receivers

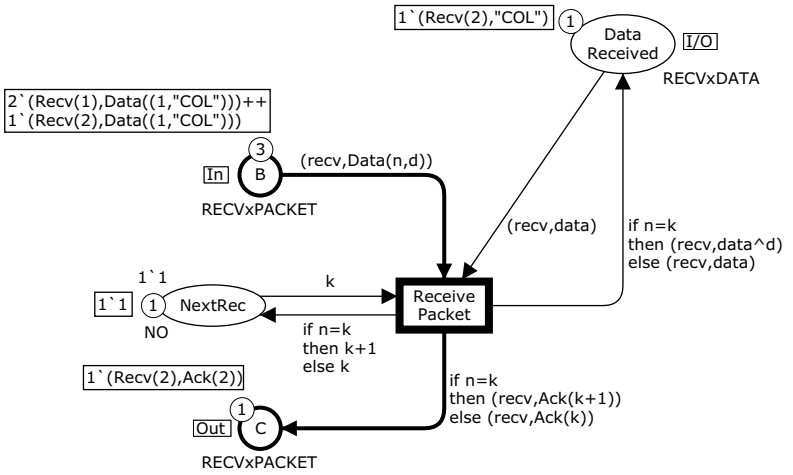


**Fig. 5.29** Receiver module instance for Receiver2 in modified protocol with two receivers

## 5.4 Model Parameterisation

The model presented in Figs 5.24–5.29 in the previous section is more compact than the first model with two receivers. A weakness of both models, however, is that it is cumbersome to add additional receivers. As an example, if a third receiver is to be added then we need to add a substitution transition Receiver3 in the Protocol module (see Fig. 5.24) and associate a third instance of the Receiver module with this new substitution transition. Also, the Sender module (see Fig. 5.25) must be modified

such that it now produces three tokens on place A and consumes three appropriate acknowledgements from place D.

We shall now present a model with multiple receivers where it is not required to make changes to the net structure and inscriptions when the number of receivers is changed. The basic idea is to fold the instances of the Receiver module in a way similar to that for the network places and the Transmit instances in the previous section. To achieve this, we revise the definition of the colour set RECV and add one new colour set RECVxNO as follows:

```
val NoRecvs = 3;

colset RECV      = index Recv with 1..NoRecvs;
colset RECVxNO   = product RECV * NO;
```

We have introduced a symbolic constant NoRecvs, which determines the number of receivers. This constant is used in the definition of the colour set RECV such that the colours in this colour set match the number of receivers. In the above case, RECV contains the colours Recv(1), Recv(2), and Recv(3).

Figure 5.30 shows the Protocol module in the initial marking. There is now a single substitution transition Receiver representing the receivers. The initial marking of DataReceived is given by the expression

```
AllRecvs ""
```

which evaluates to the following multiset of tokens:

```
1`(Recv(1),"") ++ 1`(Recv(2),"") ++ 1`(Recv(3),"")
```

This marking specifies that all receivers have initially received the empty string "". The function AllRecvs is defined as follows (the functions RECV.all and List.map will be explained shortly):

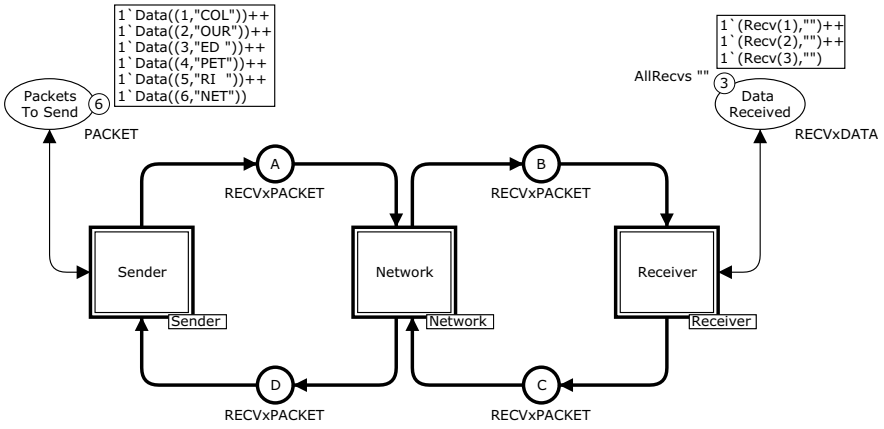


**Fig. 5.30** Protocol module for protocol with multiple receivers

```
fun AllRecvs v = List.map
                   (fn recv => (recv,v)) (RECV.all());
```

We have used the predefined colour set function `RECV.all`, which takes a unit `()` as an argument and returns the list representation of a multiset containing one appearance of each of the colours in the colour set `RECV`, i.e., it returns the multiset

```
1`Recv(1) ++ 1`Recv(2) ++ 1`Recv(3)
```

represented as the following list:

```
[Recv(1),Recv(2),Recv(3)]
```

We have also used the the curried predefined Standard ML function `List.map`, which takes a function and a list as arguments and applies the function to each element of the list. In this case, the function is `fn recv => (recv,v)`, which, given a receiver `recv` constructs the pair `(recv,v)` where `v` is the argument provided to the function `AllRecvs`. In this case it results in the list

```
[(Recv(1),""),(Recv(2),""),(Recv(3),"")]
```

representing the following multiset, which becomes the initial marking of the place PacketsToSend:

```
1`(Recv(1),"") ++ 1`(Recv(2),"") ++ 1`(Recv(3),"")
```

If the value of `NoRecv` is changed to 4, the initial marking expression of DataReceived will evaluate to the following multiset:

```
1`(Recv(1),"") ++ 1`(Recv(2),"") ++
1`(Recv(3),"") ++ 1`(Recv(4),"")
```

hence the initialisation expression of DataReceived does not have to be modified when the number of receivers is changed. It is sufficient to change the declaration of `NoRecv`.

In the above definition of the function `AllRecvs`, we have exploited the fact that multisets in CPN Tools are represented using lists, i.e., a multiset is represented as a list of the elements in the multiset where an element appears as many times in the list as its coefficient in the multiset specifies. This means that we can apply list operations (such as `List.map`) directly to the elements of a multiset and there is no need to convert between list and multiset representations.

The Network and Transmit modules do not need to be changed, so we shall present only the Sender and Receiver modules below. The Receiver module is shown in Fig. 5.31. The colour set of the place NextRec has been changed to `RECVxNO`, and the idea is to use the first component to identify the receiver, and the second component to specify the data packet expected next by the receiver identified in the first component. The initial marking expression of `NextRec` uses the function `AllRecvs` with the argument `1` to obtain the initial marking

```
1`(Recv(1),1) ++ 1`(Recv(2),1) ++ 1`(Recv(3),1)
```

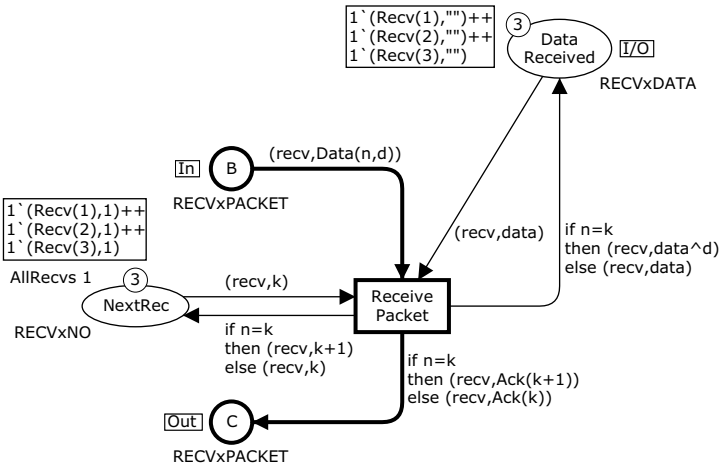


**Fig. 5.31** Receiver module for protocol with multiple receivers

which specifies that all receivers initially expect the data packet with sequence number 1. The variable `recv` is used on all input and output arcs of ReceivePacket to ensure that the update of the expected sequence number on the place NextRec, the update of the data on the place DataReceived, and the acknowledgement produced on place C all correspond to the intended receiver of the data packet removed from place B.

Figure 5.32 shows the Sender module. The expressions on the arcs connected to the two network places A and D have been modified to use the function `AllRecvs`, which, for a given packet, produces a multiset over `RECVxPACKET` with a packet for each receiver. As an example, the expression

```
AllRecvs (Data(1,"COL"))
```

evaluates to the multiset

```
1`(Recv(1),Data(1,"COL")) ++
1`(Recv(2),Data(1,"COL")) ++
1`(Recv(3),Data(1,"COL"))
```

and the expression

```
AllRecvs (Ack(2))
```

evaluates to the multiset

```
1`(Recv(1),Ack(2)) ++
1`(Recv(2),Ack(2)) ++
1`(Recv(3),Ack(2))
```

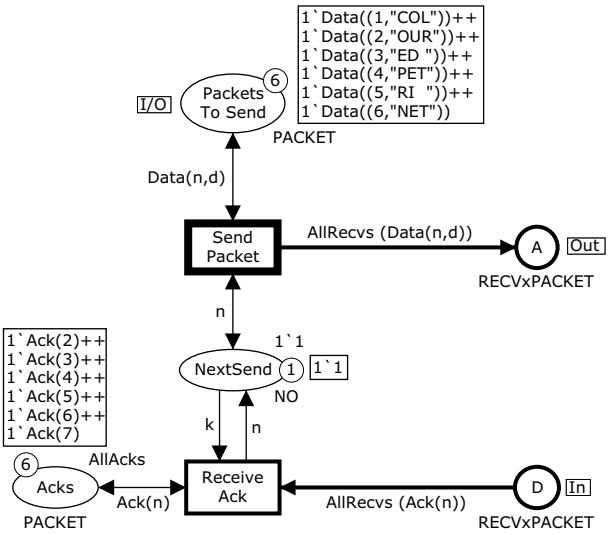


**Fig. 5.32** Sender module for protocol with multiple receivers

The place Acks contains the set of possible acknowledgements that can be received. The constant used as the initial marking of this place is defined as

```
val AllAcks = List.map
                (fn Data(n,_) => Ack(n+1)) AllPackets;
```

The definition of `AllAcks` uses the function `List.map`. In this case the first argument is the function `fn Data(n,_) => Ack(n+1)`, which, for a given data packet with sequence number `n`, constructs the corresponding acknowledgement, which has sequence number `n+1`. The second argument is the list of data packets to be transmitted.

The place Acks has been introduced to make it possible to bind the variable `n` of the transition ReceiveAck. The variable `n` can no longer be bound from the arc expression on the input arc from D, since the arc expression now uses a function, which means that it no longer qualifies as a pattern. Note that ReceiveAck is only enabled for a given value bound to `n` when place D contains an acknowledgement with sequence number `n` from all receivers.

## 5.5 Fusion Sets

It has been shown above how modules can exchange tokens via port and socket places. It is also possible for modules to exchange tokens via *fusion sets*. Fusion sets allow places in different modules to be glued together into one *compound place* across the hierarchical structure of the model. Fusion sets are in some sense similar

to the global variables known in many programming languages and should therefore be used with care. However, there are many cases where fusion sets can be convenient and below we give three typical examples of how fusion sets are used.

As a first example, consider the hierarchical model of the protocol with multiple receivers created in the previous section. Suppose now that we are interested in collecting the lost data packets and acknowledgements on a single place in the CPN model. The first step is to add a place PacketsLost to the Transmit module as shown in Fig. 5.33, and collect the tokens corresponding to the lost packets on this place.

As explained above, there are two instances of the Transmit module, and a separate marking for each of these instances. This implies that there are two instances of the place PacketsLost and that each of these has its own marking. To fold these two place instances into a single place, we use a *fusion set*. The places that are members of a fusion set are called *fusion places* and represent a single *compound place*, in a way similar to that for a related port and socket place. This means that all instances of all places in a fusion set always share the same marking and that they must have identical colour sets and initial markings. In Fig. 5.34, PacketsLost belongs to a fusion set called Lost. This can be seen from the rectangular fusion tag positioned next to the place.

Figures 5.35 and 5.36 show the two instances of Transmit in a representative marking. It can be seen that the two instances of the port place IN have different markings and the same is the case for the two instances of the port place OUT.

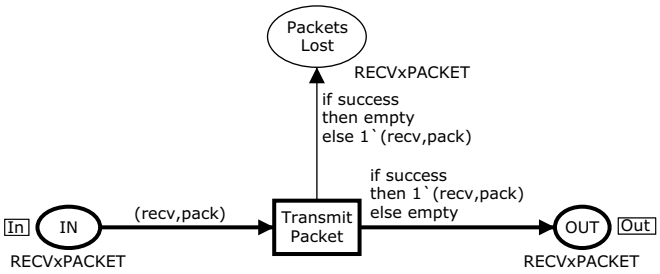


**Fig. 5.33** Transmit module for collecting lost packets: first version

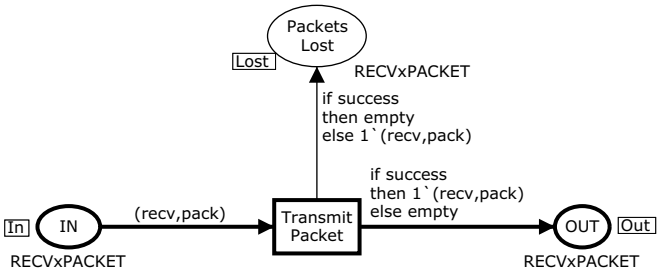


**Fig. 5.34** Transmit module for collecting lost packets: revised version with fusion set

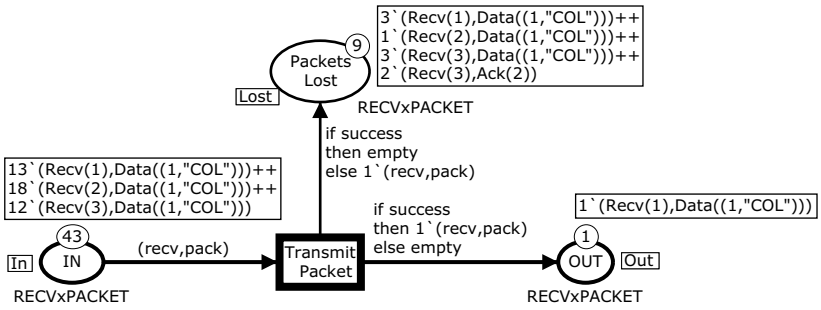


**Fig. 5.35** Marking of Transmit instance corresponding to TransmitData

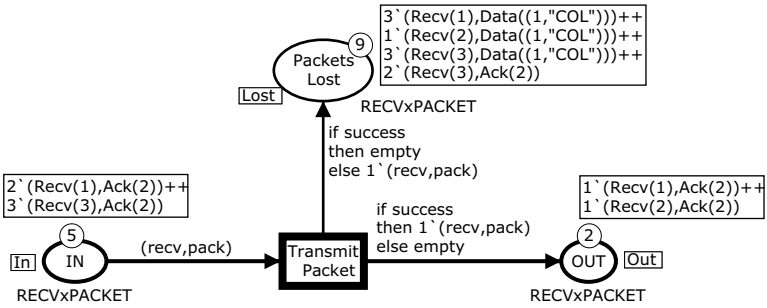


**Fig. 5.36** Marking of Transmit instance corresponding to TransmitAck

However, the two instances of the fusion place PacketsLost have the same marking, owing to the fusion set, and it can be seen that this place contains both lost data packets and lost acknowledgements. In this example, it is only the instances of a single place in a single module that belong to the fusion set. However, in general it is possible for any number of places in different modules to belong to the same fusion set. This means that all of the corresponding place instances represent a single compound place.

Another typical use of fusion sets is in the initialisation of a CPN model. It is often the case that a CPN model can be set up to run in different configurations, and larger CPN models typically have a number of parameters which determine the configuration. For the CPN model of the protocol, we might be interested in configuring the data packets to be transmitted and configuring the Transmit module such that it is possible to run the model with a reliable or unreliable network. Since this configuration information is related to several modules, it is convenient to create a single Initialisation module where it is possible to set the configuration for the entire CPN model. Figure 5.37 shows the initial marking of such an Initialisation module for configuring the protocol model as outlined above. The Initialisation module is a prime module of the CPN model, and becomes a root in the module hierarchy and the instance hierarchy in a way similar to that for the other prime module, Protocol.

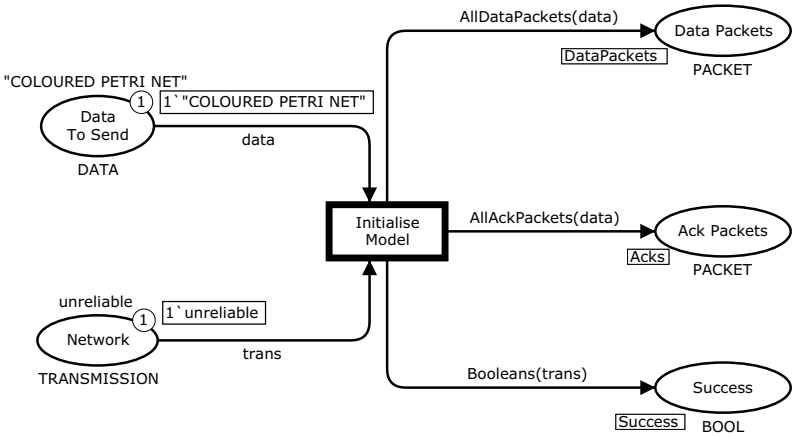


**Fig. 5.37** Initial marking of the Initialisation module

This illustrates that it is possible to have multiple prime modules in a hierarchical CPN model, and in this case the instance hierarchy becomes a forest of directed trees rather than just a single directed tree.

The place DataToSend contains a token representing the string of data to be transmitted. The place Network contains a token specifying whether the network is reliable or unreliable. By changing the initial marking of these places, we can set the configuration of the protocol. The colour set `TRANSMISSION` is an enumeration colour set defined as

```
colset TRANSMISSION = with reliable | unreliable;
```

The initial marking of the place Network is `unreliable`, meaning that packets can be lost on the network. If we set the initial marking of the place to `reliable`, no packet loss will occur. The transition InitialiseModel has two variables, declared as

```
var data  : DATA;
var trans : TRANSMISSION;
```

for accessing the configuration information given by the tokens on the places DataToSend and Network. The transition InitialiseModel is the only enabled transition in the initial marking. When the transition InitialiseModel occurs, `data` will be bound to the string on the place DataToSend and `trans` will be bound to the colour of the token on the place Network.

The transition InitialiseModel, adds tokens to the places DataPackets, AckPackets, and Success, which belongs to the fusion sets DataPackets, Acks, and Success, respectively. The functions in the arc expressions on the output arc to DataPackets and AckPackets use a common utility function `SplitData` to split the string bound to `data` into a set of data packets respecting the packet length supported by the network. This function is defined as

```
val PacketLength = 3;

fun SplitData (data) =
    let
      val pl = PacketLength;

      fun splitdata (n,data) =
          let
            val dl = String.size (data)
          in
            if dl <= pl
            then [(n,data)]
            else (n,substring (data,0,pl))::
                 splitdata
                    (n+1,substring (data,pl,dl-pl))
          end;
    in
       splitdata (1,data)
    end;
```

The function `SplitData` has a local environment for binding `pl` to the packet length and defining a recursive function `splitdata`, which does the actual splitting of the data string into a list of pairs, where each element consists of a sequence number and a data payload. As an example, the result of evaluating the expression

```
SplitData("COLOURED PETRI NET")
```

is the following list of pairs:

```
[(1,"COL"),(2,"OUR"),(3,"ED "),
 (4,"PET"),(5,"RI "),(6,"NET")]
```

The first parameter `n` of the function `splitdata` gives the sequence number of the first data packet to be produced. The second parameter is the data string to be split into data packets. The function uses a local environment to bind `dl` to the data length. The predefined function `String.size` is used to obtain the length of the data string. If the data string fits into a single data packet, such a data packet is returned. Otherwise, a data packet is generated containing the first `pl` characters of the data string, and a recursive call is made to `splitdata` to generate the data packets for the remainder of the data string. The function `substring` is used to extract the correct prefix and postfix to be used in the data packet and in the recursive call to the function.

The functions `AllDataPackets` and `AllAckPackets` are defined as follows, using the function `SplitData` from above:

```
fun AllDataPackets (data) =
    (List.map
      (fn (n,d) => Data(n,d)) (SplitData (data)));

fun AllAckPackets (data) =
    (List.map
      (fn (n,_) => Ack(n+1)) (SplitData (data)));
```

The function `AllDataPackets` uses the function `List.map`. In this case, the first argument is the function `fn (n,d) => Data(n,d)`, which, given a pair `(n,d)`, constructs the corresponding data packet. The list provided as the second argument is the list of pairs returned by the function `SplitData`. The function `AllAckPackets` is implemented in a similar way, except that it produces the acknowledgements corresponding to the data packets. The function provided to `List.map` is in this case `fn (n,_) => Ack(n+1)`. The sequence number has `1` added to it, since the acknowledgement of the data packet with sequence number `n` is `Ack(n+1)`. Recall that multisets in CPN Tools are represented using lists, i.e., a multiset is represented as a list of the elements in the multiset. This is the reason why the types of `AllDataPackets` and `AllAckPackets` match the colour sets of the places DataPackets and AckPackets, respectively.

The arc expression on the arc to the place Success uses the function `Booleans`, defined as

```
fun Booleans reliable   = 1`true
  | Booleans unreliable = 1`true ++ 1`false;
```

If the token on the place Network is `reliable`, a single token with the value `true` is put on the place Success. If the token on the place Network is `unreliable`, two tokens with the values `true` and `false` are put on the place Success. The purpose of the token(s) on the place Success will be clear when we present the modified Transmit module below.

Figure 5.38 shows the marking of the Initialisation module after the occurrence of the transition InitialiseModel in the initial marking shown in Fig. 5.37. Figure 5.39 shows the Protocol module, where the place PacketsToSend now belongs to the fusion set DataPackets. Figure 5.40 shows the Sender module, where the place Acks belongs to the fusion set Acks. This means that when the transition InitialiseModel occurs, the places PacketsToSend and Acks receive the same tokens as do the places in the Initialisation module that belong to the same fusion sets. In this way, tokens determining the configuration of the protocol are distributed to the relevant modules in the model.

The Transmit module is shown in Fig. 5.41. The place Success belongs to the fusion set Success. This place specifies the possible bindings for the variable `success`, which determine whether transmission is successful or not. In the marking shown, there are two tokens `true` and `false` present on this place. Hence, both successful transmission and loss of packets are possible. If the place Network in Fig. 5.37 initially contains the token `reliable`, then only a token with colour

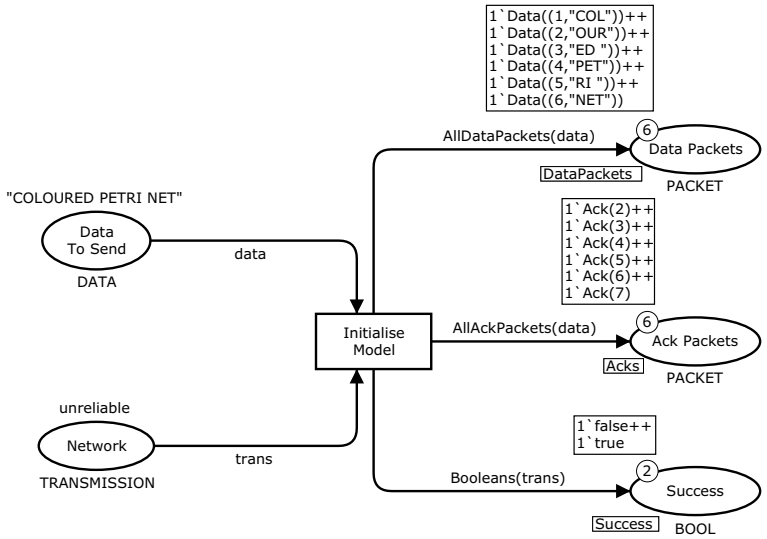


**Fig. 5.38** Marking of the Initialisation module when InitialiseModel has occurred

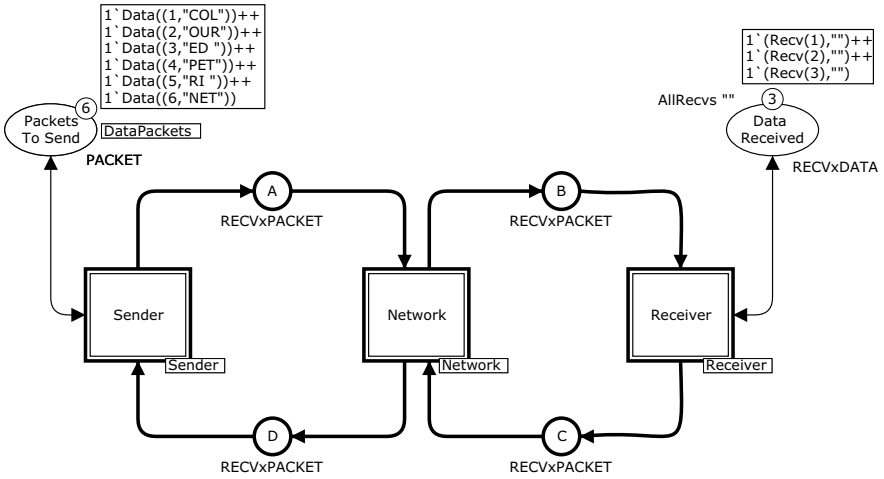


**Fig. 5.39** Marking of the Protocol module after initialisation

`true` will be present on the place Success and hence only successful transmission is possible.

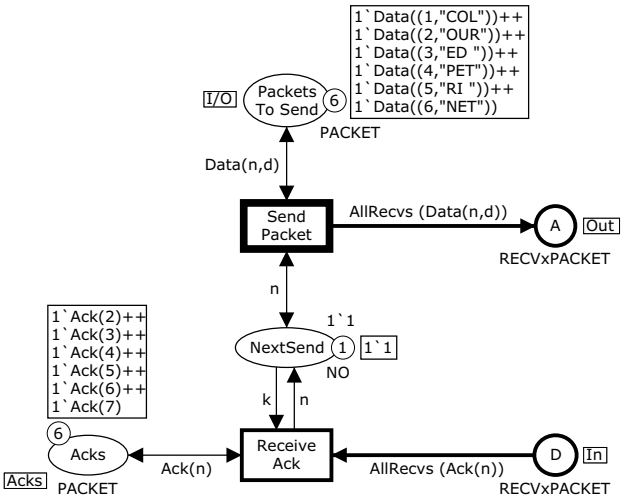


**Fig. 5.40** Marking of the Sender module after initialisation

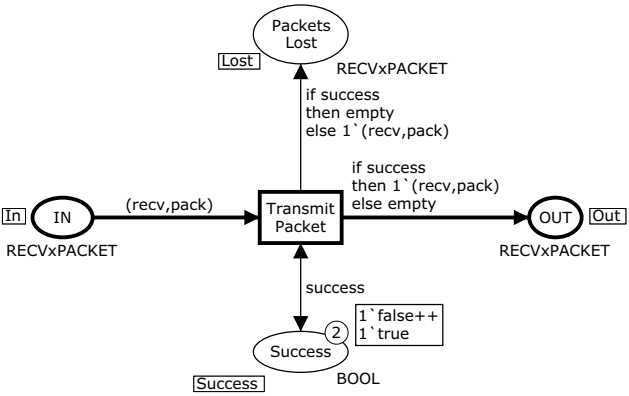


**Fig. 5.41** Marking of the Transmit module after initialisation

In the above, we have used the initial marking of certain places (in this case DataToSend and Network) to specify the configuration of the protocol. It is also possible to use files or dialogue boxes to provide the configuration information. We shall illustrate the latter in Chap. 13.

Fusion sets can also be used to reduce the number of crossing arcs in a module. When a place needs to be accessed by many different transitions, it may be impossible to avoid crossing arcs, which make the CPN model difficult to read. A way to reduce the number of crossing arcs is to split such places into two or more copies and then create a fusion set that glues them together. Now it is possible to position the copies of the places in different parts of the module and thereby reduce the number of crossing arcs.

## 5.6 Unfolding Hierarchical CPN Models

A hierarchical CPN model can always be *unfolded* into an equivalent non-hierarchical CPN model with the same behaviour using a process consisting of three steps:

1. Replace each substitution transition with the content of its associated submodule such that related port and socket places are merged into a single place.
2. Collect the content of all resulting prime modules into a single module. Recall that prime modules are the roots of the module hierarchy.
3. Merge the places in each fusion set into a single place.

To illustrate the processes of replacing substitution transitions with their associated submodules and merging the places in a fusion set into a single place, we consider the CPN model of the previous section together with the Network module shown in Fig. 5.42.

The result of replacing the two substitution transitions in Fig. 5.42 with the content of their associated submodules (see Fig. 5.41) and merging the fusion places is shown in Fig. 5.43. For the substitution transition TransmitData, we have replaced the port place IN and the related socket place A with a single place named A. The port place OUT and the related socket place B have been replaced with a single place named B. Similar replacements have been done for the ports and socket places of the substitution transition TransmitAck. The fusion places named PacketsLost which were present in each of the submodules associated with the substitution transitions have been merged into a single place named PacketsLost. A similar merging has been done with the fusion places named Success. The places PacketsLost and Success are still fusion places, as one of them eventually has to be merged with the corresponding fusion place in the Initialisation module.

The fact that a hierarchical CPN model can always be transformed into an equivalent non-hierarchical CPN model implies that the hierarchy-related concepts of CP-nets do not (in theory) add expressive power to the modelling language. Any system that can be modelled with a hierarchical CPN model can also be modelled with a non-hierarchical CPN model. In practice, however, the hierarchy constructs

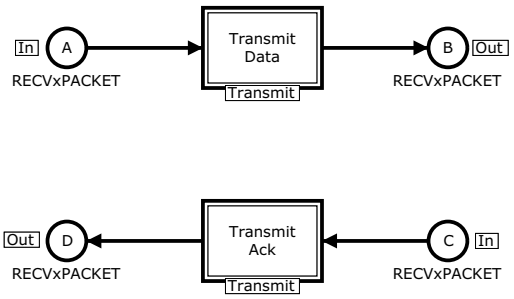


**Fig. 5.42** Network module

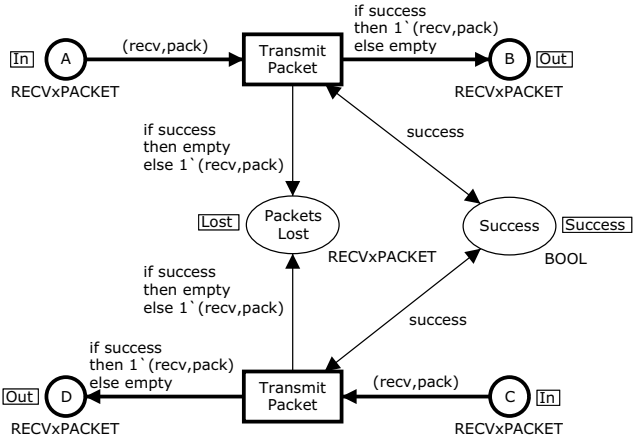


**Fig. 5.43** Unfolded Network module

have significant importance as they make it possible to structure large models and thereby cope with the complexity of large systems.

In this section, we have shown that every hierarchical CPN model can be transformed into an equivalent non-hierarchical CPN model. In Sect. 2.4 of [60], it was shown that every non-hierarchical CP-net can be transformed into an equivalent low-level Place/Transition Net (PTN) as defined in [93]. The idea behind the transformation is very simple. Each CPN place is replaced with as many PTN places as there are colours in the colour set of the CPN place, and each CPN transition is replaced with as many PTN transitions as there are possible bindings satisfying the guard for the CPN transition. For a CPN model with infinite colour sets, this will result in a PTN model with an infinite number of places and transitions.

The fact that a CPN model can always be transformed into an equivalent PTN model implies that the introduction of the coloured tokens in CP-nets does not (in theory) add expressive power to Petri Net models. Any system that can be modelled with a CPN model can also be modelled with a PTN model. In practice, however, CPN models are much more succinct and more suitable for the modelling of complex systems. The CPN modelling language allows the modeller to work on a higher abstraction level using types (colour sets) instead of bits (uncoloured tokens).

The step from PTN models to hierarchical CPN models is very similar to the step from low-level machine languages (without types, procedures, functions, or modules) to high-level programming languages offering such abstraction mechanisms. The high-level modelling and programming languages have the same (theoretical) expressive power as the corresponding low-level languages, but the high-level languages have much more (practical) structuring power, and this makes it possible for modellers and programmers to cope with the overwhelming amount of detail in real-life concurrent systems.