Int J Softw Tools Technol Transfer (2007) 9:213–254
DOI 10.1007/s10009-007-0038-x

SPECIAL SECTION CPN 04/05

# Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems

**Kurt Jensen · Lars Michael Kristensen · Lisa Wells**

Published online: 13 March 2007


**Abstract** Coloured Petri Nets (CPNs) is a language for the modelling and validation of systems in which concurrency, communication, and synchronisation play a major role. Coloured Petri Nets is a discrete-event modelling language combining Petri nets with the functional programming language Standard ML. Petri nets provide the foundation of the graphical notation and the basic primitives for modelling concurrency, communication, and synchronisation. Standard ML provides the primitives for the definition of data types, describing data manipulation, and for creating compact and parameterisable models. A CPN model of a system is an executable model representing the states of the system and the events (transitions) that can cause the system to change state. The CPN language makes it possible to organise a model as a set of modules, and it includes a time concept for representing the time taken to execute events in the modelled system. CPN Tools is an industrial-strength computer tool for constructing and analysing CPN models. Using CPN Tools, it is possible to investigate the behaviour of the modelled system using simulation, to verify properties by means of state space methods and model checking, and to conduct simulation-based performance analysis. User interaction with CPN Tools is based on direct manipulation of the graphical representation of the CPN model using interaction techniques, such as tool palettes and marking menus. A license for CPN Tools can be obtained free of charge, also for commercial use.

**Keywords** Coloured Petri Nets · Discrete-event systems · Behavioural modelling · Validation · Simulation · Verification · State space methods · Model checking · Performance analysis · Visualisation

K. Jensen · L. M. Kristensen · L. Wells (✉)
Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, 8200 Aarhus N, Denmark
e-mail: wells@daimi.au.dk

K. Jensen
e-mail: kjensen@daimi.au.dk

L. M. Kristensen
e-mail: lmkristensen@daimi.au.dk

## 1 Introduction

Systems engineering is a comprehensive discipline involving a multitude of activities such as requirements engineering, design and specification, implementation, testing, and deployment. The development of distributed systems is particularly challenging. A major reason is that these systems possess concurrency and nondeterminism which means that the execution of such systems may proceed in many different ways. It is extremely easy for the human designer to miss some important interaction patterns when designing such a system, leading to gaps or malfunctions in the system design. To cope with the complexity of modern concurrent systems, it is therefore crucial to provide methods that enable debugging and testing of central parts of the system design prior to implementation and deployment.

One way to approach the challenge of developing concurrent systems is to build an executable model of the system. Constructing a model and simulating it usually leads to significant new insights into the design and operation of the system considered and often results in a simpler and more streamlined design. Furthermore, constructing an executable model usually leads to a more complete specification facilitating a systematic

investigation of scenarios which can significantly decrease the number of design errors.

Coloured Petri Nets (CP-nets or CPNs) [16,17,19,23] is a graphical language for constructing models of concurrent systems and analysing their properties. CP-nets is a discrete-event modelling language combining Petri nets [33] and the functional programming language CPN ML which is based on Standard ML [36,37]. The CPN modelling language is a general purpose modelling language, i.e., it is not focused on modelling a specific class of systems, but aimed towards a very broad class of systems that can be characterised as concurrent systems. Typical application domains of CP-nets are communication protocols [6], data networks [5], distributed algorithms [34], and embedded systems [1,41]. CP-nets are, however, also applicable more generally for modelling systems where concurrency and communication are key characteristics. Examples of these are business process and workflow modelling [39], manufacturing systems [11], and agent systems [31]. Examples of industrial applications of CP-nets within different domains are available via [12]. An introduction to the practical use of CP-nets is also given in [19,24].

A CPN model of a system describes the states of the system and the events (transitions) that can cause the system to change state. By making simulations of the CPN model, it is possible to investigate different scenarios and explore the behaviours of the system. Very often, the goal of simulation is to debug and investigate the system design. CP-nets can be simulated interactively or automatically. An interactive simulation is similar to single-step debugging. It provides a way to "walk through" a CPN model, investigating different scenarios in detail and checking whether the model works as expected. During an interactive simulation, the modeller is in charge and determines the next step by selecting between the enabled events in the current state. It is possible to observe the effects of the individual steps directly on the graphical representation of the CPN model. Automatic simulation is similar to program execution. The purpose is to simulate the model as fast as possible and it is typically used for testing and performance analysis. For testing purposes, the modeller typically sets up appropriate breakpoints and stop criteria. For performance analysis the model is instrumented with data collectors to collect data concerning the performance of the system.

Time plays a significant role in a wide range of concurrent systems. The correct functioning of some systems crucially depends on the time taken by certain activities, and different design decisions may have a significant impact on the performance of a system. CP-nets include a time concept that makes it possible to capture the time taken to execute activities in the system. The time concept also means that CP-nets can be applied for simulation-based performance analysis, investigating performance measures such as delays, throughput, and queue lengths in the system, and for modelling and validation of real-time systems.

CPN models can be structured into a set of modules to handle large specifications. The modules interact with each other through a set of well-defined interfaces, in a similar way as in programming languages. The module concept of CP-nets is based on a hierarchical structuring mechanism, allowing a module to have submodules and allowing a set of modules to be composed to form a new module.

Visualisation is a technique that uses high-level graphics to animate the behaviour of CPN models, and it is closely related to simulation of CPN models. An important application of visualisation is that it allows for the presentation of design ideas and analysis results using application domain concepts. This is particularly important in discussions with people and colleagues unfamiliar with CP-nets. Several means exist for adding domain-specific graphics on top of a CPN model. This can be used to abstractly visualise the execution of the CPN model in the context of the application domain. One example of this is to use message sequence charts [15] (or sequence diagrams [29]) to visualise the exchange of messages in the execution of a communication protocol.

CPN models are formal—in the sense that the CPN modelling language has a mathematical definition of its syntax and semantics. This means that they can be used to verify system properties, i.e., prove that certain desired properties are fulfilled or that certain undesired properties are guaranteed to be absent. Verification of system properties is supported by a set of state space methods. The basic idea underlying state spaces is to compute all reachable states and state changes of the CPN model and represent these as a directed graph where nodes represent states and arcs represent occurring events. State spaces can be constructed fully automatically. From a constructed state space it is possible to answer a large set of verification questions concerning the behaviour of the system such as absence of deadlocks, the possibility of always being able to reach a given state, and the guaranteed delivery of a given service. The state space methods of CP-nets can also be applied to timed CP-nets. Hence, it is also possible to verify the functional correctness of systems modelled by means of timed CP-nets.

It should be stressed that for the practical use of CP-nets and their supporting computer tools, it suffices to have an intuitive understanding of the syntax and semantics of the CPN modelling language. This is

analogous to ordinary programming languages such as JAVA that are successfully applied by programmers who are usually not familiar with the formal definitions of the languages. This underpins the important property that CP-nets can be taught and learned without studying the associated formal definitions.

The practical application of CPN modelling and analysis relies heavily on the existence of computer tools supporting the creation and manipulation of models. CPN Tools [10] is a tool suite for editing, simulation, state space analysis, and performance analysis of CPN models. The user of CPN Tools works directly on the graphical representation of the CPN model. The graphical user interface (GUI) of CPN Tools has no conventional menu bars and pull-down menus, but is based on interaction techniques such as *tool palettes* and *marking menus*. A license for CPN Tools can be obtained free of charge via the CPN Tools web pages [10]. CPN Tools is currently licensed to more than 4,000 users in more than 115 different countries and is available for MS Windows and Linux.

Reader's guide

This paper gives a brief introduction to the CPN modelling language and illustrates how construction, simulation, state space analysis, performance analysis, and visualisation are supported by CPN Tools. Section 2 introduces the concepts of the CPN modelling language. Section 3 illustrates how construction of CPN models is supported by CPN Tools, and Sect. 4 shows how simulation is supported. Section 5 gives a brief introduction to state space methods and explains how they are supported in CPN Tools. Section 6 introduces the basic ideas of simulation-based performance analysis and explains how it is supported by CPN Tools. Section 7 illustrates how domain-specific visualisation is supported by CPN Tools. Finally, Section 8 concludes the paper and provides references to further material on the CPN modelling language, practical examples, and use of CPN Tools.

It is not necessary to read the entire paper or to be familiar with Standard ML to get started using CP-nets and CPN Tools. To learn the basics it is sufficient to read the following: the introduction to the concepts of non-hierarchical CP-nets (Sects. 2.1–2.3), the introduction to CPN Tools and the tools for constructing non-hierarchical models (Sects. 3.1, 3.2, 3.4, and 3.5), and the description of simulating CP-nets (Sect. 4).

The remaining sections of the paper present more advanced topics. Hierarchical CP-nets are introduced in Sect. 2.4, and tools for constructing hierarchical models are presented in Sect. 3.3. Readers interested in performance analysis should read the introduction to timed CP-nets (Sect. 2.5), and Sect. 6 on performance analysis. State space analysis and visualisation are described in Sects. 5 and 7, respectively.

A basic introduction to CPN ML and many examples of how to use CPN ML can be found in the help pages for CPN Tools (which can also be found online via [10]). It is not necessary to have a good understanding of Standard ML to use the basic features of the performance, state space, and visualisation facilities. However, to make effective use of the more advanced features of these facilities, it is necessary to understand Standard ML. Again, the help pages provide a number of examples and descriptions of how Standard ML is used to support advanced analysis techniques.

## 2 The CPN modelling language

In this section, we introduce the CPN modelling language by means of a small running example modelling a communication protocol. We use a simple protocol since it is easy to explain and understand, and because it involves concurrency, non-determinism, communication, and synchronisation which are key characteristics of concurrent systems. The protocol itself is unsophisticated, but yet complex enough to illustrate the constructs of the CPN modelling language. No prior knowledge of protocols is required.

The simple protocol consists of a *sender* transferring a number of *data packets* to a *receiver*. Communication takes place on an unreliable network, i.e., packets may be lost and overtaking is possible. The protocol uses sequence numbers, acknowledgements, and retransmissions to ensure that the data packets are delivered exactly once and in the correct order at the receiving end. The protocol uses a stop-and-wait strategy, i.e., the same data packet is transmitted until a corresponding acknowledgement is received. The data packets consist of a sequence number and the data (payload) to be transmitted. An acknowledgement consists of a sequence number specifying the number of the data packet expected next by the receiver.

### 2.1 Net structure, declarations, and inscriptions

A CPN model is usually created as a graphical drawing, and Fig. 1 shows the basic CPN model of the protocol. The left part models the sender, the middle part models the network, and the right part models the receiver. The CPN model contains eight *places* (drawn as ellipses or circles), five *transitions* (drawn as rectangular boxes), a number of directed *arcs* connecting places and transitions, and finally some textual *inscriptions* next to the places, transitions, and arcs. The inscriptions are written

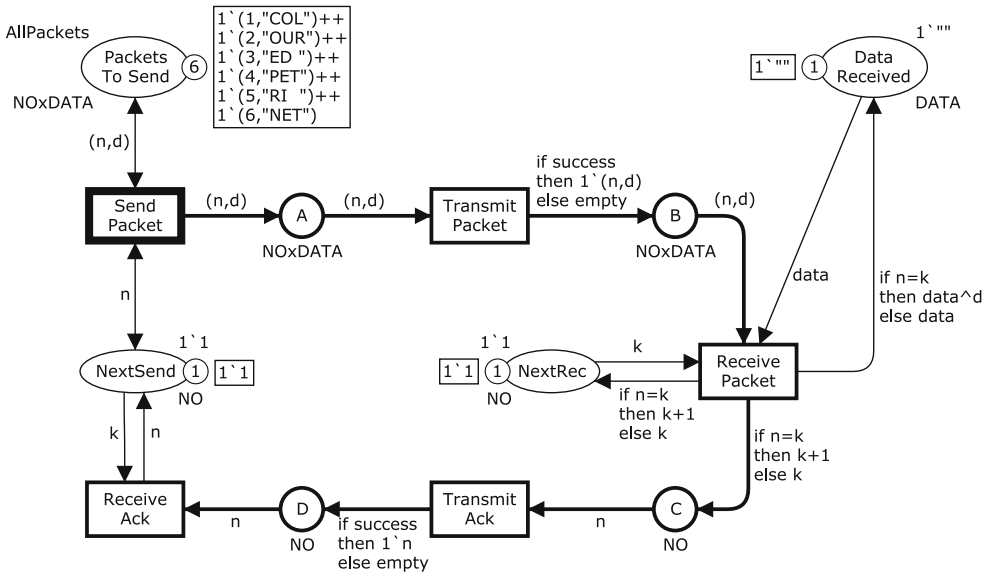



**Fig. 1** Basic CPN model of the simple protocol in the initial marking $M_0$

in the CPN ML programming language which is an extension of the Standard ML language. Places and transitions are called *nodes*. Together with the directed arcs they constitute the *net structure*. An arc always connects a place to a transition or a transition to a place. It is illegal to have an arc between two nodes of the same kind, i.e., between two transitions or two places.

The state of the modelled system is represented by the places. Each place can be marked with one or more *tokens*, and each token has a data value attached to it. This data value is called the *token colour*. It is the number of tokens and the token colours on the individual places which together represent the state of the system. This is called a *marking* of the CPN model, while the tokens on a specific place constitute the marking of that place. By convention, we write the names of the places inside the ellipses. The names have no formal meaning—but they have huge practical importance for the readability of a CPN model (just like the use of mnemonic names in traditional programming). The state of the sender is modelled by the two places PacketsToSend and NextSend. The state of the receiver is modelled by the two places DataReceived and NextRec, and the state of the network is modelled by the places A, B, C, and D.

Next to each place, there is an inscription which determines the set of token colours (data values) that the tokens on the place are allowed to have. The set of possible token colours is specified by means of a type (as known from programming languages), and it is called the *colour set* of the place. By convention the colour set is written below the place. The places NextSend, NextRec, C, and D have the colour set `NO`. In CPN Tools, colour sets are defined using the CPN ML keyword `colset`, and the colour set `NO` is defined to be equal to the integer type `int`:

```
colset NO = int;
```

This means that tokens residing on the four places NextSend, NextRec, C, and D will have an integer as their token colour. The colour set `NO` is used to model the sequence numbers in the protocol. The place Data Received has the colour set `DATA` defined to be the set of all text strings `string`. The colour set `DATA` is used to model the payload of data packets. The remaining three places have the colour set `NOxDATA` which is defined to be the `product` of the types `NO` and `DATA`. This type contains all two-tuples (pairs) where the first element is an integer and the second element is a text string. Tuples are written using parentheses `(` and `)` around a comma-separated list. The colour set `NOxDATA` is used to model the data packets which contain a sequence number and some data. The colour sets are defined as:

```
colset DATA    = string;
colset NOxDATA = product NO * DATA;
```

Next to each place, we find another inscription which determines the *initial marking* of the place. The initial marking inscription of a place is by convention written above the place. For example, the inscription at the upper right side of the place NextSend specifies that the initial marking of this place consists of one token with the colour (value) 1. This indicates that we want data packet number 1 to be the first data packet to be sent. Analogously, the place NextRec has an initial marking consisting of a single token with the colour 1.

This indicates that the receiver is initially expecting the data packet with sequence number 1. The place DataReceived has an initial marking which consists of one token with colour `""` (which is the empty text string). This indicates that the receiver has initially received no data. The inscription `AllPackets` at the upper left side of place PacketsToSend is a *constant* defined as:

```
val AllPackets = 1`(1,"COL") ++ 1`(2,"OUR") ++
                 1`(3,"ED ") ++ 1`(4,"PET") ++
                 1`(5,"RI ") ++ 1`(6,"NET");
```

which specifies that the initial marking of this place consists of six tokens with the data values:

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

The ++ and ` are operators that allow for the construction of a *multi-set* consisting of token colours. A multi-set is similar to a set, except that values can appear more than once. The infix operator ` takes a non-negative integer as left argument specifying the number of appearances of the element provided as the right argument. The ++ takes two multi-sets as arguments and returns their union (sum). The initial marking of PacketsToSend consists of six tokens representing the data packets which we want to transmit. The absence of an inscription specifying the initial marking means that the place initially contains no tokens. This is the case for the places A, B, C, and D.

The *current marking* of each place is indicated next to the place. The number of tokens on the place in the current marking is shown in the small circle, while the detailed token colours are indicated in the box positioned next to the small circle. Initially, the current marking is equal to the initial marking, denoted $M_0$. As explained earlier, the initial marking has six tokens on Packets ToSend and one token on each of the places NextSend, NextRec, and DataReceived.

The five transitions (drawn as rectangles) represent the events that can take place in the system. As with places, we write the names of the transitions inside the rectangles. The transition names also have no formal meaning but they are very important for the readability of the model. When a transition *occurs*, it removes tokens from its *input places* (those places that have an arc leading to the transition) and it adds tokens to its *output places* (those places that have an arc coming from the transition). The colours of the tokens that are removed from input places and added to output places when a transition occurs are determined by means of the *arc expressions* which are the textual inscriptions positioned next to the individual arcs. A transition and a place may also be connected by *double-headed arcs*. A double-headed arc is shorthand for two directed arcs in opposite directions between a place and a transition which both have the same arc expression. This implies that the place is both an input place and an output place for the transition. The transition SendPacket and the places PacketsToSend and NextSend are connected by double-headed arcs.

The arc expressions are written in the CPN ML programming language and are built from typed variables, constants, operators, and functions. When all variables in an expression are bound to values (of the correct type) the expression can be evaluated. An arc expression evaluates to a multi-set of token colours. As an example, consider the two arc expressions: `n` and `(n,d)` on the three arcs connected to the transition SendPacket. They contain the variables `n` and `d` declared as:

```
var n : NO;
var d : DATA;
```

This means that `n` must be bound to a value of type `NO` (i.e., an integer), while `d` must be bound to a value of type `DATA` (i.e., a text string). We may, e.g., consider the *binding*:

⟨`n=3, d="CPN"`⟩

which binds `n` to `3` and `d` to `"CPN"`. For this binding the arc expressions evaluate to the following values, where → should be read as "evaluates to":

```
n       →  1`3
(n,d)   →  1`(3,"CPN")
```

Arc expressions evaluate to a multi-set of token colours, and this means that there may be zero, exactly one token, or more than one token removed from an input place or added to an output place. If an arc expression evaluates to exactly one token, then the `1`` can be omitted from the expression by convention. For example, arc expressions `n` and `(n,d)` are shorthand for `1`n` and `1`(n,d)`.

## 2.2 Enabling and occurrence of transitions

Next let us consider the occurrence of events in a CPN model. The arc expressions on the *input arcs* of a transition together with the tokens on the input places determine whether the transition is *enabled*, i.e., is able to *occur* in a given marking. For a transition to be enabled it must be possible to find a binding of the variables that appear in the surrounding arc expressions of the transition such that the arc expression of each input arc evaluates to a multi-set of token colours that is present on the corresponding input place. When the transition

occurs with a given binding, it removes from each input place the multi-set of token colours to which the corresponding input arc expression evaluates. Analogously, it adds to each output place the multi-set of token colours to which the expression on the corresponding *output arc* evaluates.

Let us now consider transition SendPacket. In Fig. 1 transition SendPacket has a thick border line, while the other four transitions do not. In CPN Tools, this indicates that SendPacket is the only transition that has an enabled binding in the initial marking $M_0$. The other transitions are disabled, i.e., they cannot occur. When this transition occurs, it removes a token from each of the input places NextSend and PacketsToSend. The arc expressions of the two double-headed arcs are `n` and `(n,d)`.

The initial marking of place NextSend contains a single token with colour `1`. This means that the variable `n` must be bound to `1`. Otherwise the expression on the arc from NextSend would evaluate to a token colour which is not present at NextSend implying that the transition is *disabled* for that binding. Now let us consider the arc expression `(n,d)` on the arc from PacketsToSend. We have already bound `n` to `1`, and now we are looking for a binding of `d` such that the arc expression `(n,d)` will evaluate to one of the six token colours that are present on PacketsToSend. Obviously, the only possibility is to bind `d` to the string `"COL"`. Hence, we conclude that the binding:

⟨`n=1, d="COL"`⟩

is the only enabled binding for SendPacket (in the initial marking). An occurrence of SendPacket with this binding removes the token with colour `1` from the input place NextSend and removes the token with colour `(1,"COL")` from the input place PacketsToSend. Since SendPacket is connected to PacketsToSend and NextSend by means of double-headed arcs, the occurrence of SendPacket with this binding will also add a token with colour `(1,"COL")` to PacketsToSend and add a token with colour `1` to NextSend. This means that tokens removed from the places PacketsToSend and NextSend according to the result of evaluating the arc expression, are immediately replaced by new tokens with the same token colours. Thus the markings of these places do not change when the transition occurs. This allows the packet to be retransmitted (to recover from loss). The occurrence of SendPacket also adds a new token with colour `(1,"COL")` to the output place A. Intuitively, this represents that the first data packet `(1,"COL")` has been sent to the network. Figure 2 shows a fragment of the CPN model in the new marking $M_1$. We show only a fragment of the CPN model since the occurrence of a

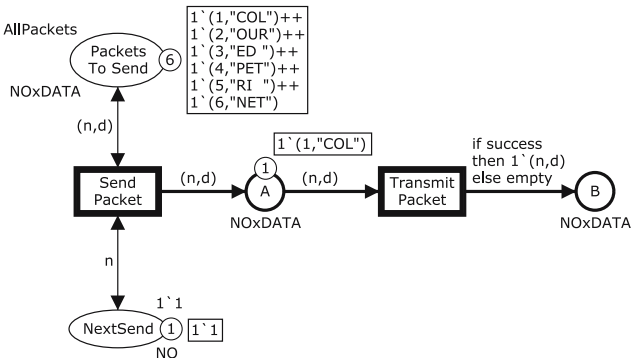



**Fig. 2** Marking $M_1$ reached when SendPacket occurs in $M_0$

transition changes only the markings of the places that are connected to the transition via an arc.

Consider the marking $M_1$ and the transition TransmitPacket which has three variables `n`, `d`, and `success`. The variable `success` is a Boolean variable declared as:

```
var success : BOOL;
```

which appears on the output arc. The colour set `BOOL` is defined to be the set of Boolean values (`{true, false}`) `bool`:

```
colset BOOL = bool;
```

In marking $M_1$, place A has a single token with colour `(1,"COL")`. The variable `success` is only found on an output arc from TransmitPacket, and this means that the variable can be bound to an arbitrary value from its colour set (which is `BOOL`). Based on the arc expression `(n,d)` on the input arc from A, it is straightforward to conclude that transition TransmitPacket is enabled with two different bindings in $M_1$:

$b_+$ = ⟨`n=1, d="COL", success=true`⟩
$b_-$ = ⟨`n=1, d="COL", success=false`⟩

The first of these bindings $b_+$ represents successful transmission over the network. If it occurs in $M_1$ the following happens:

- The data packet `(1,"COL")` is removed from input place A.
- A new token representing the same data packet is added to the output place B (in the `if-then-else` expression the condition `success` evaluates to `true` while `1`(n,d)` evaluates to `1`(1,"COL")`).

Figure 3 shows part of the marking $M_2^+$ which is the result of an occurrence of the binding $b_+$ in $M_1$.

The second binding $b_-$ represents an unsuccessful transmission, i.e., that the data packet is lost by the network. If it occurs in $M_1$ the following happens:

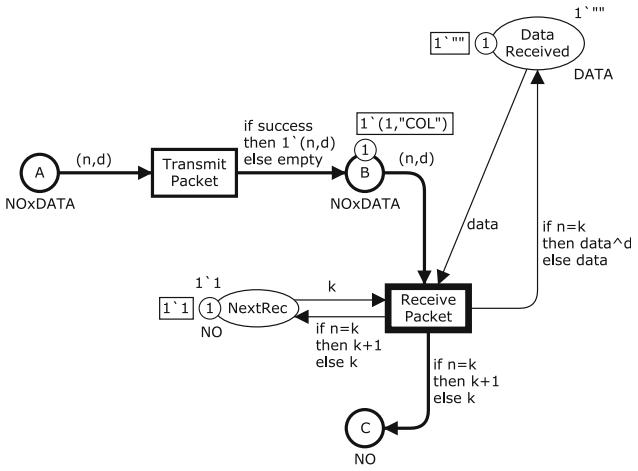



**Fig. 3** Marking $M_2^+$—successful transmission in $M_1$

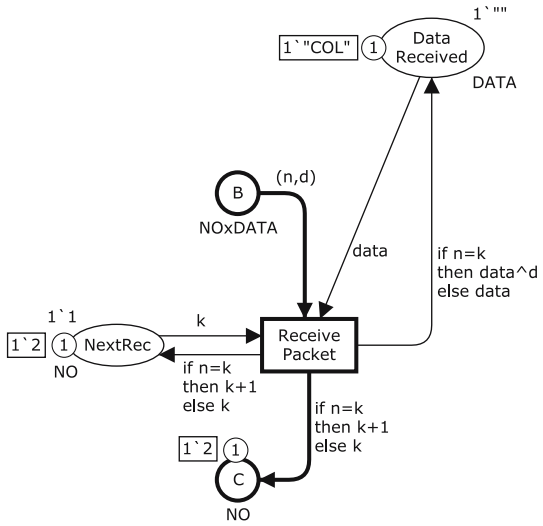



**Fig. 4** Marking $M_3$ reached when ReceivePacket occurs

– The data packet `(1,"COL")` is removed from input place A.
– No token is added to the output place B (in the `if-then-else` expression the condition `success` evaluates to `false` while the predefined constant `empty` evaluates to the empty multi-set).

An occurrence of the binding $b_-$ in $M_1$ leads back to the initial marking $M_0$ shown in Fig. 1.

Let us now consider the reception of data packets in marking $M_2^+$. The token on place NextRec represents the sequence number of the data packet that the receiver expects to receive next. The variable `k` is bound to the value of this sequence number. The variable `data` has type `DATA` (i.e., text string):

```
var data : DATA;
```

The variable `data` will be bound to the text string in the token colour of the token on place DataReceived. This text string contains the data from all of the data packets that have been received by the receiver.

When a data packet is present at place B there are two different possibilities. Either `n=k` evaluates to `true` which means that the data packet being received is the one that the receiver expects, or `n=k` evaluates to `false` which means that it is not the data packet expected. If the data packet on place B is the expected data packet (i.e., `n=k`), the following happens:

– The data packet is removed from place B.
– The data in the data packet is concatenated to the end of the data which the receiver has already received (the operator ^ is the concatenation operator for text strings).
– The token colour on place NextRec changes from `k` to `k+1`, which means that the receiver now waits for the next data packet.
– An acknowledgement is put on place C. The acknowledgement contains the sequence number of the data packet that the receiver is expecting next.

The occurrence of the transition ReceivePacket in the marking $M_2^+$ from Fig. 3 corresponds to the reception of the expected data packet. Figure 4 shows the marking $M_3$ reached when ReceivePacket occurs in $M_2^+$.

If the data packet on B is not the expected data packet (i.e., `n`$\neq$`k`), the following happens:

– The data packet is removed from place B.
– The data in the data packet is ignored (the marking of DataReceived does not change).
– The token colour on place NextRec does not change, which means that the receiver is waiting for the same data packet as before.
– An acknowledgement is put on place C. The acknowledgement contains the sequence number of the data packet that the receiver is expecting next.

Transition TransmitAck has a behaviour which is similar to the behaviour of TransmitPacket. It removes acknowledgements from place C and adds them to place D in case of a successful transmission. Let $M_4$ be the marking reached from $M_3$ by the occurrence of TransmitAck with the binding ⟨`n=2`, `success=true`⟩.

Let us now consider the reception of acknowledgements. When the transition ReceiveAck occurs, it removes an acknowledgement from place D and updates the token on NextSend to contain the sequence number specified in the acknowledgement. The sender is now able to send the next packet, according to the stop-and-wait strategy.

Suppose that the transition ReceiveAck occurs with the binding ⟨n=2, k=1⟩ in marking $M_4$. This will lead to a marking $M_5$ which represents a state where the sender is ready to send data packet number 2 (since the first data packet is now known to have been successfully received). This marking is similar to the initial marking, but the tokens on places NextSend and NextRec have colour 2 (instead of 1), and the token on DataReceived has colour "COL" (instead of "").

Above, we have described the sending, transmission, and reception of data packet number 1 and the corresponding acknowledgement. In the CPN model this corresponds to the occurrence of five transitions with enabled bindings. A pair consisting of a transition and a binding for the variables of the transition is called a *binding element*. Below we have listed the five occurring binding elements:

(SendPacket,⟨n=1, d="COL"⟩)
(TransmitPacket, ⟨n=1, d="COL", success=true⟩)
(ReceivePacket, ⟨n=1, d="COL", k=1, data=""⟩)
(TransmitAck, ⟨n=2, success=true⟩)
(ReceiveAck, ⟨n=2, k=1⟩)

Transitions are also allowed to have a *guard*, which is a Boolean expression. When a guard is present it must evaluate to true for the binding to be enabled, otherwise the binding is disabled and cannot occur. Hence, a guard puts an additional constraint on the enabling of bindings for the transition. An example of a guard will be given in Sect. 6.1.

### 2.3 Steps, concurrency and conflict

Now let us consider the behaviour of the CPN model in further detail. We have seen that a single binding element is enabled in the initial marking:

(SendPacket, ⟨n=1, d="COL"⟩)

When it occurs, it leads to the marking $M_1$ that is shown in Fig. 2. In marking $M_1$ three different binding elements are enabled:

SP = (SendPacket,⟨n=1, d="COL"⟩)
$TP_+$ = (TransmitPacket, ⟨n=1, d="COL", success=true⟩)
$TP_-$ = (TransmitPacket, ⟨n=1, d="COL", success=false⟩)

The first binding element represents a retransmission of data packet number 1. The second binding element represents a successful transmission of data packet number 1 over the network, while the third binding element represents the loss of the data packet by the network. The last two binding elements, $TP_+$ and $TP_-$, are in *conflict* with each other. Both of them are enabled, but only one of them can occur since each of them needs a token from place A, and there is only one such token in $M_1$. However, the binding elements SP and $TP_+$ can occur *concurrently* (i.e., in parallel). To occur, SP needs a token from place PacketsToSend and a token on NextSend, while $TP_+$ needs a token from place A. In other words, both binding elements can get the tokens they need without competition/interference with the other binding element. A multi-set of binding elements is *concurrently enabled* in a given marking if there are enough tokens on the input places of the transitions in question to simultaneously satisfy the demands of all of the binding elements. By a similar argument, we see that SP and $TP_-$ are concurrently enabled.

A *step* in general consists of a (non-empty and finite) multi-set of concurrently enabled binding elements. A step may consist of a single binding element. We do not consider the empty multi-set of binding elements to be a legal step since it would have no effect and always be enabled. The effect of the occurrence of a set of concurrently enabled binding elements is the sum of the effects caused by the occurrence of the individual binding elements. This means that the marking reached will be the same as the one which will be reached if we let the set of binding elements occur *sequentially* (i.e., one after each other in some arbitrary order).

Now let us assume that the first and second of the three enabled binding elements in marking $M_1$ occur concurrently with each other, i.e., that we have the following step (written as a multi-set of binding elements):

1‘(SendPacket,⟨n=1, d="COL"⟩) ++
1‘(TransmitPacket, ⟨n=1, d="COL", success=true⟩)

We then reach the marking $M_2$ which is partly shown in Fig. 5. In marking $M_2$ we have four enabled binding elements:

SP = (SendPacket, ⟨n=1, d="COL"⟩)
$TP_+$ = (TransmitPacket, ⟨n=1, d="COL", success=true⟩)
$TP_-$ = (TransmitPacket, ⟨n=1, d="COL", success=false⟩)
RP = (ReceivePacket, ⟨n=1, d="COL", k=1, data=""⟩)

As before, we have a conflict between $TP_+$ and $TP_-$, while all the other binding elements are concurrently enabled since there are enough input tokens to simultaneously satisfy the demands of each binding element.

An execution of a CPN model is, in general, described by means of an *occurrence sequence*, which specifies the steps that occur and the *intermediate markings* that

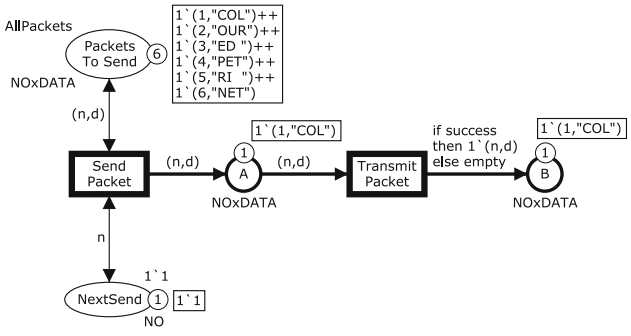



**Fig. 5** Marking $M_2$ reached when SendPacket and TransmitPacket occur in $M_1$

are reached. A marking that is reachable via an occurrence sequence starting in the initial marking is called a *reachable marking*. The existence of a reachable marking with more than one enabled binding element makes the CPN model *non-deterministic*. This means that there exist different occurrence sequences containing different sequences of steps and leading to different reachable markings. It is important to stress that it is only the choice between the enabled steps which is non-deterministic. The individual steps themselves are deterministic in the sense that once an enabled step has been selected in a given marking, the marking resulting from its occurrence is uniquely determined (unless random functions are used, as we will illustrate in Sect. 6).

Figure 6 shows an intermediate marking $M^*$ reached after the occurrence of a sequence of steps of the CPN model. In the marking $M^*$ place A has two tokens with colour `(1,"COL")` and one token with colour `(2,"OUR")`. In this situation, TransmitPacket is enabled in bindings corresponding to transmission of data packet 1 and data packet 2. In a CPN model there are no requirements on the order in which tokens are removed from places. This means that in $M^*$ it is possible to let the binding of TransmitPacket corresponding to a successful transmission of data packet 2 occur, and in this way let the data packet with sequence number 2 on place A overtake the two data packets with sequence number 1. Since there are two tokens with colour `(1,"COL")` it also means that $TP_+$ and $TP_-$ can occur concurrently because there is a token on A for each of the two binding elements. It also means that $TP_+$ can occur *concurrently with itself*, and the same is true for $TP_-$. Thus it is possible to transmit multiple packets on the network concurrently.

The transition SendPacket is enabled in $M^*$ in a binding corresponding to a retransmission of data packet 2 even if data packet 2 is still in transit. The retransmission is possible because the transmission of data packet 2 is too slow and hence could be out-raced by the second occurrence of SendPacket (i.e., the retransmission of packet number 2). This means that we have described a time-related behaviour without the explicit use of time. What is important at the chosen abstraction level is not when a retransmission may occur, but the simple fact that it is possible that such a retransmission can occur. We will discuss the execution of CPN models in more detail in Sect. 4 when presenting how simulation is supported by CPN Tools.

## 2.4 Modules

We now show how a CPN model can be organised as a set of hierarchically related *modules*—in a similar way as programs are constructed from modules. To illustrate

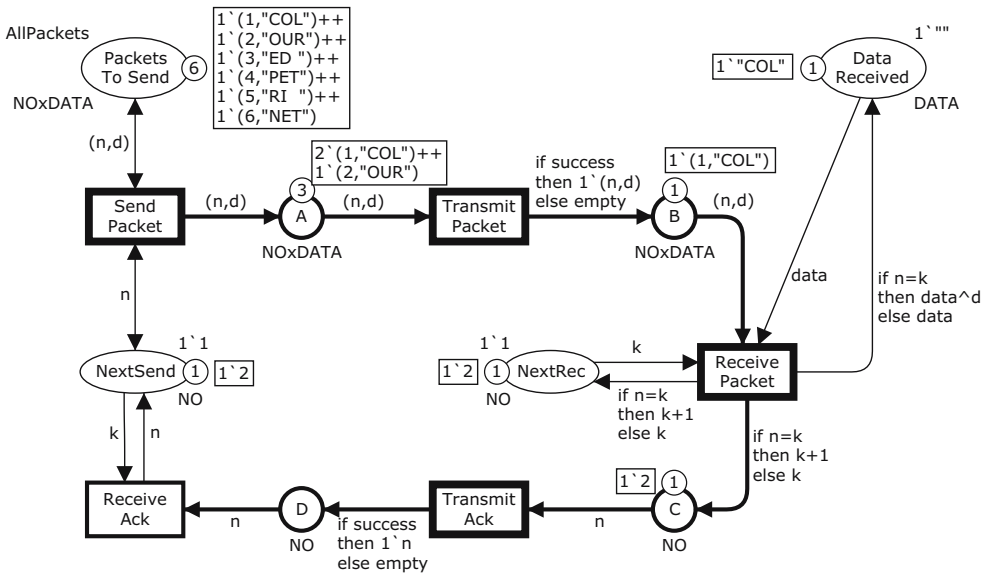



**Fig. 6** Intermediate marking $M^*$ reached after an occurrence sequence starting in $M_0$

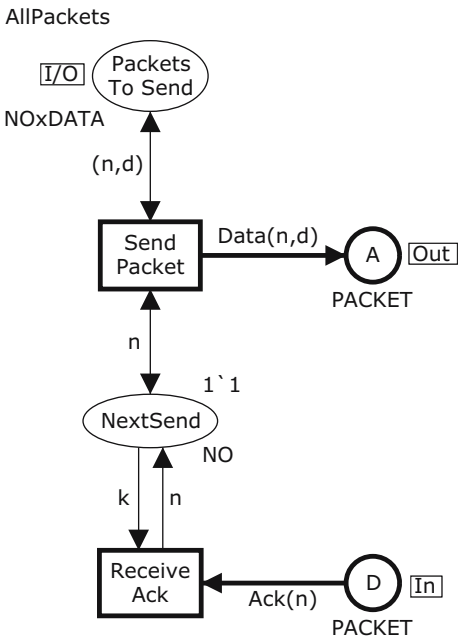



**Fig. 7** Module for the sender

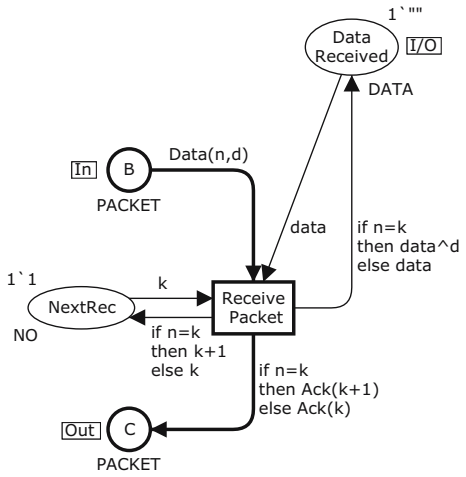



**Fig. 8** Module for the receiver

the use of modules, we revisit the CPN model of the protocol from Fig. 1 and develop a hierarchical CPN model for the protocol example. A straightforward idea is to create a module for the sender, a module for the network, and a module for the receiver. Furthermore, if we take a closer look at the network part of the model in Fig. 1, we notice that it contains two transitions TransmitPacket and TransmitAck that have a very similar behaviour. Hence, it would be natural to use the same module to represent the behaviour of Transmit-Packet and Transmit-Ack. However, the involved token colours are slightly different. The TransmitPacket transition deals with data packets, represented by tokens of type `NOxDATA`, while the TransmitAck transition deals with acknowledgements, represented by tokens of type `NO`. This means that we cannot immediately use the same module to represent the behaviour of TransmitPacket and TransmitAck. To overcome this problem, we use the `union` colour set `PACKET` defined as follows:

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

The colour set `PACKET` 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)`).

Figure 7 shows the Sender module, which contains two transitions and four places. Place D is an *input port*, place A is an *output port*, while place PacketsToSend is an *input/output* port. This means that places A, D, 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 it will export tokens via the output port A. An input/output port is a port through which the module can both import and export tokens. In CPN Tools, port places can be recognised by the rectangular port-type tags 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 only relevant to the Sender module itself. The Sender module is identical to the sender part of Fig. 1 except that the colour set of the places A and D are now `PACKET` and that we use the constructors `Data` and `Ack` in the arc expressions on the surrounding arcs of these places.

Figure 8 shows the Receiver module. It has an input port B, an output port C, an input/output port DataReceived, and an internal place NextRec.

Figure 9 shows the Network module. 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. The Network module has two *substitution transitions* (drawn as rectangular boxes with double lines), TransmitData and TransmitAck. The basic

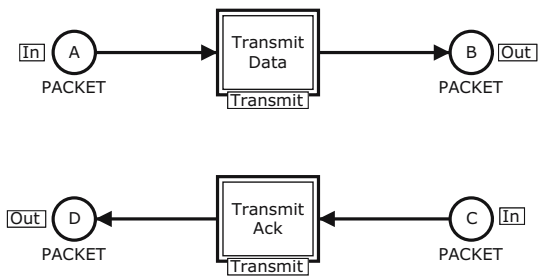



**Fig. 9** Module for network

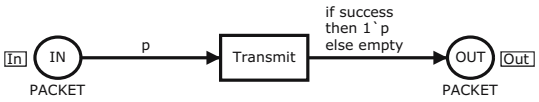



**Fig. 10** Module for packet transmission

idea in hierarchical models is to associate a module with each substitution transition. When a module is associated with a substitution transition it is said to be a *submodule*. In CPN Tools, substitution transitions can be recognised by the double boxes and rectangular submodule tags positioned next to them. The tag contains the name of the submodule 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 similar way as the implementation of a procedure provides a more detailed view of the effect of a procedure call. In Fig. 9, both substitution transitions have the Transmit module as their associated submodule.

The Transmit module is shown in Fig. 10. 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`.

To tie the modules together, we use the Protocol module shown in Fig. 11. It represents a more abstract view of the (entire) protocol system. The substitution transition Sender has the Sender module from Fig. 7 as its associated submodule, Network has the Network module from Fig. 9 as its associated submodule, and Receiver has the Receiver module from Fig. 8 as its associated submodule. In the Protocol module, we can see that the Sender, Network, and Receiver modules 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 modules do. In Fig. 11 each substitution transition has the same name as its submodule, but this is not required.

The input places of substitution transitions are called *input sockets*, while the output places are called *output sockets*. This means that in the Protocol module A is an output socket for the substitution transition Sender, and an input socket for the substitution transition Network. 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, we need to tell how the interface of each submodule is related to the interface of its substitution transition. This is done by means of a *port assignment*, which maps the port places of the submodule to the socket places of the substitution transition. Input ports are assigned to input sockets, output ports to output sockets, and input/output ports to input/output sockets. For the substitution transitions and associated submodules in Fig. 11, each port has the same name as the socket to which it is assigned, but this is not required.

When a port is assigned to a socket, the two places constitute two different views of a single place. This means that the port and socket place always share the same marking and hence conceptually become the same *compound place*. Figures 12 and 13 show the marking of the Sender and Network modules after an occurrence of the SendPacket transition in the initial marking.

When transition SendPacket occurs, it adds a token to the output port A in the Sender module (see Fig. 12). This port place is assigned to the output socket A of the substitution transition Sender in the Protocol module (see Fig. 11). Hence, the new token will also appear at place A in the Protocol module. This place is also an

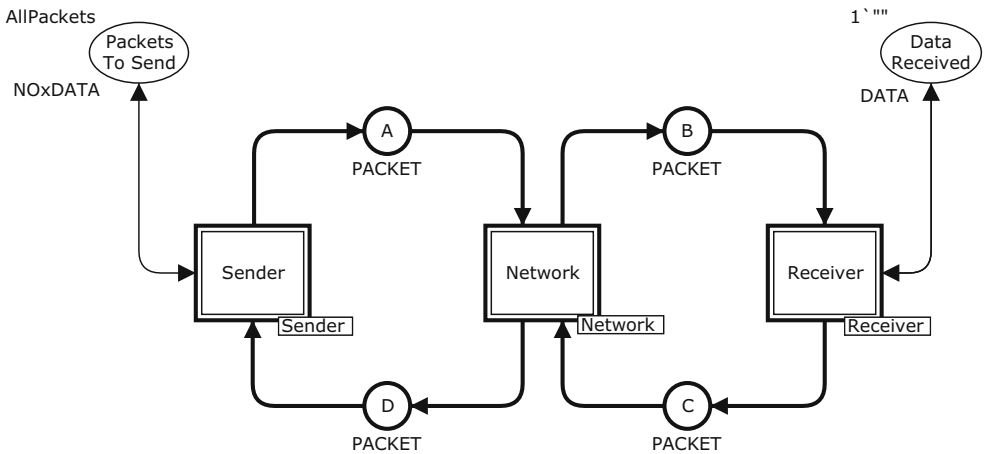



**Fig. 11** Protocol module—top-level module of the hierarchical protocol model

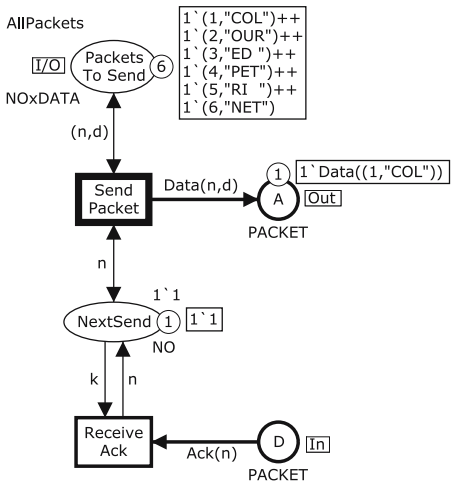



**Fig. 12** Marking of Sender module—after occurrence of SendPacket

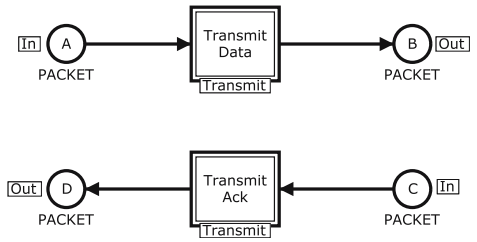



**Fig. 13** Marking of Network module—after occurrence of SendPacket

input socket for the substitution transition Network and has the port place A in the Network module (see Fig. 13) assigned to it. Hence, we conclude that 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.

Above we have seen that two related port and socket places constitute different views of a single compound 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 identical initial markings. It should be noted that substitution transitions do not have guards, and arcs connected to substitution transitions do not have arc expressions. 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 three levels of abstraction. The highest abstraction level is the Protocol module, while the lowest abstraction level is the Sender, Transmit, and Receiver modules, and the Network module is in between. In general, there can be an arbitrary number of abstraction levels. CPN models of larger systems typically have up to ten abstraction levels.

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

Each instance of a module has its own marking. This means that the marking of the instance of the Transmit module corresponding to the TransmitData substitution transition is independent of the marking of the instance of the Transmit module corresponding to the TransmitAck substitution transition. The marking of each instance of a port place matches the tokens present on the corresponding socket place of the associated substitution transition. CPN tools automatically instantiates the appropriate number of instances of each module and associates these instances with substitution transitions.

Above we have seen 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 a number of places (which may belong to different modules) to be glued together into one compound place across the hierarchical structure of the model.

### 2.5 Modelling of time

We now describe how timing information can be added to CPN models. This will allow us to evaluate how efficiently a system performs its operations, and it also allows us to model and validate real-time systems, i.e., systems where the correctness of the system relies on the proper timing of the events. With a timed CPN model we may calculate performance measures, such as maximum queue lengths, mean waiting times, and throughput. This will be illustrated in Sect. 6. We may also, e.g., verify whether a real-time system meets the required deadlines. We present the time concept of CP-nets using the non-hierarchical variant of the simple protocol. The

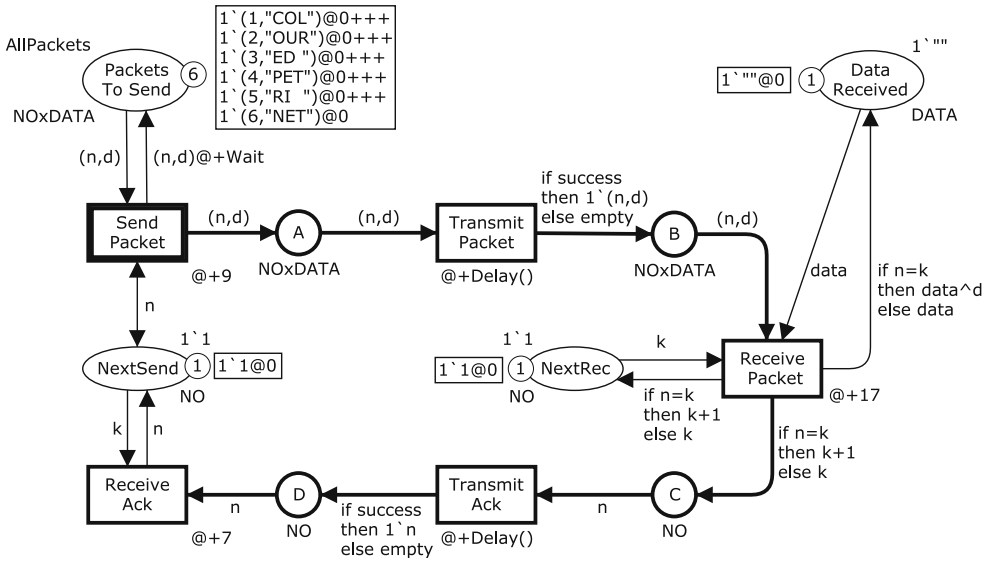



**Fig. 14** Timed CPN model for the protocol in the initial marking $M_0$

timing constructs apply also to hierarchical CPN models, and CPN Tools supports simulation and analysis of timed hierarchical CP-nets.

Now let us look at Fig. 14 which contains a timed CPN model for the simple protocol. It is easy to see that the CPN model is very closely related to the untimed CPN model in Fig. 1.

The main difference between timed and untimed CPN models is that the tokens in a timed CPN model—in addition to the token colour—can carry a second value called a *time stamp*. This means that the marking of a place where the tokens carry a time stamp is now a *timed multi-set* specifying the elements in the multi-set together with their number of appearances and their time stamps. Furthermore, the CPN model has a *global clock* representing *model time*. The distribution of tokens on the places together with their time stamps and the value of the global clock is called a *timed marking*. In a hierarchical timed CPN model there is a single global clock that is shared among all the modules.

In general, a time stamp can be a non-negative integer or real. In the current implementation of CPN Tools, only non-negative integers are supported. The time stamp tells us the time at which the token is *ready* to be used, i.e., the time at which it can be removed from the place by an occurring transition. The tokens on a place will carry a time stamp if the colour set of the place is timed. A colour set is declared to be timed using the CPN ML keyword `timed`. Figure 15 shows the colour set declarations for the timed model. It can be seen that all colour sets, except `BOOL` are defined to be *timed colour sets*.

The execution of a timed CPN model is controlled by the global clock, and it works in a similar way as

```
colset NO      = int timed;
colset DATA    = string timed;
colset NOxDATA = product NO * DATA timed;
colset BOOL    = bool;
```

**Fig. 15** Colour sets for the timed CPN model in Fig. 14

the event queue found in most simulation engines for discrete-event simulation. The model remains at a given model time as long as there are binding elements that are *colour enabled* (i.e., have the needed input tokens) and are *ready* for execution (i.e., the required tokens have time stamps which are less than or equal to the current value of the global clock). Hence, in a timed CPN model an enabled binding element must be both colour enabled and ready in order to be able to occur. When there is no longer such a binding element to be executed, the clock is advanced to the earliest 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 of time). As for untimed CPN models, we may have conflicts and concurrency between binding elements (and binding elements may be concurrent to themselves)—but only if the binding elements are ready to be executed at the same time.

Consider now the initial marking of the timed CPN model for the protocol shown in Fig. 14. The colours of the tokens are the same as in the initial marking of the untimed CPN model of the protocol, but now the tokens carry time stamps. As an example, the initial marking of the place PacketsToSend is:

```
1`(1,"COL")@0 +++
1`(2,"OUR")@0 +++
```

```
1`(3,"ED ")@0 +++
1`(4,"PET")@0 +++
1`(5,"RI ")@0 +++
1`(6,"NET")@0
```

The time stamp of tokens are written after the `@` symbol which is pronounced as "at". In this case, all tokens carry the time stamp 0. The `+++` operator takes two timed multi-sets as arguments and returns their union. All other tokens in the initial marking also carry the time stamp `0`. The value of the global clock in the initial marking is also 0. The initial marking of all places are specified as an (untimed) multi-set. CPN Tools will automatically attach the time stamp `0` if the initial marking inscription of a place with a timed colour set does not explicitly specify the time stamps of the tokens.

Consider the transition SendPacket and the binding ⟨`n=1, d="COL"`⟩ in Fig. 14. To occur, this binding needs the presence of a token with colour `1` on place NextSend and the presence of a token with colour `(1,"COL")` on place PacketsToSend. This is determined by the input arc expressions by means of the enabling rule explained in Sect. 2.2. We see that the two tokens that are needed by SendPacket exist on the input places and that both of them carry the time stamp `0`, which means that they can be used at time `0`. Hence, the transition can occur at time `0`. When the transition occurs, it removes the two tokens from the input places and adds a token to each of the three output places. The colours of these tokens are determined by the output arc expressions by means of the occurrence rule explained in Sect. 2.2. However, it is now also necessary to calculate the time stamps to be given to the three output tokens. This is done by using *time delay inscriptions* attached to the transition and/or to the individual output arcs. Time delays inscribed on the transition apply to all output tokens created by that transition, while time delays inscribed on an output arc only apply to tokens created at that arc. In Fig. 14 we have attached a time delay inscription `@+9` to the SendPacket transition, and a time delay inscription `@+Wait` to the outgoing arc to PacketsToSend, where `Wait` is a constant defined as:

```
val Wait = 100;
```

The arc expressions on the output arcs to the places A and NextSend have no separate time delays. The time stamp given to the tokens created on an output arc is the value of the global clock plus the result of evaluating the time delay inscription of the transition plus the result of evaluating the time delay inscription of the arc. Hence, we conclude that the tokens added to the places NextSend and A get the time stamp:

$$0 + 9 + 0 = 9 \tag{1}$$

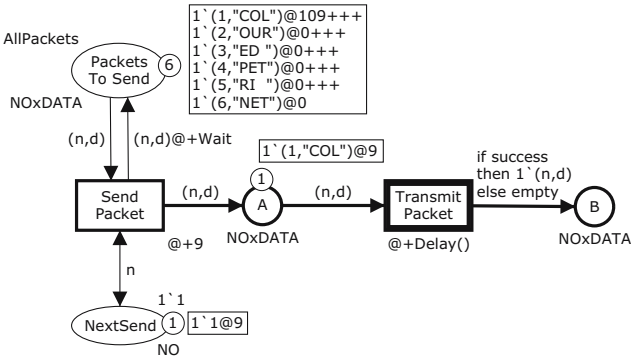



**Fig. 16** Marking $M_1$ reached when SendPacket occurs at time 0 in $M_0$

The first 0 is the time at which the transition occurs as given by the global clock, 9 is the time delay inscribed on the transition, and the second 0 is the time delay on the output arc (since there is no time delay on the output arc). Intuitively this models that the execution of the send packet operation has a duration of 9 time units.

The arc expression on the output arc to place PacketsToSend has a separate time delay: `@+Wait`. This means that the token added to PacketsToSend gets the time stamp:

$$0 + 9 + 100 = 109 \tag{2}$$

The 0 is the time at which the transition occurs, 9 is the time delay inscribed on the transition, while 100 is time delay inscribed on the output arc. Intuitively, this represents the fact that we do not want to resend data packet number 1 until time 109, i.e., until 100 time units after the end of the previous send operation. This is achieved by giving the token for data packet number 1 the time stamp 109—thus making it unavailable until that moment of time. However, it should be noticed that data packet number 2 still has time stamp 0. Hence, it will be immediately possible to transmit this data packet, if an acknowledgement arrives before time 109. When SendPacket occurs at time 0, we reach the marking $M_1$ which is partially shown in Fig. 16.

In marking $M_1$ there are three binding elements that have the needed tokens on their input places:

$$\begin{aligned} \text{SP} &= (\text{SendPacket}, \langle \texttt{n=1, d="COL"} \rangle) \\ \text{TP}_+ &= (\text{TransmitPacket}, \\ &\quad \langle \texttt{n=1, d="COL", success=true} \rangle) \\ \text{TP}_- &= (\text{TransmitPacket}, \\ &\quad \langle \texttt{n=1, d="COL", success=false} \rangle) \end{aligned}$$

SP can occur at time 109 (since it needs a token with time stamp 109 and a token with time stamp 9). However, $\text{TP}_+$ and $\text{TP}_-$ can occur already at time 9 (because they need a token with time stamp 9). Since $\text{TP}_+$ and

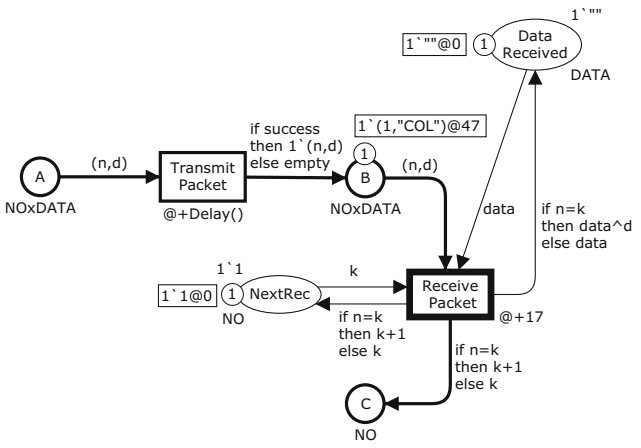



**Fig. 17** Marking $M_2$ reached when TransmitPacket occurs at time 9 in $M_1$

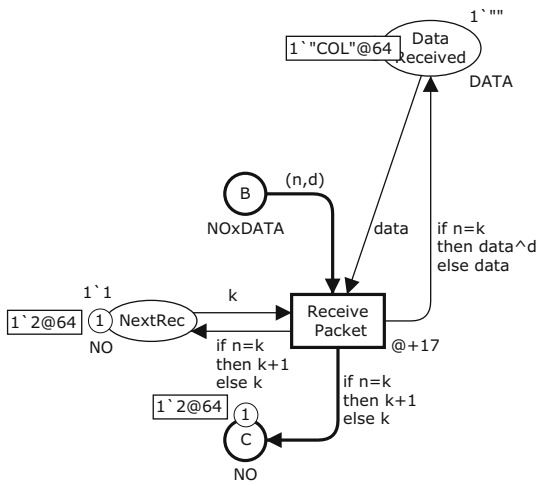



**Fig. 18** Marking $M_3$ reached when ReceivePacket occurs at time 47 in $M_2$

TP$_-$ are the first binding elements that are ready to occur, one of these will be chosen (the two binding elements are in conflict with each other) and it will occur as soon as possible, i.e., at time 9. Let us assume that TP$_+$ is chosen to occur. It will remove the token from place A and add a token to place B. The time stamp of this token will be the sum of the time at which the transition occurs (9) and the value obtained by evaluating the time delay expression `@+Delay()` inscribed on the transition. The function `Delay` takes a unit (`()`) as argument and is defined as follows:

```
fun Delay() = discrete(25,75);
```

The function `discrete` is a predefined function providing a discrete uniform distribution over the closed interval specified by its arguments. This means that a call `Delay()` returns an integer from the interval [25,75] and that all numbers in the interval have the same probability of being chosen. Intuitively, this represents that the time needed to transmit a packet over the network may vary between 25 and 75 time units, e.g., due to the load on the network. Let us assume that `Delay()` evaluates to 38. Then we reach the marking $M_2$ which is partially shown in Fig. 17. The above illustrates how random functions can be used to give time stamps to tokens.

In marking $M_2$ there are two binding elements that have the needed tokens on their input places:

SP = (SendPacket, ⟨`n=1, d="COL"`⟩)
RP = (ReceivePacket, ⟨`n=1, d="COL", k=1, data=""`⟩)

As before, SP can occur at time 109. However, RP can occur already at time 47 (because it needs a token with time stamp 47 and two tokens with time stamp 0). Hence RP will be chosen and we will reach the marking $M_3$ which is partially shown in Fig. 18.

In marking $M_3$ there are three binding elements that have the needed tokens on their input places:

SP = (SendPacket, ⟨`n=1, d="COL"`⟩)
TA$_+$ = (TransmitAck, ⟨`n=2, success=true`⟩)
TA$_-$ = (TransmitAck, ⟨`n=2, success=false`⟩)

SP can occur at time 109. However, TA$_+$ and TA$_-$ can occur already at time 64 (because they need a token with time stamp 64). Hence TA$_+$ or TA$_-$ will be chosen. The transmission and reception of an acknowledgement are similar as in the untimed CPN model and hence we will not explain them in further detail.

In the timed CPN model considered above, all tokens carry a time stamp since all colour sets of the places were declared to be timed. However, this is generally not the case. The modeller is allowed to specify whether each individual type (colour set) is timed or not. The tokens of timed colour sets carry time stamps while the tokens of untimed colour sets do not. Tokens without time stamps are always ready to participate in occurrences of binding elements.

Consider the token on place NextSend. If the token carries a time stamp, it is not possible for transitions SendPacket and ReceiveAck to occur at the same model time. This is due to the fact that when one of these transitions occurs, the time stamp of the token is increased, which eliminates the possibility that the other transition can occur at the same model time. This could represent a sender that uses a single thread for sending data packets and receiving acknowledgements. On the other hand, if the token on NextSend did not carry a time stamp, then these two transitions could occur at the same model

time. This could be used to model a sender with separate threads for sending and receiving.

It should be noted that the possible occurrence sequences of a timed CPN model always form a subset of the occurrence sequences for the underlying untimed CPN model, i.e., the model in which we remove all time delay inscriptions (and all time stamps). This means that the time delay inscriptions merely enforce a set of additional constraints on the execution of the CPN model—forcing colour-enabled binding elements to be chosen when they are ready. Turning an untimed CPN model into a timed model cannot create new behaviour in the form of new occurrence sequences. As a consequence it is often a useful modelling strategy to start by investigating the functional/logical properties by means of an untimed CPN model. Then the timing related to events can be considered afterwards.

The occurrence of a transition is instantaneous, i.e., takes no time. However, as shown in the protocol example above, it is easy to model that some actions in a system have a non-zero duration. This is done by giving the output tokens created by the corresponding transition time stamps that prevent the tokens from being used until the time at which the action has finished. As an example, TransmitPacket cannot occur until 9 time units after the occurrence of SendPacket—this represents that the action to send a data packet takes 9 time units. An alternative approach would have been to allow the occurrence of a transition to have a non-zero duration. We could then remove the input tokens at the moment where the occurrence begins and add the output tokens when the occurrence ends. However, such an approach would make the relationship between a timed CPN model and its underlying untimed CPN model much more complex. There would then be a lot of reachable markings in the timed CPN model which are unreachable in the untimed CPN model—because they correspond to situations where one or more transitions are partway through their occurrence (having removed tokens from the input places, but not yet having added tokens to the output places).

## 3 Construction of CPN models

This section introduces the GUI of CPN Tools and the tools and features in CPN Tools for constructing CPN models. There are tools for creating declarations, net structure, inscriptions, and hierarchical models. Additional tools are available to improve the readability of a model, including tools for changing line colours and widths, and for aligning elements.

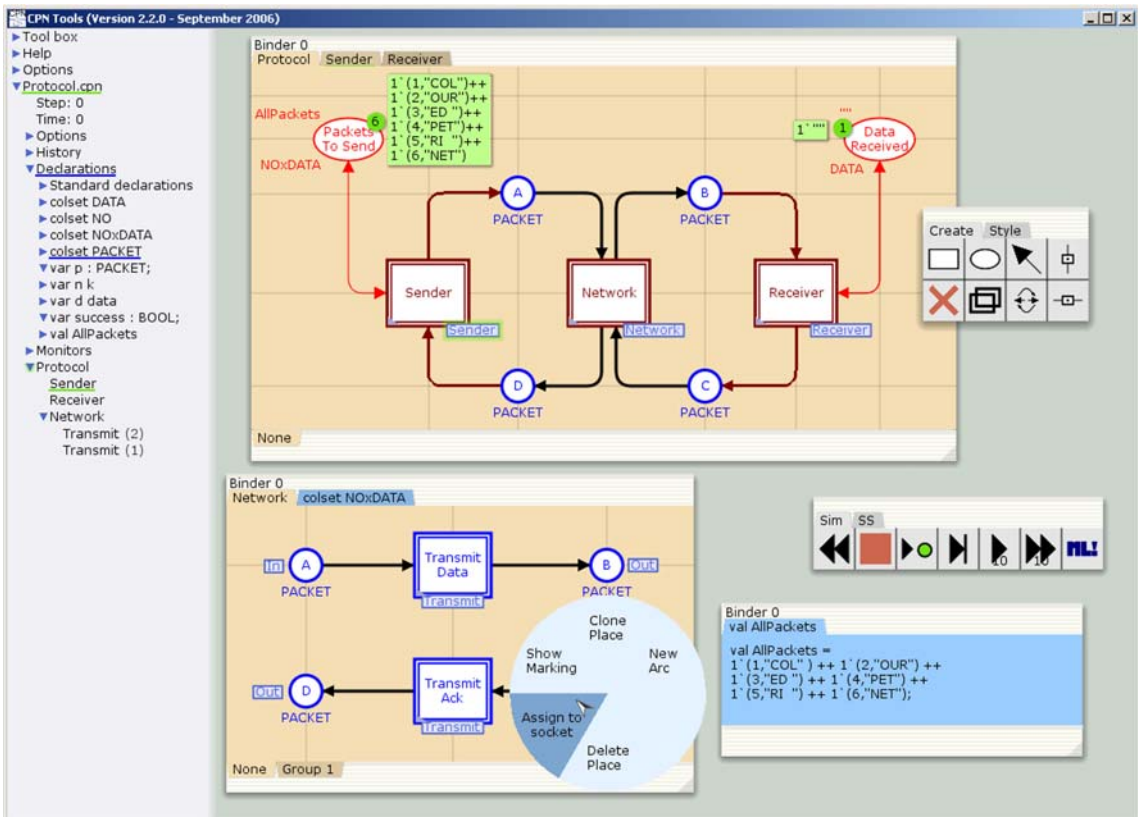



**Fig. 19** Screenshot of CPN Tools

### 3.1 Overview of the GUI

Figure 19 shows a screenshot of CPN Tools. The rectangular area to the left is the *index*. It includes the Tool box which contains many of the tools that are available for manipulating the declarations and modules that constitute the CPN model. The Tool box includes tools for creating and cloning (i.e., copying) the basic elements of CPN models. Additionally, it contains a wide selection of tools for manipulating the graphical layout and the appearance of the objects in the CPN model. The latter set of tools is very important in order to be able to create readable and graphically appealing CPN models. The index also contains a *model overview* for each of the models that are open. A model overview shows a variety of information including the name of the model, the Declarations for the model, the modules of the model, and the hierarchical structure of the model. The History for a model shows a list of the operations that have been performed on the model. This list contains only those operations that can be undone and redone. Many operations can be undone, including editing operations, while others cannot be undone, such as executing simulation steps.

A small triangle to the left of an entry in the index indicates either that the entry contains subentries or that the entry can be expanded to show more information for the entry. Clicking on a small triangle will open and close the corresponding index entry. A triangle that points to the right indicates a closed entry, while a triangle pointing downwards indicates an open entry. A subentry in the index is indented to the right of its parent entry. For example, in Fig. 19 the Tool box entry has been opened to show its nine subentries (from Auxiliary to View), and the declaration for the variable `success` has been opened to show the type of the variable.

The remaining part of the screen is the *workspace*, which in this case contains five *binders* (the rectangular windows) and a circular *marking menu*. Each binder holds a number of items which can be accessed by clicking the tabs at the top of the binder (only one item is visible at a time in each binder). There are two kinds of binders. One kind contains the elements of the CPN model, i.e., the modules and declarations. The other kind contains *tool palettes* which contain the tools which the user applies to construct and manipulate CPN models. A tool in a tool palette can be picked up by clicking on the appropriate *tool cell*. After picking up a tool, the mouse cursor will change to show which tool has been picked up. A tool that has been picked up is applied by clicking on an appropriate target. In Fig. 19 one binder contains three modules named Protocol, Sender, and Receiver, another binder contains a single module named Network together with the declaration of the colour set `NOxDATA`, and a third binder contains the declaration of the constant `AllPackets`. The two remaining binders contain four different tool palettes to Create elements, change their Style, perform simulations (Sim), and construct state spaces (SS). Some items can be dragged from the index to the binders, and from one binder to another binder of the same kind. It is possible to position the same item in two different binders, e.g., to view a module in two different zoom factors.

A circular marking menu has been popped up on top of the bottom-left binder. Marking menus are contextual menus that make it possible to select between some of the possible operations for a given object. In this example it shows some of the operations that can be performed on a port place. Most of the tools that are available in a marking menu are also available in a tool palette, and vice versa.

### 3.2 Construction of model elements

The New Net and Load Net tools are used to create a new model and load an existing model, respectively. These tools can be found in the Net tool palette or in the marking menu for the workspace. When a model is created or loaded, its model overview will be added to the index.

The colour sets, variables, and functions that are used in inscriptions must be defined in *Declarations* for the model. The Declarations that belong to a model can be seen in the index in the model overview. New declarations are added using the New Declaration tool which can be found in relevant marking menus. Declarations can be grouped in *declaration blocks*. In Fig. 19, many of the declarations for the Protocol.cpn model can be seen, and the Standard declarations entry is a declaration block that contains a number of default declarations that are included when a new model is created. Keyboard shortcuts can be used to jump from one declaration to the next and to add a new declaration after a declaration that is being edited. Declarations can be viewed and edited in the index and in *declaration sheets* in binders.

The tools for creating net structure are found in the Create palette which is shown in Fig. 20. The available tools (from left to right and top to bottom) are:

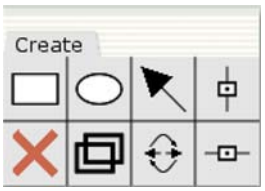



**Fig. 20** Tools for creating net structure in the Create palette


**Fig. 21** Tool options in the index for the Create Place tool

- Create a transition.
- Create a place.
- Create an arc.
- Create a vertical guideline.
- Delete an element.
- Clone, i.e., copy, an element.
- Change direction of an arc.
- Create a horizontal guideline.

The tools from the Create palette can also be found in marking menus, with the exception of the tools for creating guidelines which will be explained below. Each tool can be applied to certain kinds of targets. For example, the tools for creating places and transitions can be applied to modules in binders, while the tool for creating an arc must be applied first to a place (transition), and then to a transition (place).

A number of tools have *options* that affect the behaviour of the tool. These tool options can be changed, and they are accessible in the index and via tool cells in palettes. Figure 21 shows the tool options for the Create Place tool. With these options it is possible to change the default width and height of new places.

Inscriptions must be added to nodes and arcs. After creating a new place, transition or arc, text-edit mode will be entered, and it will be possible to add the first inscription to the element. Arcs have only one inscription, while places and transitions have several kinds of inscriptions. The TAB key is used to cycle between the different inscriptions for a node.

There are a number of different tools and features that can be used to improve the layout, and therefore the readability, of a model. The tools in the Style palette are used to change the graphical attributes of model objects. There are tools for changing line and fill colour, line widths, and arrowhead size.

Nodes and arc bend points can be aligned using *guidelines* and automatic snap-to-alignment features. Objects will automatically be snapped to vertical or horizontal alignment whenever possible. For example, if an arc is added between two nodes that are almost aligned, then they will be moved into alignment with each other, or if a node that is attached to another node via an arc is moved, then it will be snapped into alignment with the other node whenever it is moved close enough into alignment. Similarly, inscriptions can be snapped to various different snap points for their parent objects. For example, transitions have snap points at each corner as well as at the middle of each side, and inscriptions can be snapped to these points. Several different elements can be aligned by snapping them to guidelines. All of the elements that are snapped to a guideline will be moved when the guideline is moved.

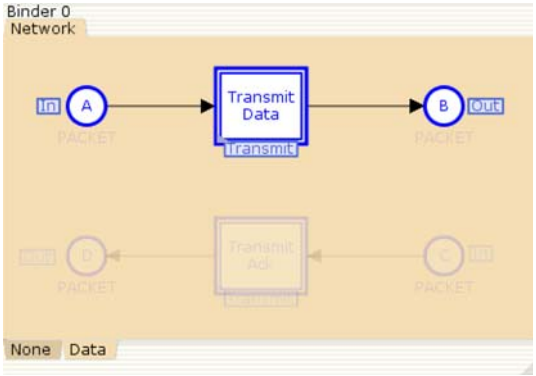



**Fig. 22** A group of elements in a module

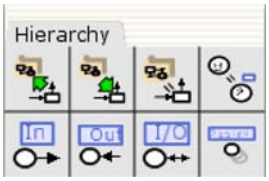
New model elements do not need to be created from scratch because they can be created by cloning, i.e., copying, existing elements. Many kinds of elements can be cloned, including inscriptions, nodes together with all their inscriptions and modules. When an object is cloned, all of its graphical attributes, such as size and line colour, are cloned as well. Objects can be cloned within the same model or between two models (more than one model may be open at a time in CPN Tools). It is also possible to clone just the graphical attributes from one object to another.

It can be tedious to perform the same operation on a number of individual elements, such as changing the colour of model elements, or moving and realigning model elements. In CPN Tools, it is possible to create *groups*, and to perform operations on the elements in a group. *Group tabs* at the bottom left of a module in a binder indicate which groups are defined for the module. The elements that are not in the group are dimmed, as shown in Fig. 22. If a tool is applied to an element in a group, the tool will be applied to all of the other relevant elements in the group. For example, changing the colour of any element in the group will change the colour of all of the elements in the group, while changing the direction of an arc will only affect the arcs in the group. Regular groups can only contain elements from the same module, but *global groups* can contain elements from any module in the model.

### 3.3 Construction of hierarchical models

The tools for creating hierarchical nets are found in the Hierarchy palette which is shown in Fig. 23. The available tools (from left to right and top to bottom) are:

**Fig. 23** The hierarchy palette



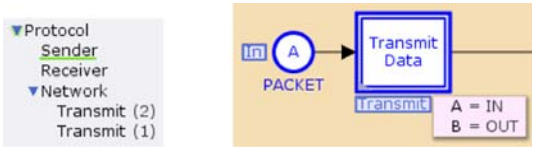




**Fig. 24** Overview of hierarchical relationships

- Move a transition or group to a new submodule.
- Replace a substitution transition with its submodule.
- Assign a submodule to a substitution transition.
- Assign a port to a socket.
- Set port type to input.
- Set port type to output.
- Set port type to input/output.
- Assign a place to a fusion set.

These tools support both top-down and bottom-up construction of CPN models. Supporting a top-down approach, the Move to Submodule tool will move a group of elements from one module to a new submodule, create a substitution transition with appropriate arcs in the original module, create appropriate port places in the submodule, and assign ports to sockets. The Assign Submodule tool supports a bottom-up approach, in that it will assign an existing module to be the submodule associated with an existing (substitution) transition, and it will automatically assign port places to sockets whenever possible. In addition, the Clone tool can also be used to clone hierarchy elements, such as port-type tags, or even a substitution transition and all of its submodules.

Information about hierarchical relationships can be seen in modules and the index, as shown in Fig. 24. The left-hand side of the figure shows the names of the modules from a model overview. A small triangle next to a module name indicates that it has submodules, and the submodules of the module are listed below and indented to the right of the supermodule. In the example, the Protocol module is the top-level module, and it has three submodules. One of these submodules (Network) also has two submodules. A number in parentheses after the module name indicates that there are multiple instances, while a missing number indicates that there is only one instance of that module in the model.

Instantiation of modules is handled fully automatically by CPN Tools, and the user can access the individual instances of modules. It should be noted that instantiation of modules is done 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.

The right-hand side of Fig. 24 shows hierarchical information that can be seen in a module. The small tag immediately below TransmitData indicates that it is a substitution transition, and that it is associated with the submodule Transmit. The list to the lower right of TransmitData shows the assignments between its sockets and the port places in submodule Transmit. The list is opened and closed by clicking on the small triangle in the lower left-hand corner of the substitution transition. Place A is a socket for TransmitData, but the small tag next to the place indicates that it is also an input port. The tags associated with port places, substitution transitions, and fusion places are collectively referred to as *hierarchy tags*. Marking menus for some hierarchy tags contain operations for opening and showing sub- and supermodules in binders. This makes it possible to navigate efficiently.

### 3.4 Syntax check and code generation

CPN Tools performs syntax and type checking, and simulation code generation. Error messages are provided to the user in a contextual manner next to the object causing the error. Figure 25 shows an example of an error message for a place inscription.

The syntax check and code generation are incremental and performed in parallel with editing. This means that it is possible to execute parts of a CPN model even if the model is not complete, and it means that when parts of a CPN model are modified, syntax check and code generation are only performed on the elements that depend on the parts that were modified. Some elements will not be checked until they have enough infor-

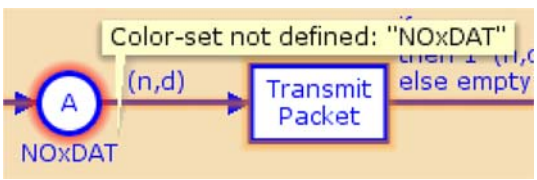



**Fig. 25** Contextual error message

mation to be syntactically correct. For example, a place will not be checked until it has a colour set inscription, and a transition will not be checked until all of its surrounding places can be checked, and all of its surrounding arcs have arc inscriptions.

The main outcome of the code generation is the *simulation code*. The simulation code contains the functions for inferring the set of enabled binding elements in a given marking of the CPN model, and for computing the marking reached after an enabled binding element occurs.

### 3.5 Graphical feedback and help

CPN Tools uses several kinds of graphical feedback to provide information when editing and analysing a CPN model. Help information is also available. Colour-coded *auras* are used to highlight objects with particular characteristics or to indicate different kinds of relationships between objects. For example, bright red auras indicate errors. Auras are associated with places, transitions, arcs, inscriptions, declarations, module tabs, and index entries, such as module names. Auras are propagated to parent-like objects whenever possible. In Fig. 25 the place has an inscription with an error, and the place will therefore have a red aura. The red aura will be propagated to the tab for the module in the binder, to the name of the module in the index, and to the model name. Error auras are always shown, and they will be removed only if the error is fixed. Other kinds of auras appear only when the cursor hovers over a particular kind of object. For example, dark blue auras indicate dependencies between declarations and model objects, and they appear only when the cursor hovers over a declaration or a node. In Fig. 19, one of the declarations has an aura (in dark blue) because the cursor has been used to open a marking menu for a place that is dependent on that declaration.

A *speech bubble* is a yellow rectangle that provides context-sensitive information, such as an error message. Some speech bubbles appear automatically, while others appear after a slight delay when the cursor is moved over an appropriate object. For example, moving the cursor over a model name will cause a speech bubble containing the full path to the model to appear. On the other hand, speech bubbles with error messages for net structure, like the one in Fig. 25 appear automatically.

*Status bubbles* are colour-coded bubbles that occasionally appear at the bottom of the index. A speech bubble is often associated with the status bubble, as shown in Fig. 26. It may be necessary to move the cursor over a status bubble to see the corresponding speech


**Fig. 26** Status bubble at bottom of index

bubble. Green indicates that an operation completed successfully, red indicates an error, and light purple indicates that a time-consuming operation, such as a long simulation, is being performed.

Detailed help information can be accessed in a number of ways. Dragging the Help index entry to the workspace will open the main page for the offline help in a Web browser. A brief tool tip describing the functionality of the tool will appear if the cursor hovers over a tool cell in a tool palette. Marking menus for tool cells in palettes and for palette tabs in binders contain tools for opening a relevant help page in a browser.

## 4 Simulation

The simulator of CPN Tools exploits a number of advanced data structures and algorithms for efficient simulation of large hierarchical CPN models [28]. The simulator exploits the locality property of Petri nets, which ensures that the occurrence of a transition only affects its immediate surroundings. This ensures that the number of steps executed per second in a simulation is independent of the number of places and transitions in the CPN model. This means that simulation scales to large CPN models.

The CPN Tools simulator only executes steps consisting of a single binding element. The marking resulting from the occurrence of an enabled step with multiple binding elements is the same as letting the binding elements in the step occur one after each other in some arbitrary order. Hence, markings that can be reached via occurrence sequences consisting of steps with multiple binding elements can also be reached via occurrence sequences with steps consisting of a single binding element.

CPN Tools uses graphical *simulation feedback*, as shown in Fig. 27, to provide information about the markings that are reached and the binding elements that are enabled and occur during a simulation. A small circle next to a place indicates the number of tokens on the place in a marking, and a box next to the circle shows the colours of the tokens. In Fig. 27, place DataReceived contains one token with value `"COL"`. Green auras indicate enabled transitions, and the auras can be found on

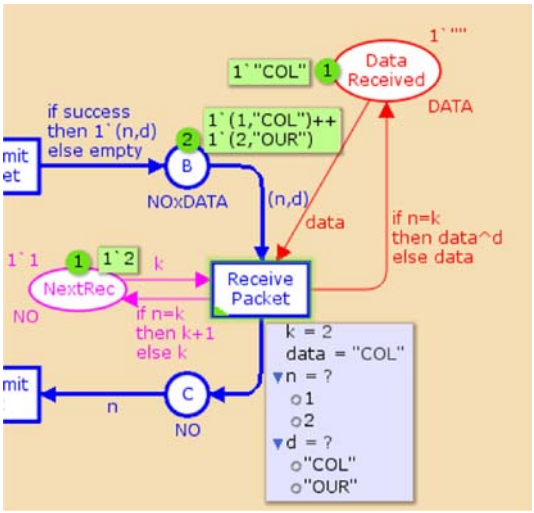



**Fig. 27** Simulation feedback in CPN Tools


**Fig. 28** Simulation tool palette

transitions, on module names in binders and the index, and on submodule tags. The box below ReceivePacket will be discussed below.

Many of the tools that are available for simulating CPN models can be found in the simulation tool palette shown in Fig. 28. A VCR (Video Cassette Recorder) metaphor is used for the graphical symbols representing the simulation tools. The simulation tools can be picked up with the mouse cursor and applied to the CPN model. The available tools (from left to right) are:

- Return to the initial marking.
- Stop an ongoing animated automatic simulation.
- Execute a single transition with a manually chosen binding.
- Execute a single transition with a random binding.
- Execute an animated automatic simulation, i.e., execute an occurrence sequence with randomly chosen binding elements and display the current marking between each step.
- Execute a fast automatic simulation, i.e., execute an occurrence sequence with randomly chosen binding elements without displaying the current marking between each step.
- Evaluate a CPN ML expression (to be explained in Sect. 6.6).

### 4.1 Interactive and automatic simulations

When a CPN model is simulated in *interactive mode*, the simulator calculates the set of enabled transitions in each marking encountered. Then it is up to the user to choose between the enabled transitions and bindings. Figure 27 shows an example where the user is in the process of choosing between the enabled bindings of the ReceivePacket transition. The choice between the enabled bindings is done via the rectangular box opened next to the transition. It lists the variables of the transition and the values to which they can be bound in the current marking. In this case, the value `2` has already been bound to the variable `k` and the value `"COL"` has been bound to `data`. This is done automatically by the simulator since there is only one possible choice for these variables. The user still has a choice in binding values to the variables `n` and `d`. The user may also leave the choice to the simulator which uses a random number generator for this purpose. In the above case it suffices for the user to bind either `n` or `d` since the value bound to the other variable is then uniquely determined and will be automatically bound by the simulator.

After the chosen binding element has been executed, the marking and enabling information is updated and presented to the user, who either chooses a new enabled binding element or leaves the choice to the simulator, and so on. This means that it is the simulator that makes all the calculations (of the enabled binding elements and the effect of their occurrences), while it is the user who chooses between the different occurrence sequences (i.e., the different behavioural scenarios). If it is not necessary to choose a particular binding of variables, the user can use the Single Step tool to execute a single transition. The tool can be applied to different targets, and it will have different effects depending on the target. For example, if the tool is applied to an enabled transition, then that particular transition will occur, and if the tool is applied to a binder, then a randomly chosen enabled transition on a module in the binder will occur. An interactive simulation is by nature slow, since it takes time for the user to investigate the markings and enablings and to choose between them. This means that only a few steps can be executed each minute and the working style is very similar to single-step debugging known from conventional programming environments.

When a CPN model is simulated in *animated automatic mode*, the simulator calculates the set of enabled transitions in each marking encountered. The simulator also chooses between the enabled transitions and bindings. The simulator feedback is updated after each step in an animated automatic simulation. The Play tool, i.e., the third tool from the right in Fig. 28, has tool options

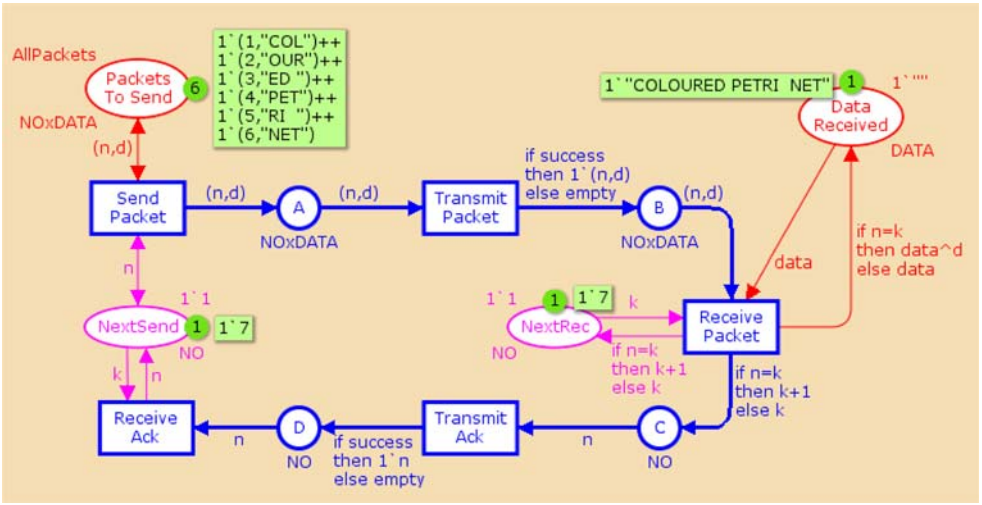



**Fig. 29** Dead marking $M_{\text{dead}}$ reached at the end of a simulation

which determine how many steps should be executed in an animated automatic simulation, and how long the simulator should pause between each step. The simulator will perform the specified number of steps, unless it reaches a state in which there are no more enabled transitions or until the Stop tool from the simulation palette is applied.

A CPN model can also be simulated in *fast automatic mode*. This kind of simulation is similar to a program execution, and a speed of several thousand steps per second is typical. Before the start of a fast automatic simulation, the user specifies one or more stop criteria, e.g., that 100,000 steps should occur or 50,000 units of model time should elapse. If one of the stop criteria becomes fulfilled or if there are no more enabled transitions, the simulation stops and the user can inspect the marking which has been reached. There are also a number of different ways in which the user can inspect the markings and the binding elements that occurred during the simulation. We shall briefly return to this at the end of this section.

A simulation of the protocol example may stop in the marking $M_{\text{dead}}$ shown in Fig. 29. This marking is a *dead marking* because there are no enabled transitions. Due to the non-determinism in the CPN model, we cannot guarantee that the dead marking will be reached since it is possible to keep losing packets and acknowledgements. However, if we reach a dead marking it will be the marking shown in Fig. 29. Here, we see that all six data packets have been received (in the correct order). The sender has stopped sending because NextSend has a token with colour 7 and there is no data packet with number 7. All the places A, B, C, and D connecting the network to the sender and receiver are empty. Finally, place NextRec has a token with colour 7. Hence, this marking represents the desired terminal state of the protocol system. By making a number of automatic simulations of the CPN model, it is possible to test that the simple protocol design appears to be correct. Conducting a set of automatic simulations does not guarantee that all possible executions of the protocol have been covered. Hence, we cannot in general use simulation to verify properties of the protocol, but it is a powerful technique for testing the protocol and for locating errors. In Sect. 5 we will introduce state space analysis which makes it possible to ensure that all possible executions are covered. This makes it possible to verify systems, i.e., prove that different behavioural properties are present or absent in a model.

As mentioned earlier in this section, the user may be interested in inspecting some of the markings and some of the binding elements that occurred during a simulation. A simple (and brute-force) way to do this is to inspect the *simulation report* which lists the steps that have occurred. Figure 30 shows the beginning of a simulation report for the hierarchical model from Sect. 2.4. We see the first three transitions that occurred. A simulation report specifies the names of the transitions that occur during a simulation, the module instances where the transitions are located, and the user determines if the report should specify the values bound to the variables of the occurring transitions. In this case, the SendPacket transition in instance 1 of the Sender module occurred in step 1, the Transmit transition in instance 2 of the Transmit module occurred in step 2, and the ReceivePacket transition in instance 1 of the Receiver module occurred in step 3. The number 0 following the step number specifies the model time at which the transition

```
1       0       SendPacket @ (1:Sender)
 - n = 1
 - d = "COL"
2       0       Transmit @ (2:Transmit)
 - p = Data(1,"COL")
 - success = true
3       0       ReceivePacket @ (1:Receiver)
 - n = 1
 - d = "COL"
 - k = 1
 - data = ""
```

**Fig. 30** Beginning of a simulation report

occurs. Since the hierarchical model of the simple protocol presented in Sect. 2.4 is untimed, all steps occur at time zero.

### 4.2 Simulation breakpoint monitors

Simple simulation tool options can specify that a simulation should stop after a certain number of steps or after a certain amount of model time has passed, but in many cases it can be useful to stop a simulation in a particular state or after a particular transition has fired. In CPN Tools, *monitors* can be used to examine the binding elements that occur and the markings that are reached during a simulation. Different kinds of monitors can be used for different purposes. *Breakpoint monitors* can be used to stop simulations when specific conditions are fulfilled. A *transition enabled monitor* is a standard breakpoint monitor that can be associated with a transition, and the monitor will stop a simulation when the transition is enabled (or disabled, as determined by an option for the monitor). Another standard breakpoint monitor can be used to stop a simulation when the marking of a particular place is empty (or not empty, as determined by an option for the monitor). A *generic breakpoint monitor* can be used to define a model-specific condition that will determine when a simulation should stop. It is then checked at certain steps in the simulation whether the condition is fulfilled.

Each monitor has *monitoring functions* that determine its functionality. For transition enabled monitors, the monitoring functions are hidden from the user. However, for generic breakpoint monitors, the monitoring function is accessible, and it must be modified by the user. When a new monitor is created, template code for the accessible monitoring functions is automatically generated. The template code must be modified to obtain the desired behaviour. This means that the user does not have to write monitoring functions from scratch.

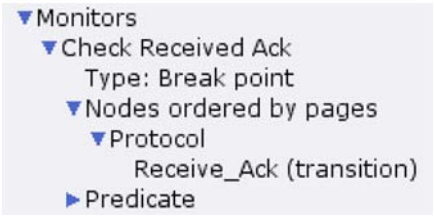



**Fig. 31** Overview of a monitor in the index

A monitoring function often consists of 5–10 lines of CPN ML code. Each monitor is associated with a group of nodes consisting of zero or more places and zero or more transitions in the model. A monitor can only examine the nodes with which it is associated. For a generic breakpoint monitor, the user must define a *predicate function* that determines when a simulation should stop. A predicate function will be called after certain steps in a simulation, and it should return the value `true` when the simulation should stop.

An example of a model-specific breakpoint monitor would be a monitor that will stop a simulation of the protocol if the sender receives an acknowledgement which is lower than the sequence number of the data packet that is currently being sent. In this case, the monitor is associated with the ReceiveAck transition only. The predicate function for this monitor will be invoked each time the ReceiveAck transition occurs and it will return true if the value bound to the variable `k` is greater than the value bound to the variable `n`.

A monitor is created by applying one of the tools from the Monitoring palette (not shown) to an appropriate target. The target may be a single place or transition, a group of nodes, a global group of nodes, or the name of the model. Note that a monitor may be associated with nodes from different modules. After a monitor has been created, it will be added to the index, where different kinds of information related to the monitor can be viewed and modified. Figure 31 shows the information that is added to the index when a generic breakpoint monitor is created for the ReceiveAck transition in the Protocol module. The monitor overview shows the user-specified name of the monitor (CheckReceivedAck), the type of the monitor ((generic) breakpoint), the nodes that the monitor is associated with, and the accessible monitoring functions. The predicate function for the CheckReceivedAck monitor looks as follows:

```
fun pred
  (Protocol'Receive_Ack (1, {k,n})) = n < k
```

Since this monitor is associated with only one transition, the predicate function can only examine binding elements for the ReceiveAck transition. The predicate

function will automatically be called in CPN Tools after the ReceiveAck transition occurs, and it will not be called after any of the other transitions in the model occur. The function inspects the values of the variables `n` and `k` and compares them as described above. For further details on implementing model-specific monitors we refer to [10].

It is also possible to disable monitors, which means that it is possible to ensure that a monitor will not be activated during a simulation without having to remove the monitor from the model.

## 5 State space analysis

Simulation can only be used to consider a finite number of executions of the model being analysed. This makes simulation suited for detecting errors and for obtaining increased confidence in the correctness of the model, and thereby the system. For the simple protocol we may conduct a number of simulations which show that the model of the protocol always seems to terminate in the desired state where all data packets have been received in the correct order. This makes it likely that the protocol works correctly, but it cannot be used to ensure this with 100% certainty since we cannot guarantee that the simulations cover all possible executions. Hence, after conducting a set of simulations, there may still exist executions of the model leading to a state where, e.g., the data packets are not received in the correct order.

Full state spaces represent all possible executions of the model being analysed. The basic idea of full state spaces is to calculate all reachable states (markings) and all state changes (occurring binding elements) of the CPN model and represent these in a directed graph where the nodes correspond to the set of reachable markings and the arcs correspond to occurring binding elements. The state space of a CPN model can be computed fully automatically and makes it possible to automatically *verify*, i.e., prove in the mathematical sense of the word that the model possesses a certain formally specified property. We present state spaces and behavioural properties using a non-hierarchical CPN model. However, full state spaces generalise to hierarchical and timed CPN models, and CPN Tools supports full state spaces for hierarchical and timed CPN models.

### 5.1 Revised model for state space analysis

To introduce state space analysis we consider the simple protocol from Fig. 1. Before we construct a state space for the model of the protocol, we will make a small modification. The CPN model in Fig. 1 has infinite occurrence sequences in which the transition SendPacket occurs an infinite number of times immediately after each other (retransmitting the first packet an infinite number of times). This means that there is an infinite number of reachable markings. To obtain a finite number of reachable markings, we limit the number of tokens which may *simultaneously* reside on the network buffer places A, B, C, and D. This is done by adding a new place Limit as shown in Fig. 32. It has the colour set `UNIT` defined as:

```
colset UNIT = unit;
```

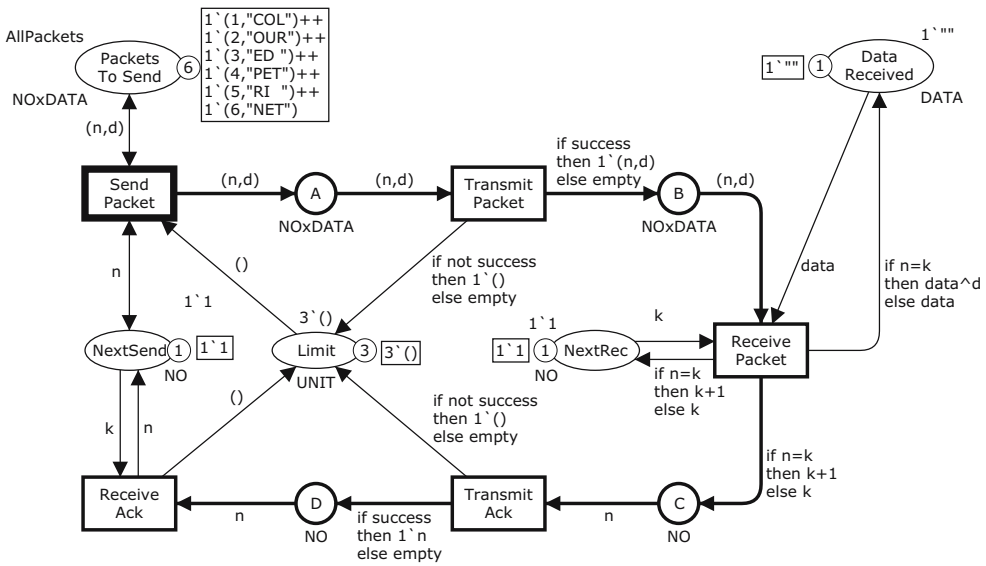



**Fig. 32** CPN model used for state space analysis

where `unit` is the basic CPN ML type containing the single value `()`. The initial marking of Limit is the multiset `3`()`. Tokens with the token colour `()` can be thought of as being "uncoloured" tokens where the value attached carries no information (since it can only take one possible value). A token is removed from place Limit each time a packet is sent to the network, and a token is added to place Limit, each time a packet is removed or lost from the network. This means that the total number of tokens on the five places A, B, C, D, and Limit is constant and identical to the number of tokens which Limit has in the initial marking.

Clearly, making this kind of modification changes the behaviour of the model, and it should be done with care. However, for models with very large state spaces, it is often useful to analyse the restricted behaviour of a model in order to increase our confidence in the correctness of the unrestricted model. For the protocol example, we have chosen to limit the number of tokens on the network buffer places to three tokens. This configuration allows packets to overtake each other, and it also allows for duplicate packets to be in the buffer places. However, it significantly limits how often packets overtake each other as well as the number of duplicate packets in buffer places. If analysis shows that the restricted model is correct, then this will increase our confidence in the fact that the unrestricted model is also correct. Similarly, if errors are found in the restricted model, then the same errors exist in the unrestricted model of the protocol. It is unlikely that we would obtain additional insights into the behaviour of the unrestricted model by increasing the number of tokens allowed in the buffer places in the restricted model. On the other hand, if we had limited the number of tokens in buffer places to just one, then it would no longer be possible for packets to overtake each other, nor would it be possible to have duplicate packets in the buffer places. In this case, the restriction is probably too radical, and the behaviour of the restricted model would be significantly different from the unrestricted model.

### 5.2 Full state spaces

A *full state space* is a directed graph, where there is a node for each reachable marking and an arc for each occurring binding element. There is an arc labelled with a binding element $(t, b)$ from a node representing a marking $M_1$ to a node representing a marking $M_2$ if and only if the binding $(t, b)$ is enabled in $M_1$ and the occurrence of $(t, b)$ in $M_1$ leads to the marking $M_2$.

Figure 33 shows an initial fragment of the state space of the CPN model in Fig. 32. This fragment has been created using the support for visualisation of state spaces in CPN Tools. Each node is inscribed with three integers. The topmost integer is the node number and the two integers separated by a colon give the number of predecessor and successor nodes. Node 1 corresponds to the initial marking, and the figure shows all markings reachable by the occurrence of at most three binding elements starting in the initial marking. The rectangular *node descriptor* associated with each node gives information about the marking of the individual places in the state represented by the node. The node descriptor lists the places which have a non-empty marking. We have omitted the marking of place PacketsToSend since this place always contains the six tokens corresponding to the data packets to be sent. The rectangular *arc descriptor* associated with each arc gives information about the corresponding binding element. The node and arc descriptors have a default content, but options are available for the user to control the contents of the descriptors.

In the initial marking only one binding element (SendPacket,⟨`n=1,d="COL"`⟩) is enabled and it leads to the marking which is identical to the initial marking except that there is now also a token with colour `(1,"COL")` on place A, and there is one less token on place Limit. In Fig. 33, this marking is represented by node 2. In the marking corresponding to node 2, we have three enabled binding elements:

$$
\begin{aligned}
\mathrm{SP} &= (\mathsf{SendPacket}, \langle \texttt{n=1}, \texttt{d="COL"} \rangle) \\
\mathrm{TP}_{+} &= (\mathsf{TransmitPacket}, \\
&\quad \langle \texttt{n=1}, \texttt{d="COL"}, \texttt{success=true} \rangle) \\
\mathrm{TP}_{-} &= (\mathsf{TransmitPacket}, \\
&\quad \langle \texttt{n=1}, \texttt{d="COL"}, \texttt{success=false} \rangle)
\end{aligned}
$$

and their occurrence lead to the markings represented by nodes 3, 4, and 1, respectively. The full state space for the CPN model has 13,215 nodes and 52,784 arcs and is far too big to be conveniently represented as a drawing. Drawing fragments of a state space, like the one in Fig. 33, can, however, be a very effective way of analysing the markings reachable within a small number of steps from a given marking.

State spaces are calculated fully automatically by the CPN state space tool using a state space construction algorithm. CPN Tools stores the directed graph representing the state space in internal memory. This means that the full state space can only be generated if it fits into the available computer memory. The tool supports a number of stop and branching options that makes it possible for the user to control the state space generation.

The generation of the full state spaces is in most cases followed by the generation of the *Strongly Connected Component Graph* (SCC-graph) which is derived from

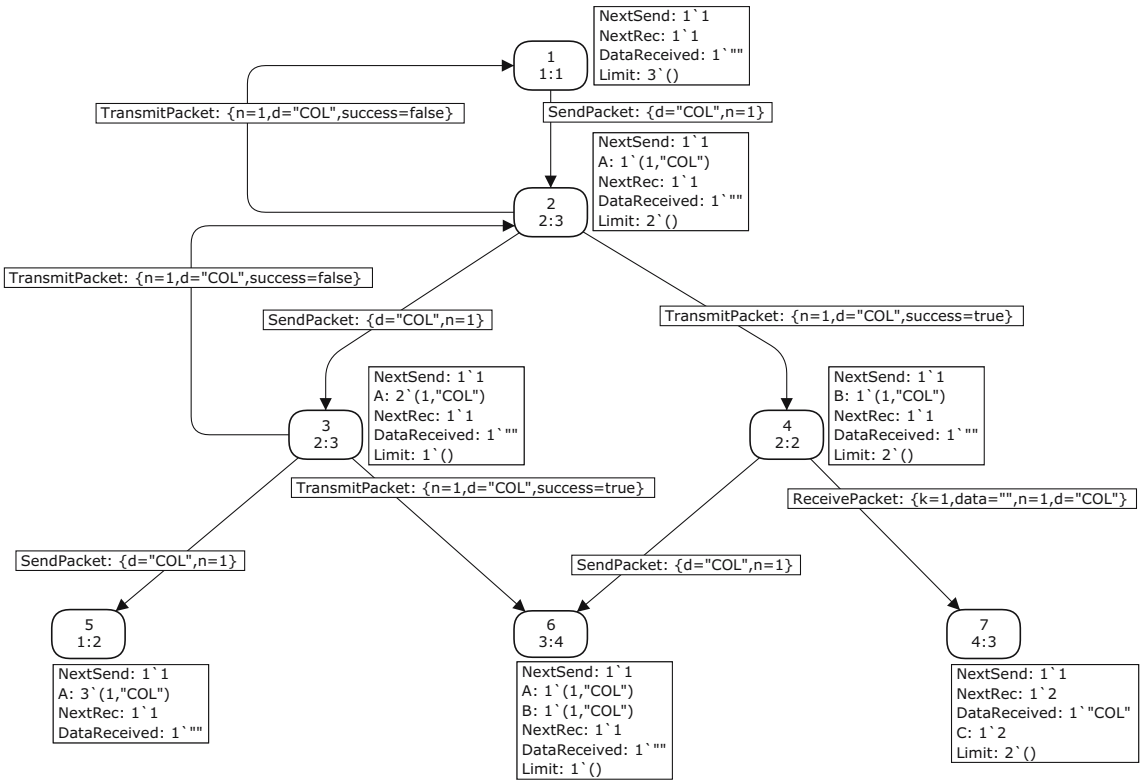


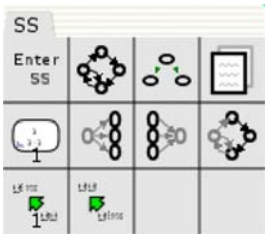

**Fig. 33** Initial state space fragment for CPN model in Fig. 32

the graph structure of the state space. The nodes in the SCC-graph are subgraphs called *strongly connected components* (SCCs) and are obtained by making a disjoint division of the nodes in the state space such that two state space nodes are in the same SCC if and only if they are mutually reachable, i.e., there exists a path in the state space from the first node to the second node and vice versa. The SCC-graph is used by CPN Tools to determine a number of the standard behavioural properties of the model (as we will explain below) and the structure of the SCC-graph quite often gives useful information about the overall behaviour of the model being analysed.

Tools for generating, investigating, and displaying state spaces are found in the State Space palette of CPN Tools which is shown in Fig. 34. The available tools (from left to right and top to bottom) are:

- Generate model-specific code for state space analysis.
- Generate a state space.
- Generate an SCC graph.
- Save a state space report.
- Display a node in a state space.
- Display the successors of a state space node.
- Display the predecessors of a state space node.

**Fig. 34** The State Space palette




- Evaluate a CPN ML expression that returns a list of either state space nodes or state space arcs, and display the resulting nodes and/or arcs from the state space.
- Display the marking corresponding to a state space node in the simulator.
- Add the current simulator state to the state space.

## 5.3 State space report

The first step when conducting state space analysis is usually to ask for a *state space report*, which provides some basic information about the size of the state space and standard behavioural properties of the CPN model. For the CPN model in Fig. 32, the state space report looks as shown in Figs. 36, 37, 38, and 39. First we have some *state space statistics* (see Fig. 35) telling how large the state space is. For the model of the protocol we have

13,215 nodes and 52,784 arcs. The construction of the full state space took 53 s. We also get statistics about the SCC-graph. It has 5,013 nodes and 37,312 arcs, and was calculated in 2 s. The fact that there are fewer nodes in the SCC-graph than in the state space immediately tells us that there exist cycles in the state space of the simple protocol. This implies that we can have infinite occurrence sequences and that the protocol may not terminate.

The next two parts of the state space report contain information about the *boundedness properties*. The boundedness properties tell how many (and which) tokens a place may hold—when we consider all reachable markings.

Figure 36 specifies the best upper and lower integer bounds. The *best upper integer bound* of a place specifies the maximal number of tokens that can reside on a place in any reachable marking. The best upper integer bound of the place DataReceived is 1 which means that there is at most one token on place DataReceived, and there exists reachable markings where there is one token on DataReceived. This is what we would expect, since DataReceived is always supposed to contain a single token with a colour corresponding to the data that has been received. The place A has a best upper integer bound of 3 which means that in any reachable marking there are at most three tokens on A, and there exists a reachable marking where there is exactly three tokens on A. Similar remarks apply to the places B, C, and D. This is what we would expect, since we modified the original model by introducing the Limit place to ensure that there are at most three tokens simultaneously on the places A, B, C, and D. What we learn from the best upper integers bounds of the four network places is that there are markings where the maximum number of packets allowed simultaneously on the network are all in one network buffer.

The *best lower integer bounds* for a place specifies the minimal number of tokens that can reside on the place in any reachable marking. The place DataReceived has a best lower integer bound of 1 which means that there is always at least one token on this place. Together with the best upper integer bound of 1 this means that there is exactly one token on this place in any reachable marking. When the best upper and lower integer bound are equal it implies that the place always contains the same number of tokens (as given by the two integer bounds)—even if the colour of these tokens may vary. For example, place DataReceived always contains exactly one token, and place PacketsToSend always contains exactly six tokens. The best lower integer bound of the place A is 0 which means that there exists a reachable marking in which there are no tokens on this place. A similar remark applies to the places B, C, and D.

Above, we have considered the minimal and maximal *number* of tokens that may be present on a place ignoring the token colours. Figure 37 specifies the best upper and lower multi-set bounds. These bounds consider not only the number of tokens, but also the colours of the tokens. The *best upper multi-set bound* of a place specifies for each colour in the colour set of the place the maximal numbers of tokens that is present on this place with the given colour in any reachable marking. This is specified as a multi-set, where the coefficient of each value is the maximal number of tokens with the given value.

As an example, the place C has the following multi-set as the best upper multi-set bound:

```
3`2 ++ 3`3 ++ 3`4 ++ 3`5 ++ 3`6 ++ 3`7
```

This specifies that there is a maximum of three tokens with the colour 2 on C in any reachable marking (and similarly for the colours 3, 4, 5, 6, 7). It also specifies that there exists a reachable marking where there are three tokens with the colour 2 on the place. The best upper multi-set for C also specifies that there can never be a token with the colour 1 on the place. This is expected, since the acknowledgements sent by the receiver always specify the data packet expected next, and because the first acknowledgement (with sequence

```
State Space Statistics

  State Space                 Scc Graph

    Nodes:  13215               Nodes:  5013
    Arcs:   52784               Arcs:   37312
    Secs:   53                  Secs:   2
    Status: Full
```

**Fig. 35** State space report—statistics

```
Best Integers Bounds      Upper      Lower

  PacketsToSend             6          6
  DataReceived              1          1
  NextSend, NextRec         1          1
  A, B, C, D                3          0
  Limit                     3          0
```

**Fig. 36** State space report—integer bounds

```
Best Upper Multi-set Bounds

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

  DataReceived        1`""++1`"COL"++1`"COLOUR"++
                      1`"COLOURED "++
                      1`"COLOURED PET"++
                      1`"COLOURED PETRI "++
                      1`"COLOURED PETRI NET"

  NextSend, Nextrec   1`1++1`2++1`3++1`4++1`5++
                      1`6++1`7

  A, B                3`(1,"COL")++3`(2,"OUR")++
                      3`(3,"ED ")++3`(4,"PET")++
                      3`(5,"RI ")++3`(6,"NET")

  C, D                3`2++3`3++3`4++3`5++3`6++3`7

  Limit               3`()

Best Lower Multi-set Bounds

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

  DataReceived        empty

  NextSend, NextRec   empty

  A, B, C, D          empty

  Limit               empty
```

**Fig. 37** State space report—multi-set bounds

number 2) is sent after the data packet with sequence number 1 is received.

As another example, consider the place DataReceived which has the following best upper multi-set bound:

```
1`"" ++ 1`"COL" ++ 1`"COLOUR" ++
1`"COLOURED " ++ 1`"COLOURED PET" ++
1`"COLOURED PETRI " ++ 1`"COLOURED PETRI NET"
```

This specifies a maximum of one token with the colour `""` on DataReceived in any reachable marking (and similarly for the other values in the multi-set). The size of the above multi-set is 7—even though DataReceived has a single token in each reachable marking as specified by the best upper and lower integer bounds in Fig. 36. From the best upper multi-set bound and the best upper and lower integer bounds it follows that the possible markings of the place DataReceived are:

```
1`""
1`"COL"
1`"COLOUR"
1`"COLOURED "
1`"COLOURED PET"
1`"COLOURED PETRI"
1`"COLOURED PETRI NET"
```

This corresponds to the expected prefixes of the data being sent from the sender. From the boundedness properties we cannot see the order in which these markings are reached.

Above we have illustrated that the integer and the multi-set bounds often tell us different and complementary "stories". The integer bounds of DataReceived tell us that the place always has exactly one token, but nothing about the possible colours of this token. The best upper multi-set bound of DataReceived tells us the tokens colours we may have at the place, but not that there is only one token at a time. It should be noted that there is no guarantee that there exists a reachable marking with the multi-set equal to the best upper multi-set bound. This is illustrated by the place DataReceived.

The *best lower multi-set bound* of a place specifies for each colour in the colour set of the place the minimal numbers of tokens that is present on this place with the given colour in any reachable marking. This is specified as a multi-set, where the coefficient of each value is the minimal number of tokens with the given value. Best lower multi-set bounds give, therefore, information about how many tokens of each colour that are always present on a given place. All places for the simple protocol except PacketsToSend have the empty multi-set `empty` as best lower multi-set bound. This means that there are no tokens which are always present on these places. However, we cannot conclude that there exists a reachable marking with no tokens on these places. This is illustrated by DataReceived, NextSend, and NextRec which always have one token each. The best lower multi-set for PacketToSend is:

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

This means that there is at least one token with the colour `(1,"COL")` on PacketToSend in any reachable marking (and similarly for the other values in the multi-set). This is as expected since the data packet being removed from PacketsToSend when SendPacket occurs is immediately put back again.

```
Home Properties

  Home Markings:      [4868]
```

**Fig. 38** State space report—home properties

Figure 38 shows the part of the state space report specifying the *home properties*. The home properties tell us that there exists a single *home marking*, which has the node number 4868. A home marking $M_{\text{home}}$ is a marking which can be reached from any reachable marking. This means that it is impossible to have an occurrence sequence which cannot be extended to reach $M_{\text{home}}$. In other words, we cannot do things which will make it impossible to reach $M_{\text{home}}$ afterwards.

In the protocol system we have a single home marking. By asking CPN Tools to display the marking corresponding to node 4868 from the state space, we get the marking that is shown in Fig. 29. It can be seen that this is the marking in which the protocol has successfully finished the transmission of all six data packets. The fact that this is a home marking means that no matter what happens when the protocol is executed (e.g., packet loss and overtaking of packets on the network), it is always possible to reach the marking where the transmission of all six data packets has been completed successfully. It should be noted that we only know that it is *possible* to reach the home marking $M_{\text{home}}$ from any reachable marking $M$. There is no guarantee that $M_{\text{home}}$ actually will be reached from $M$, i.e., there may exist occurrence sequences that start in $M$ and never reach $M_{\text{home}}$. As an example, the simple protocol has the infinite occurrence sequence in which SendPacket followed by TransmitPacket with a binding losing the data packet occur infinitely many times immediately after each other. In this case we will never reach the marking in Fig. 29. If we want to exclude that kind of behaviour, we would introduce a counter which limits the number of retransmissions allowed for each individual packet.

The liveness properties in Fig. 39 specify that there is a single *dead marking* which has the node number 4868. A dead marking is a marking in which no binding elements are enabled. This means that the marking corresponding to node 4868 is both a home marking and a dead marking. The fact that node 4868 is the only dead marking tells us that the protocol as specified by the CPN model is *partially correct*—if execution terminates we have the correct result. Furthermore, because node 4868 is also a home marking it is always possible to terminate the protocol with the correct result. It may be a bit surprising that a dead marking can be a home marking, but this is possible because any marking can be reached from itself by means of the trivial occurrence sequence of length zero.

```
Liveness Properties

  Dead Markings:      [4868]
  Dead Transitions:   None
  Live Transitions:   None
```

**Fig. 39** State space report—liveness properties

Figure 39 tells us that there are no *live transitions*. A transition is *live* if from any reachable marking we can always find an occurrence sequence containing the transition. In other words, we cannot do things which will make it impossible for the transition to occur afterwards. We have already seen that our protocol has a dead marking, and this is the reason why it cannot have any live transitions—no transitions can be made enabled from the dead marking.

Finally, Fig. 39 also tells us that there are no *dead transitions*. A transition is *dead* if there are no reachable markings in which it is enabled. That there are no dead transitions means that each transition in the protocol has the possibility to occur at least once. If a model has dead transitions then they correspond to parts of the model that can never be activated. Hence, we can remove dead transitions from the model without changing the behaviour of it.

### 5.4 Query functions

Above, we have discussed the contents of the state space report (with the exception of the so-called fairness properties). It is produced totally automatically, and it contains information about a number of key properties for the CPN model under analysis. The behavioural properties investigated in the state space report are standard properties that can be investigated for any model. Hence, the state space report is often the first thing which the user asks for. However, the user may also want to investigate properties that are not general enough to be part of the state space report. For this purpose a number of predefined *query functions* are available in CPN Tools that make it possible to write user-defined and model-dependent queries. These queries are written in the CPN ML programming language. The CPN state space tool uses the predefined query functions when computing the content of the state space report. CPN Tools additionally contains a library that makes it possible to formulate queries in a temporal logic [8].

An example of a model-specific query for the CPN model in Fig. 32 would be to check whether the protocol obeys the stop-and-wait strategy, i.e., whether the sequence number of the data packet currently being sent by the sender is at most one less than the sequence number expected by the receiver. For this purpose we can implement a predicate `StopWait` which given a marking, i.e., a node from the state space, checks whether the difference between the sequence number in the receiver side (represented by the token on place NextRec) and the sequence number in the sender side (represented by the token on place NextSend) is at most one. The implementation of the `StopWait` predicate is as follows:

```
fun StopWait (n:Node) =
  let
   val Sender_Seq   =
       ms_to_col (Mark.Protocol'NextSend 1 n);
   val Receiver_Seq =
       ms_to_col (Mark.Protocol'NextRec 1 n);
  in
   (Receiver_Seq - Sender_Seq) <= 1
  end;
```

The function extracts the colours of the tokens on the places NextSend and NextRec in the current marking given by the parameter `n` which is a `Node` in the state space. It then compares the value of the two sequence numbers as described above. The `StopWait` predicate can then be provided to the query function `PredAllNodes` which lists all nodes in the state space satisfying a given predicate. Surprisingly, not all nodes satisfy the predicate. The reason for this is that acknowledgements may overtake each other on the places C and D which means that it is possible for the sender to receive an old acknowledgement that causes the sender to decrement its sequence number. Using the query functions `ArcsInPath` provided by CPN Tools it is easy to obtain a counter example, i.e., an occurrence sequence leading from the initial marking to a marking where the predicate does not hold and have it visualised using the drawing facilities of CPN Tools.

One of the main advantages of state space methods is that they are relatively easy to use, and they have a high degree of automation. The ease of use is primarily due to the fact that with state space methods it is possible to hide a large portion of the underlying complex mathematics from the user. This means that quite often the user is only required to formulate the property which is to be verified and then apply a computer tool. The main disadvantage of state spaces is the *state explosion problem* [38]: even relatively small systems may have an astronomical or even infinite number of reachable states, and this is a serious problem for the use of state space methods for the verification of real-life systems. CPN Tools includes a collection of reduction techniques for alleviating the state explosion problem inherent in state space-based verification. These advanced state space techniques typically represent the state space in a compact form or represent only parts of the state space. The state space reduction is done in such a way that it is still possible to verify properties of the system. A discussion of these reduction methods is, however, beyond the scope of this paper, for details see, e.g., [9,18,25,26].

The state space of a timed CPN model is defined in a similar way as for untimed CPN models, except that each state space node now corresponds to a timed marking, i.e., the timed multi-sets specifying the markings of the places and the value of the global clock. CPN Tools also supports state space analysis of timed CPN models.

## 6 Performance analysis

Simulation-based performance analysis is supported via automatic simulation combined with data collection. The basic idea of simulation-based performance analysis is to conduct a number of lengthy simulations of the model during which data about the performance of the system is collected. The data typically provides information such as the size of queues, delay of packets, and load on various components. The collection of data is based on the concept of *data collector monitors* that allow the user to specify when and what data is to be collected during the individual steps of a series of automatic simulations. The data can be written in log files for post-processing, e.g., in spreadsheets, or a *performance report* can be saved summarising key figures for the collected data such as average, standard deviation, and confidence intervals. Simulation-based performance analysis typically uses *batch simulation* that makes it possible to explore the parameter space of the model and conduct multiple simulations of each parameter configuration to obtain results that are statistically reliable. We illustrate performance analysis using a timed model of the simple protocol.

### 6.1 Revised model for performance analysis

We will develop a slightly modified version of our protocol model to be used for performance analysis. This model contains a module for the arrival of the data packets to be sent by the sender and a module for the protocol. These two modules are shown in Figs. 41 and 42, and they are tied together via the System module shown in Fig. 40.

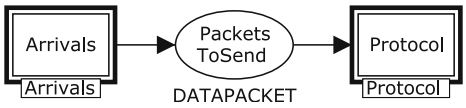



**Fig. 40** System module—top-level module for the hierarchical, timed protocol model

Let us first consider the data packets that the sender must send. When analysing the performance of a system, one is often interested in measuring the performance of a system when it processes a particular kind of workload. For example, the workload for the timed protocol is data packets. With CPN models it is possible to use both fixed workloads, i.e., workloads that are predetermined at the start of a simulation, and dynamic workloads.

In previous sections, the initial marking inscription of the place PacketsToSend has been used to determine exactly which data packets should be sent. When debugging the model or when examining the logical correctness of the protocol via state space analysis, it is sufficient to examine the behaviour of the model for a limited number of data packets. However, it is unlikely that such a protocol would be used to send a small, fixed number of data packets that are always available in a buffer. If data packets arrive much faster than they can be sent and acknowledged, then a large queue of data packets will grow at the sender, and an unacceptably large amount of time may pass from when a packet arrives until it is acknowledged. The arrival of data packets will affect the performance of the protocol, and this behaviour should, therefore, be modelled accurately. For this system, it is useful to consider a dynamic workload.

The arrival of data packets is modelled in the Arrivals module shown in Fig. 41. A single timed token on the place Next is used to control the arrival of new data packets. The colour of the token represents the sequence number of the next data packet that will arrive, and the time stamp determines when a new data packet will arrive. In the marking shown in Fig. 41, the next data packet will arrive at time `3161`, and it will get the sequence number `16`. When the CreatePacket transition occurs, the time delay of the token that is added to the place Next is determined by the `nextArrival` function defined as:

```
fun nextArrival() = discrete(200,220);
```

Intuitively, the value returned by the `nextArrival` function represents the amount of time that will pass before the next data packet arrives. Here the `discrete` probability distribution function is used, but CPN Tools provides support for a number of probability distributions, including, uniform, normal, Erlang, and exponential.

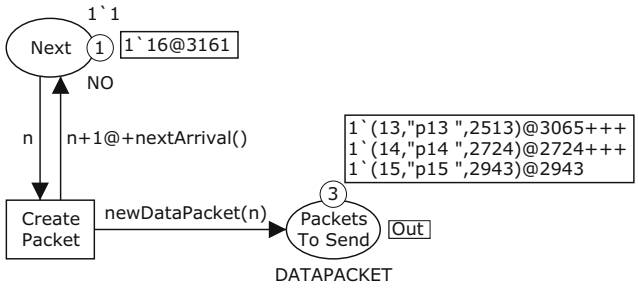



**Fig. 41** Module for the arrival of data packets

When doing performance analysis, it is often interesting to measure the amount of time that passes between different events. In the timed protocol example, it is interesting to measure the amount of time that passes from when a data packet arrives at the sender until the data packet is received by the receiver. To be able to do this it must be possible to record and remember the time at which the first event occurs. In a CPN model, the easiest way to record this kind of information is to include it in token values. The colour set `DATAPACKET` shown below is used to model the data packets. Here the token colour of a data packet is a triple consisting of a sequence number, the data contents, and the time of arrival for the data packet. The time of arrival is represented by the integer colour set `TOA`.

```
colset TOA        = int;
var t             : TOA;
colset DATAPACKET = product NO*DATA*TOA
                    timed;
```

When the CreatePacket transition occurs, a new data packet is created by the `newDataPacket` function (not shown) which returns a value of type `DATAPACKET`, i.e., a triple. The sequence number of the new data packet is determined by the argument `n`, shown in Fig. 41, and the time of arrival of the data packet will be equal to the model time at which the transition occurs.

Figure 42 shows a variant of the timed protocol that differs from the timed model in Fig. 14 in a number of ways. In Fig. 42, data packets are discarded after they have been acknowledged. Transition RemovePacket removes data packets from the place PacketsToSend after they have been acknowledged. The inscription to the upper left of the transition, i.e., the inscription `[n<k]`, is the guard for the transition. This guard ensures that only packets that have a sequence number that is smaller than the sequence number on NextSend will be removed from place PacketsToSend. A time delay inscription on the arc from PacketsToSend to RemovePacket allows tokens to be removed ahead of time from the place PacketsToSend. The arc inscription on the arc

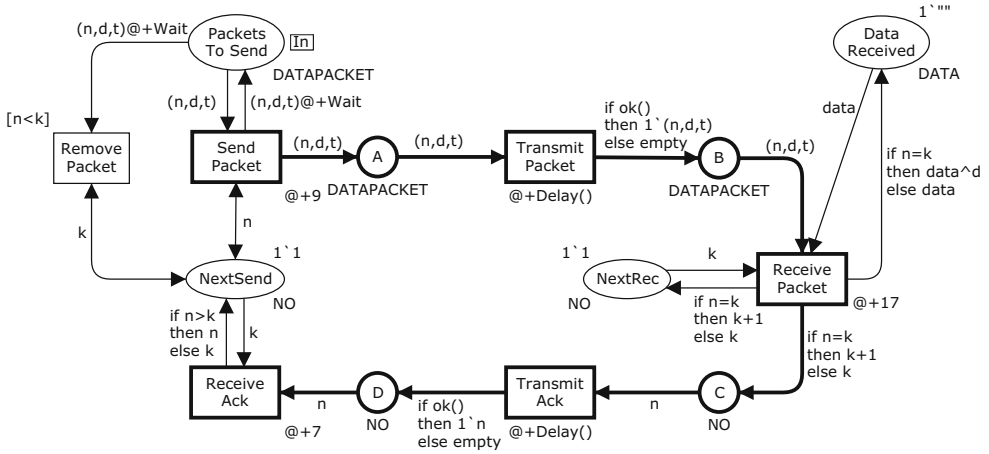



**Fig. 42** Module for the protocol

from ReceiveAck to NextSend has also been changed so that the sequence number on NextSend will never be decreased. This means that the sender will not retransmit packets that have already been received if a duplicate acknowledgement is received.

The loss of data packets is also represented slightly differently. For automatic simulations of the model shown in Fig. 14, the variable `success` for the transition TransmitPacket would be randomly bound to either `true` or `false` each time the transition occurs, and both values would be equally likely. In other words, approximately 50% of the data packets would be lost during automatic simulations. Many networks are more reliable than this, so when studying the performance of the protocol, it is important to represent the loss rate more accurately. In Fig. 42 the `ok` function determines whether a packet will be transmitted successfully or lost:

```
fun ok() = uniform(0.0,1.0) <= 0.9;
```

The `uniform` function will return a random value between 0.0 and 1.0, and all values in the interval have the same probability of being chosen. The `ok` function specifies that there is a 90% chance that data packets will be transmitted successfully. The loss of acknowledgements is modelled in a similar manner. Since fewer packets will be lost, it is not necessary to retransmit packets as often. Therefore, the value of the constant `Wait`, which determines how long the sender should wait before retransmitting a data packet, has been changed to 175.

## 6.2 Performance measures and data collectors

There are a number of interesting performance measures for the timed protocol example. For example, it could be interesting to know how many data packets or how many duplicate data packets are received by the receiver during a simulation. Measuring the number of packets to send will indicate whether there is a backlog of data packets at the sender. In this example, packet delay will be the amount of time that passes from when a data packet arrives at the sender until it is correctly received by the receiver. Calculating average and maximum packet delay will indicate whether data packets are received in a timely fashion.

Such performance measures can be calculated based on numerical data that is extracted or collected from a CPN model during simulations. In CPN Tools, *data collector monitors* are used for this purpose. As we shall see, the numerical data can be extracted from the binding elements that occur and the markings that are reached during a simulation. Different kinds of data collector monitors can be used for different purposes. There are standard or predefined data collector monitors that can be used for any CPN model. We will also see examples of user-defined or generic data collector monitors which collect data that is model specific. Data collector monitors are created with tools from the Monitoring palette, and they can be created by cloning and modifying existing data collector monitors.

Calculating the number of data packets that are received by the receiver is simply a matter of counting the number of times the ReceivePacket transition occurs during a simulation. In CPN Tools a *count transition occurrences monitor* is a standard monitor for just this purpose. The monitor for counting the number of received data packets is named ReceivedPackets.

In the model, a duplicate data packet is received when the ReceivePacket transition occurs with a binding where `n≠k`. *Generic data collector monitors* can be used to collect any kind of numerical data from a CPN model. The behaviour of such monitors is determined by their monitoring functions which are accessible to the

user. Here we will use a generic data collector monitor to calculate the number of duplicate packets received by counting the number of times a transition occurs with a particular binding. The monitor is named ReceivedDuplicatePackets.

So, we need to be able to determine at least two things for the ReceivedDuplicatePackets monitor: (1) when data should be collected from the model for updating the counter of duplicate data packets, and (2) the value with which the counter should be increased. This functionality is determined by the monitoring functions for the data collector monitor.

In a data collector monitor, a *predicate function* will be called periodically, and it should return `true` whenever the monitor should collect data from the model. The predicate function for the ReceivedDuplicatePackets monitor looks like this:

```
fun pred (Protocol'Receive_Packet
             (1,{d,data,k,n,t})) = true
```

The predicate function can examine the bindings of the variables of the transition, but it is defined so that it ignores the bindings of the variables. The function returns `true` every time ReceivePacket occurs.

In a data collector monitor, an *observation function* collects numerical data from the model. An observation function is called each time the predicate function for the same monitor is called and returns `true`. The following observation function collects data that is used to calculate the number of duplicate data packets received:

```
fun obs (Protocol'Receive_Packet
             (1, {d,data,k,n,t})) =
    if n=k then 0 else 1
```

The predicate function above determines that this observation function will be called each time ReceivePacket occurs. The function will return 1 whenever a duplicate data packet is received by the receiver, and 0 when a packet is received the first time. The data values that are returned by the observation function are used to calculate statistics, such as the sum, average, and maximum of the data values that are collected. The sum of the data values collected by this monitor will indicate how many duplicate data packets have been received during a simulation. The average of the data values will be the proportion of duplicate data packets to the total number of data packets received.

Each data collector monitor has two additional monitoring functions: an *initialisation function* and a *stop function*. The initialisation function can be used to collect data from the initial marking of the model. Similarly, the stop function can be used to collect data from the final marking of a simulation. Initialisation and stop functions cannot be used to collect data from binding elements. For the ReceivedDuplicatePackets monitor, neither the initialisation function nor the stop function is used to collect data from markings.

We define another generic data collector monitor called PacketDelay which collects data from occurring binding elements. It is defined to measure packet delay, and to calculate average and maximum packet delay. When the ReceivePacket transition occurs (see Fig. 42), the variable `t` is bound to the arrival time of the data packet that is being received, and this value can be used to calculate the packet delay. This data collector monitor is associated only with the ReceivePacket transition.

The predicate function for this monitor is exactly the same as the one for the ReceivedDuplicatePackets monitor. The observation function for measuring packet delay subtracts the time of arrival for the data packet `t` from the model time at which the transition occurs, and the time that is required to receive the data packet (`17`) is added, since the time delay for receiving the data packet ought to be included in the packet delay. The initialisation and stop functions are not used to collect data from markings.

We have seen a number of examples of how data collector monitors can be used to collect data from binding elements that occur during a simulation. We will now see how monitors can be used to collect data from the markings that are reached during a simulation.

The number of tokens on the place PacketsToSend in a particular marking is equal to the number of packets to be sent. In CPN Tools, a *marking size monitor* is a standard monitor that is used to measure the number of tokens on a place during a simulation. Such a monitor can calculate the average and maximum number of tokens on a place during a simulation. We will use a marking size monitor named PacketsToSend for measuring the number of tokens on the place PacketsToSend during a simulation, and to calculate the average and maximum number of data packets to be sent during a simulation.

One way to measure the number of tokens on a place is to count the number of tokens on the place in the initial marking and after every step in a simulation. If the model is not timed, then this is a good way to collect the data for calculating the average number of tokens on the place, and a marking size monitor for untimed models does, in fact, use this technique.

However, for a timed model, it is often desirable to use timing information when calculating the average number of tokens on a place. Such timing information will be taken into consideration if we calculate the *time-average* number of packets to send. By time average we mean a weighted average of the possible number of data

packets to send (0, 1, 2, …) weighted by the proportion of time during the simulation that there were that many data packets to send. When calculating the time average, it is sufficient to measure the number of tokens at the place only when one of the transitions surrounding the place occurs. When the number of tokens is measured, the interval of model time that passes until the number of tokens is measured again is used to weight the first measurement. A marking size monitor for timed models uses this technique.

We define another generic data collector monitor called WaitingForTransmission which calculates the time-average number of data packets and acknowledgements that are waiting to be transmitted. In other words, it measures the sum of the number of tokens on places A and C. Since it calculates the time-average number of tokens on these two places, it is sufficient to measure the number of tokens when one of the transitions connected to either of the places occurs. The monitor is associated with the following nodes: places A and C, and transitions SendPacket, TransmitPacket, ReceivePacket, and TransmitAck. The predicate function will be called only when one of these transitions occurs, and the function will return `true` whenever it is called. The observation function returns the total number of tokens on the two places:

```
fun obs (bindelem,
    Protocol'A_1_mark : DATAPACKET tms,
    Protocol'C_1_mark : NO tms) =
 (size Protocol'A_1_mark) +
 (size Protocol'C_1_mark)
```

The initialisation and stop functions for the WaitingForTransmission monitor are similar to the observation function of the monitor.

Generic data collector monitors are not required to collect data regularly during simulations. For the timed protocol example, we would like to calculate throughput as the number of unique data packets that were received by the receiver per second. This can be calculated at the end of a simulation by dividing the number of unique packets that are received during the simulation by the length of the simulation expressed as seconds of model time. In this model, one second is equal to 1,000,000 units of model time. The stop function for generic data collector monitor named Throughput is used to calculate throughput. It is also necessary to define predicate, observation, and initialisation functions for the Throughput monitor, even though these functions will not be used to collect data.

### 6.3 Statistics

Since most simulation models contain random behaviour, the simulation output data are also random, and care must be taken when interpreting and analysing the output data. Performance measures are estimated by calculating statistics for the data that is collected by data collector monitors during a simulation. Below we give a very brief description of some of the statistical concepts used when conducting performance analysis using CPN Tools. For a more detailed introduction to statistics please see a textbook on statistics or simulation, such as [27] or [3].

Statistics that are calculated relative to a collection of discrete data values are known as *discrete-parameter statistics* [21]. For example, the average packet delay is defined relative to the collection of discrete observations $D_i$ where $D_i$ is the packet delay for the $i$th data packet received during a simulation.

As discussed in the previous section, the time-average number of data packets waiting to be sent is equal to the time-average number of tokens on place PacketsToSend. This is a different kind of "average" than the average packet delay, because time-average is taken over (continuous) time, rather than over data packets (which are discrete). Time-average is an example of a *continuous-time statistic*. Intuitively, continuous-time statistics are those that result from taking the (time) average, minimum, or maximum of a plot of something during the simulation, where the x-axis is continuous time [21].

A data collector monitor can calculate either the (regular) average or the time-average. A monitor that calculates the (regular) average is said to calculate discrete-parameter statistics. A monitor that calculates time-average is said to calculate continuous-time statistics. Both kinds of monitors can calculate a number of different statistics, including: count (number of observations), minimum, maximum, sum, and average. Each data collector monitor has predefined functions that can be used to access the statistics that are calculated for the monitor, such as `count`, `sum`, `avrg`, and `max`.

It is important to remember that running different simulations will result in different estimates of performance measures. *Confidence intervals* are often used to evaluate the accuracy of performance measure estimates. Accurate confidence intervals can only be calculated for data values that are *independent and identically distributed (IID)*. Intuitively, data values are IID if they are not related to each other, and if they have the same probability distribution. CPN Tools can calculate 90, 95, and 99% confidence intervals. Figure 43 shows an example of how the 95% confidence interval for average

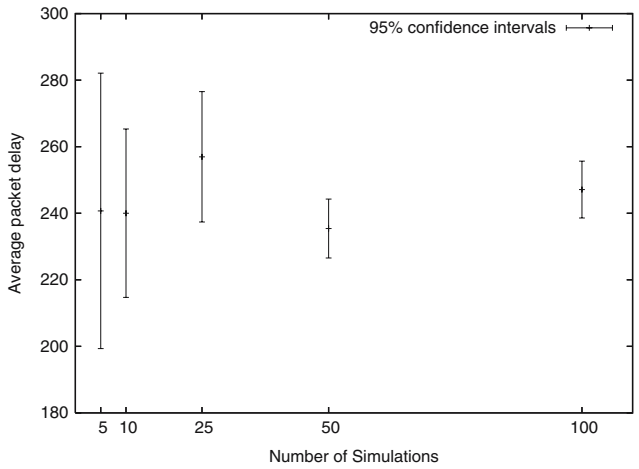



**Fig. 43** 95% confidence intervals for average packet delay

packet delay generally decreases as more IID estimates are collected from increasing numbers of simulations.

Since all of the data collected by a single data collector monitor is not likely to be IID, it is necessary to find other methods for collecting IID estimates of performance measures. One widely used method is to collect IID estimates from independent, *simulation replications*, which start in the same initial state and stop when the same stop criterion is fulfilled. For example, simulation replications of the timed protocol example could all stop after 1,000 data packets have been received, or after one hour of model time has passed.

The *batch-means method* is another commonly used technique for obtaining IID estimates of performance measures. In this method, IID estimates of performance measures are derived from data values from a single, long simulation. The idea behind this method is to group individual observations into a number of batches, to calculate the averages of the observations within each batch, and then to use the averages from each of the batches as IID estimates of a performance measure.

As we will see in Sect. 6.5, determining whether IID estimates should be obtained from simulation replications or via the batch-means method will depend on the kinds of simulation experiments that are to be done.

## 6.4 Performance output

Several different kinds of output can be generated for data collector monitors. In this section, we will see some examples of performance-related output, including log files, statistical reports, and scripts for plotting data values.

All of the data that is collected by a data collector can be saved in a *data collector log file*. The log file also contains information about the steps and model

```
data counter step time
  0      1     0    0
  1      2     1    0
  1      3     2    0
  0      4     7    169
  1      5     8    220
  1      6     9    220
```

**Fig. 44** Data collector log file for PacketsToSend monitor

times at which the data was collected. An option for a data collector monitor determines whether a log file should be generated for the monitor. Figure 44 shows an example of a log file for the PacketsToSend monitor. The last line of Fig. 44 shows that there was 1 token on place PacketsToSend after the 9th simulation step which occurred at model time 220, and this was the 6th time that the number of tokens on the place was measured. The monitor measures the number of tokens on the place whenever one of the surrounding transitions occurs. Since the number of tokens on the place does not change when the SendPacket transition occurs, successive data values that are collected by the monitor may be the same, as can be seen in the first column of Fig. 44. The last two columns show that more than one data value may be collected after different simulation steps that occur at the same model time.

Data collector log files can be post-processed after a simulation has completed. For example, they can be imported into a spreadsheet or plotted. CPN Tools generates scripts for plotting data collector log files with the gnuplot program [14]. Figure 45 shows an example of how a log file for the PacketsToSend monitor can be plotted with gnuplot.

The statistics that are calculated for data collector monitors are saved in different kinds of reports. A *sim-*

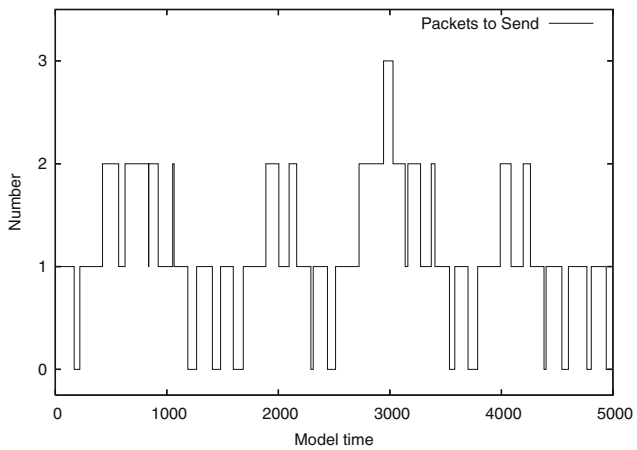



**Fig. 45** Plotting data for PacketsToSend monitor

| Continuous-time statistics | | | | | |
|---|---|---|---|---|---|
| Name | Count | Avrg | Min | Max | First Time |
| PacketsToSend | 2911 | 1.57 | 0 | 7 | 0 |
| WaitingForTransmission | 4223 | 0.14 | 0 | 1 | 0 |

| Discrete-parameter statistics | | | | | |
|---|---|---|---|---|---|
| Name | Count | Sum | Avrg | Min | Max |
| PacketDelay | 1000 | 273264 | 273.26 | 51 | 1275 |
| ReceivedDuplicatePackets | 1000 | 102 | 0.10 | 0 | 1 |
| ReceivedPackets | 1000 | 1000 | 1.00 | 1 | 1 |
| Throughput | 1 | 4756.99 | 4756.99 | 4756.99 | 4756.99 |

**Fig. 46** Statistics from a simulation performance report

*ulation performance report* contains statistics that are calculated for the data that is collected by data collectors during one simulation. Figure 46 shows statistics from a simulation performance report. The simulation stopped after 1,000 data packets had been received by the receiver. In addition to statistics, the report contains information (not shown in Fig. 46) indicating that the simulation stopped at model time 188,775 after the execution of 6,916 simulation steps.

The upper part of Fig. 46 shows the continuous-time statistics from the simulation performance report. There are five columns with different kinds of statistics. The user can determine which statistics should be included in a simulation performance report. The statistics for the PacketsToSend monitor show that the time-average number of data packets to send was 1.57, and that the maximum number of data packets to send was 7. The count statistic for the monitor shows that it collected 2,911 data values, i.e., it measured the number of tokens on place PacketsToSend 2,911 times. The first time the monitor measured the number of tokens was at model time 0. The time average for the WaitingForTransmission monitor shows the time-average number of packets waiting to be transmitted, i.e., the time-average number of tokens on places A and C, was 0.14.

The lower part of Fig. 46 shows the discrete-parameter statistics from the simulation performance report. The count statistic for the PacketDelay monitor shows that the monitor measured the packet delay for 1,000 data packets. The average packet delay is 273.26 units of time, with a minimum and maximum packet delay of 51 and 1,275, respectively. Note that not all statistics in a simulation report will be useful, for example the average, minimum, and maximum values for the ReceivedPackets monitor are not interesting, and the sum and count statistics show the same value, namely the number of packets that were received during the simulation. The statistics for the ReceivedDuplicatePackets monitor indicate that 102 out of 1,000 received packets were duplicates. Finally, the throughput for the simulation was 4,756.99 (unique) data packets per 1,000,000 time units, i.e., per second. Recall that the throughput is calculated by a monitor stop function at the end of a simulation, and this explains why the count statistic for the monitor is 1, and why the sum, average, minimum and maximum values for the monitor are the same.

The statistics in a simulation performance report are unreliable, because they are just one estimate of various different performance measures. Different statistics would be obtained if another simulation were run. A reliable estimate of a performance measure can be obtained by calculating a confidence interval for the average of a set of IID estimates for the performance measures. For example, IID estimates of the average packet delay can be obtained by running a number of independent simulation replications in CPN Tools. Such IID estimates can also be saved in log files. Figure 47 shows IID estimates from ten simulation replications for average packet delay for 1,000 packets. This data can then be used to calculate a confidence interval for a reliable estimate of average packet delay for the timed protocol example.

Another performance report contains reliable estimates of performance measures based on IID data values. Figure 48 shows an excerpt of such a performance report. The statistics shown in the figure are calculated for IID estimates of performance measures that were collected from ten simulation replications. A value in the 95% *Half Width* column is equal to half of the length of the 95% confidence interval for the average in the same row. For example, the data from Fig. 47 were used to calculate the statistics in the *avrg_iid* row under the *PacketDelay* heading, and the 95% confidence interval

| data | counter | data | counter |
|---|---|---|---|
| 255.41 | 1 | 311.17 | 6 |
| 203.79 | 2 | 206.15 | 7 |
| 213.70 | 3 | 254.84 | 8 |
| 212.62 | 4 | 271.66 | 9 |
| 255.54 | 5 | 215.25 | 10 |

**Fig. 47** Log file with IID estimates of average packet delay

| Name | Avrg | 95% Half Width | Min | Max |
|---|---|---|---|---|
| PacketDelay | | | | |
| count_iid | 1000.00 | 0.00 | 1000 | 1000 |
| max_iid | 1243.90 | 188.94 | 866 | 1750 |
| min_iid | 51.00 | 0.00 | 51 | 51 |
| sum_iid | 240012.90 | 25320.65 | 203789 | 311167 |
| avrg_iid | 240.01 | 25.32 | 203.79 | 311.17 |
| PacketsToSend | | | | |
| count iid | 2898.40 | 10.91 | 2879 | 2936 |
| max iid | 6.70 | 0.96 | 5 | 9 |
| min iid | 0.00 | 0.00 | 0 | 0 |
| avrg iid | 1.43 | 0.12 | 1.25 | 1.76 |

**Fig. 48** Reliable statistics based on data from ten replications

for the average packet delay based on the data from the ten simulations is 240.01±25.32. A value in the *Min* (*Max*) column is the minimum (maximum) of the IID estimates that were collected for the performance measure in the first column of the same row. For example, the minimum average packet delay from the ten replications is 203.79, while the maximum average packet delay is 311.17. Some, but rarely all, of the statistics shown in this performance report will represent useful performance measures for the model.

### 6.5 Conducting simulation experiments

Performance analysis studies are conducted for different reasons, e.g., to evaluate existing or planned systems, to compare alternative configurations, or to find an optimal configuration of a system. *Experimental design* [27] is concerned with determining which scenarios are going to be simulated and how each of the scenarios will be simulated in a simulation study. When deciding how many simulations to run and how long to run each simulation for a scenario, it is necessary to consider what kind of system is being modelled and what the purpose of the simulation study is. There are two kinds of systems, *terminating systems* and *non-terminating systems*, which will be described below. As we shall see, different statistical techniques are used to analyse these different kinds of systems.

Terminating systems are characterised by having a fixed starting condition and a naturally occurring event that marks the end of the system. An example of a terminating system is a business day at a bank that starts at 10 A.M. and ends at 4 P.M. The purpose of simulating terminating systems is to understand their behaviour during a certain period of time, and this is also referred to as studying the *transient behaviour* of the system.

*Terminating simulations* are used to simulate terminating systems. The length of a terminating simulation is determined either by the system itself, if the system is a terminating system, or by the objective of a simulation study. The length of a terminating simulation can be determined by a fixed amount of time, e.g., 6 h, or it can be determined by some condition, e.g., the departure of the tenth customer. Simulation replications are generally used to collect IID estimates of performance measures for terminating simulations.

In a non-terminating system, the duration of the system is not finite. The Internet exemplifies a non-terminating system. *Non-terminating simulations* are used to simulate non-terminating systems. In a non-terminating simulation, there is no event to signal the end of a simulation, and such simulations are typically used to investigate the long-term behaviour of a system. Non-terminating simulations must, of course, stop at some point, and it is a non-trivial problem to determine the proper duration of a non-terminating simulation.

If the behaviour of a non-terminating system becomes fairly stable at some point, then there are simple techniques for analysing the *steady-state behaviour* of the system using non-terminating simulations. Determining when, or if, a model reaches a steady state is also a complicated issue. IID estimates of performance measures for steady-state simulations are often obtained by applying the batch-means method during a long simulation.

It is often useful to be able to define a *warmup period* in which data should not be collected at the beginning of a simulation. When analysing steady-state behaviour using non-terminating simulations, the warmup period is used to ignore the behaviour of the model during the time it takes the model to reach a steady state. It can also be useful to define a warmup period for terminating simulations.

### 6.6 Model parameters and comparing configurations

Simulation-based performance analysis is often used to compare different scenarios or configurations of a system. The performance of a system is often dependent on several parameters. For example, the performance of the timed protocol example is influenced by parameters that determine the probability that packets will be transmitted successfully, the minimum and maximum times between arrivals of data packets, and even the distribution of the interarrival times of data packets. Changing these parameters will most likely affect the performance measures of the model. In the original version of the model, these parameters were hardcoded into functions, such as the `ok` function:

```
fun ok() = uniform(0.0,1.0) <= 0.9;
```

The probability that a packet will be transmitted successfully is 90%, as determined by the `0.9` in the `ok` function. This parameter can be changed by modifying the declaration of the `ok` function. In CPN Tools, these changes require the syntax of the declaration and the parts of the model that depend on the declaration to be rechecked. Making such changes can therefore be time consuming, especially if many parts of a large model must be rechecked. If parameters are defined in this way, then it is not possible to automatically simulate a number of configurations without manual interaction by a user.

These problems can be avoided if parameters are declared as *reference variables*. It is possible to change the value of a parameter declared as a reference variable without having to recheck the syntax of any part of

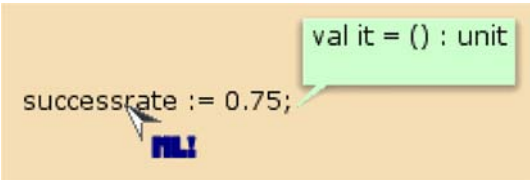



**Fig. 49** Using the Evaluate ML tool to change a parameter value

a model. Here is the declaration of a reference variable that determines the probability that a packet is transmitted successfully, and the `ok` function that uses the reference variable:

```
globref successrate = 0.9;
fun ok() =
     uniform(0.0,1.0) <= !successrate;
```

The keyword `globref` indicates that a global reference variable is being declared, i.e., the reference variable can be accessed from any part of the CP-net. The name of the reference variable is `successrate`, and the initial contents of the reference variable is `0.9`. The `!` operator is used to access the contents of a reference variable. Figure 49 shows how the value of `successrate` could be changed to `0.75` by picking up the Evaluate ML tool from the simulation tool palette (shown in Fig. 28) and applying the tool to an auxiliary text. The value of a parameter could also be changed in a function. If model parameters are declared as reference variables, then it is very easy to change the values of the parameters and to automatically simulate different model configurations.

If the scenarios of a simulation study are not predetermined, then the purpose of the study may be to locate the parameters that have the most impact on a particular performance measure or to locate important parameters in the system. *Sensitivity analysis* [22] investigates how large changes in parameters affect performance measures. *Gradient estimation* [27] examines how small changes in numerical parameters affect the performance of the system. *Optimisation* [2] is often just a sophisticated form of comparing alternative configurations, in that it is a systematic method for trying different combinations of parameters in the hope of finding the combination that gives the best results.

## 7 Visualisation

Even though the CPN modelling language supports abstraction and a module concept, there can, in many cases, be an overwhelming amount of detail in the constructed CPN model. Furthermore, observing every single step in a simulation is often too detailed for investigating the behaviour of a model, especially for large CPN models. This level of detail can be a limitation, in particular when presenting and discussing a CPN model with colleagues unfamiliar with the CPN modelling language. The idea of animation graphics and visualisation is to add high-level graphics to CPN models. This means that feedback from simulations can be obtained at a more adequate level of detail using application domain concepts, and in such a way that the underlying formal CPN model is fully transparent to the observer.

CPN Tools can interact with the BRITNeY Suite animation tool [40] that supports the creation of domain-specific graphics on top of CPN models. The animation tool supports a wide range of diagram types via an animation plug-in architecture. Below we give two examples of how the animation tool can be used to create domain-specific graphics. One example is the use of message sequence charts (MSCs) to illustrate the exchange of messages in the simple protocol. The second example illustrates how it is possible to provide input and control a simulation of a CPN model by interacting with system-specific graphics. We use the CPN model previously shown in Fig. 1 as a basis for both examples.

### 7.1 Message sequence charts

Figure 50 shows an example of an MSC created during a simulation of the CPN model of the simple protocol. The MSC has four columns. The leftmost column represents the sender and the rightmost column represents the receiver. The two middle columns represent the sender and receiver side of the network. The MSC captures a scenario where the first data packet sent by the sender is lost which then causes a retransmission of the data packet to occur. The retransmitted data packet is then successfully transmitted to the receiver and the corre-

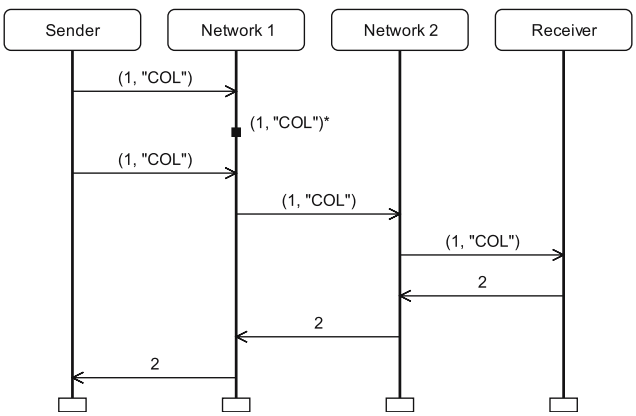



**Fig. 50** Example of a message sequence chart

sponding acknowledgement is successfully received by the sender.

The graphical feedback from the execution of the CPN model is achieved by attaching *code segments* to the transitions in the CPN model. A code segment consists of a piece of sequential CPN ML code that is executed whenever the corresponding transition occurs in the simulation of the CPN model. As an example, the transition SendPacket has the following code segment attached:

```
input (n,d);
output ();
action
msc.addEvent ("Sender","Network 1",
    NOxDATA.mkstr
    (n,d))
```

The code segment is provided with the value bound to the variables `n` and `d` via the input part of the code segment. The code segment then uses the function `msc.addEvent` provided by the animation tool to create an event from the Sender column to the Network1 column labelled with the value bound to `n` and `d`. The function `NOxDATA.mkstr` converts the pair `(n,d)` into a corresponding string used to label the arc of the MSC. The output part of the code segment is not used in this code segment, but its use will be illustrated in the next subsection. The other transitions of the CPN model have similar code segments. Each code segment essentially consists of invoking the appropriate primitive in the animation tool. An alternative to using code segments is to use a *user-defined monitor* to invoke the appropriate primitive depending on which transition occurs. For further details regarding user-defined monitors we refer to [10].

## 7.2 Interaction graphics

Figure 51 shows an example of a system-specific animation graphic created using an animation plug-in based on the SceneBeans framework [35]. The graphic illustrates the system modelled by the CPN model of the simple protocol. The computer to the left represents the sender, the computer to the right represents the receiver. The cloud in the middle represents the network. When a simulation is started, a modal dialog pops up that allows the observer to enter the text string to be transmitted from the sender to the receiver. In this case it was the text string `"Coloured Petri Nets"` that was entered. The modal dialog is created by adding a transition Init connected to the PacketsToSend as shown in Fig. 52. The Init transition is the only enabled transition in the initial marking, and when it occurs it removes

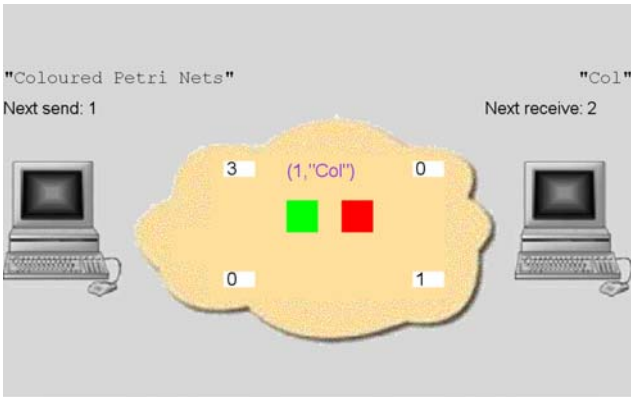



**Fig. 51** Example of system-specific animation graphics

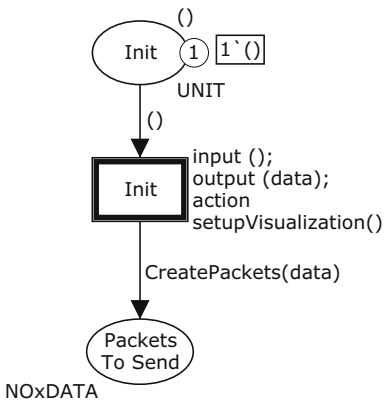



**Fig. 52** Transition Init and surroundings

the token with colour `()` from place Init, executes the attached code segment, and creates data packets on the place PacketsToSend according to the string entered by the user in the modal dialog. The function `setupVisualisation` (not shown) in the code segment of the Init transition invokes the primitive in the animation tool for creating a modal dialog box and returns the string entered. The string entered will be bound to the variable `data` used in the output part of the code segment. The string bound to `data` is then used as the argument for the function `CreatePackets` which splits the string into corresponding data packets.

The text on top of the sender computer shows the text string to be transmitted and is hence a representation of the marking of place PacketsToSend. Similarly, the text on top of the receiver computer shows the text string received by the receiver and is hence a representation of the marking of place DataReceived. The two counters on top of the sender and the receiver are representations of the values of the tokens on places NextSend and NextRec, respectively. The four numbers at the edges of the network cloud represent the number of tokens on the network places A, B, C, and D. In the topmost part of

the network a data packet `(1,"Col")` is shown which is currently in transit on the network. The two square boxes in the middle below the data packet (and coloured green and red) let the user decide whether the packet is to be lost (if the user clicks on the red square) or successfully transmitted (if the user clicks on the green square). This illustrates how it is possible to provide input to an ongoing simulation via the animation graphics. The interaction graphics shown in Fig. 51 is created in a similar way as the MSCs above by attaching code segments invoking the animation primitives to the transitions of the CPN model. The only difference is that the code segments now invoke primitives from a different animation plug-in. Furthermore, the interaction graphics has an XML scene file that describes the different elements in the animation, i.e., the computers, the network cloud, the labels, and the buttons. Further details on the animation tool can be found in [7,40].

## 8 Conclusion

To cope with the complexity of modern concurrent systems, it is crucial to provide methods that enable debugging and testing of central parts of the system designs prior to implementation and deployment. One way to approach the challenge of developing concurrent systems is to build an executable model of the system. Constructing a model and simulating it usually lead to significant new insights into the design and operation of the system considered and often results in a simpler and more streamlined design. Furthermore, constructing an executable model usually leads to a more complete specification of the design and makes it possible to make a systematic investigation of scenarios which can significantly decrease the number of design errors. The construction of a model of the system design typically means that more effort is spent in early phases of system development, i.e., requirements engineering, design, and specification. This additional investment is, in most cases, justified by the additional insight into the properties of the system that can be gained prior to implementation. Furthermore, many design problems and errors can be discovered and resolved in the requirements and design phase rather than in the implementation, test, and deployment phases. Finally, models are, in most cases, simpler and more complete than traditional design documents which means that the construction and exploration of the model has resulted in a more solid foundation for doing the implementation. This may in turn shorten the implementation and test phases significantly and decrease the number of flaws in the final system.

The development of CP-nets has been driven by the desire to develop an industrial-strength 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. CP-nets are, however, not a modelling language designed to replace other modelling languages (such as UML). In our view it should be used as a supplement to existing modelling languages and methodologies and can be used together with these or even integrated into them. High-level Petri Nets is an ISO/IEC standard [4] and the CPN modelling language and supporting computer tools conform to this standard. The practical application of CP-nets typically relies on a combination of interactive and automatic simulation, visualisation, state space analysis, and performance analysis. These activities in conjunction result in a *validation* of the system under consideration in the sense that it has been justified that the system has the desired properties and a high-degree of confidence and understanding of the system has been obtained. CPN models can be used to validate both the functional/logical correctness and the performance of a system. This saves a lot of time, because we do not need to construct two totally independent models of the system. Instead we can use a single model or (more often) two models that are very closely related to each other. There exist a number of modelling languages that are in widespread use for performance analysis of systems, e.g., queueing theory. However, most of these modelling languages cannot be used for modelling and validation of the logical properties of systems. Some of these are also unable to cope with performance analysis of systems which have irregular behaviour.

The paper has given a brief introduction to the CPN modelling language and the associated analysis methods. The reader interested in a complete treatment of the modelling language and analysis methods are referred to [16,17,19] or the forthcoming book [20]. The web site associated with [20] contains an extensive set of slides, exercises, and projects for using CP-nets and CPN Tools in courses. Further detailed information on the use of CPN Tools can be found via [10] which contains an elaborate set of manuals, tutorials, and other examples of CPN models. The CPN Tools web site also explains how to obtain a licence for CPN Tools. Beyond what was presented in this paper, CPN Tools further includes a collection of libraries for different purposes. One example is Comms/CPN [13] for TCP/IP communication between CPN models and external applications. CPN Tools generally has an architecture that allows the user to extend its functionality, such as experimenting with new state space methods. Hence, in addition to being a tool for modelling and validation it also provides a prototyping

environment for researchers interested in experimenting with new analysis algorithms.

We have illustrated the use of CP-nets for modelling and validation of a simple protocol. Readers interested in more elaborate industrial use of CPN models and CPN Tools are referred to [12,19,20,24], the proceedings of the annual CPN workshop [32], and the proceedings of the annual conference on theory and application of Petri Nets [30].

**Acknowledgments** There are many people who have influenced the development of CP-nets, their analysis methods, and their tool support. Unfortunately, we cannot mention them all here, but we are extremely grateful for their numerous contributions. We would like to thank the many developers of CPN Tools. In particular, Søren Christensen and Kjeld Høyer Mortensen who played key roles in the development of CPN Tools and its predecessor Design/CPN, Michel Beudouin-Lafon and Wendy E. Mackay who played a central role in designing the GUI of CPN Tools, and Henry Michael Lassen who has been instrumental in the development of CPN Tools since the start of the project. Users of CPN Tools have provided invaluable feedback that has helped to improve the tool. In particular, feedback from Wil van der Aalst and his group has helped to significantly improve the stability of the tool. Finally, we would also like to thank the reviewers of this paper for (among other things) their detailed and thoughtful comments: Charles Lakos, Guy Gallasch, Jens Bæk Jørgensen, João Miguel Fernandes, Jonathan Billington, Laure Petrucci, Lin Liu, and Simon Tjell. L. M. Kristensen was supported by the Carlsberg Foundation and the Danish Research Council for Technology and Production. Lisa Wells was supported by the ISIS Katrinebjerg Competence Centre.

## References

1. Adamski, M.A., Karatkevich, A., Wegrzyn, M. (eds.): Design of Embedded Control Systems. Springer, Berlin (2005)
2. Andradóttir, S.: Simulation optimization. In: Banks [3], chap. 9
3. Banks, J. (ed.): Handbook of Simulation. Wiley, New York (1998)
4. Billington, J.: ISO/IEC 15909-1:2004 Software and system engineering. High-level Petri nets, Part 1: Concepts, definitions and graphical notation, 2004
5. Billington, J., Diaz, M., Rozenberg, G. (eds.): Application of Petri Nets to Communication Networks, vol. 1605. Springer, Berlin (1999)
6. Billington, J., Gallasch, G.E., Han, B.: A Coloured Petri Net approach to protocol verification. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. Advances in Petri Nets. In: Proceedings of 4th Advanced Course on Petri Nets, Lecture Notes in Computer Science, vol. 3018 pp. 210–290. Springer, Berlin (2004)
7. BRITNeY Suite. http://www.wiki.daimi.au.dk/britney/
8. Cheng, A., Christensen, S., Mortensen, K.H.: Model checking coloured Petri Nets exploiting strongly connected components. In: Proceedings of International Workshop on Discrete Event Systems, pp. 169–177 (1996)
9. Christensen, S., Kristensen, L.M., Mailund, T.: Condensed state spaces for timed Petri Nets. In: Proceedings of International Conference on Application and Theory of Petri Nets. Lecture Notes in Computer Science, vol. 2075 pp. 101–120. Springer, Berlin (2001)
10. CPN Tools.: http://www.daimi.au.dk/CPNTools/
11. Desrochers, A.A., Al-Jaar, R.Y.: Applications of Petri Nets in Manufacturing Systems: Modeling, Control, and Performance Analysis. IEEE, (1994)
12. Examples of Industrial Use of CP-nets. http://www.daimi.au.dk/CPnets/intro/example_indu.html
13. Gallasch, G.E., Kristensen, L.M.: COMMS/CPN: A Communication Infrastructure for External Communication with Design/CPN. In: Proceedings of Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, DAIMI PB-554, pp. 75–91. Department of Computer Science, University of Aarhus, Denmark (2001)
14. Gnuplot.: http://www.gnuplot.info
15. ITU (CCITT).: Recommendation Z.120: MSC. Technical report, International Telecommunication Union, 1992
16. Jensen, K.: Coloured Petri Nets. Basic concepts, analysis methods and practical use. Basic Concepts, vol. 1. Springer, Berlin (1992)
17. Jensen, K.: Coloured Petri Nets. Basic concepts, analysis methods and practical use. Analysis Methods, vol. 2. Springer, Berlin (1994)
18. Jensen, K.: Condensed state spaces for symmetrical Coloured Petri Nets. Formal Methods in System Design, vol. 9, (1996)
19. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Practical use, vol. 3. Springer, Berlin (1997)
20. Jensen, K., Kristensen, L.M.: Coloured Petri Nets. Modelling and Validation of Concurrent Systems. Springer Textbook (in preparation) Companion web site: www.daimi.au.dk/CPnets/cpnbook.
21. Kelton, W.D., Sadowski, R.P., Sadowski, D.A.: Simulation with Arena, 2nd edn. McGraw-Hill, (2002)
22. Kleijnen, J.P.C.: Experimental design for sensitivity analysis, optimization, and validation of simulation models. In: Banks [3]
23. Kristensen, L.M., Christensen, S., Jensen, K.: The Practitioner's Guide to Coloured Petri Nets. Int. J. Softw. Tools Technol. Transf. **2**(2), 98–132 (1998)
24. Kristensen, L.M., Jørgensen, J.B., Jensen, K.: Application of Coloured Petri Nets in System Development. In: Lectures on Concurrency and Petri Nets. Advances in Petri Nets. Proceedings of 4th Advanced Course on Petri Nets. Lecture Notes in Computer Science, vol. 3098, pp. 626–685. Springer, Berlin (2004)
25. Kristensen, L.M., Mailund, T.: A generalised sweep-line method for safety properties. In: Proceedings of Formal Methods Europe, Lecture Notes in Computer Science, vol. 2391, pp. 549–567. Springer, Berlin (2002)
26. Kristensen, L.M., Valmari, A.: Finding Stubborn Sets of Coloured Petri Nets Without Unfolding. In: Proceedings of International Conference on Application and Theory of Petri Nets. Lecture Notes in Computer Science, vol. 1420, pp. 104–123. Springer, Berlin (1998)
27. Law, A.M., Kelton, W.D.: Simulation Modeling and Analysis, 3rd edn. McGraw-Hill, (2000)
28. Mortensen, K.H.: Efficient data-structures and algorithms for a Coloured Petri Nets Simulator. In: Proceedings of Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, 2001
29. Object Management Group. Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04, 2005
30. Proceedings of International Conference on Application and Theory of Petri Nets and Other Models of Concurrency. Springer, Berlin 1980–present
31. Proceedings of Workshop on Modelling of Objects, Components, and Agents, 2001–present

32. Proceedings of Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, 1998–present. http://www.daimi.au.dk/CPnets/
33. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science, vol. 4 Springer, Berlin (1985)
34. Reisig, W.: Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets. Springer, Berlin (1998)
35. SceneBeans. http://www.dse.doc.ic.ac.uk/Software/SceneBeans/
36. Standard ML of New Jersey. http://www.smlnj.org
37. Ullman, J.D.: Elements of ML Programming. Prentice-Hall, Englewood Cliffs (1998)
38. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models. Lecture Notes in Computer Science, vol. 1491 pp. 429–528. Springer, Berlin (1998)
39. van der Aalst, W., van Hee, K.: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge, MA (2002)
40. Westergaard, M., Lassen, K.B.: The BRITNeY Suite Animation Tool. In: Proceedings of 27th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency. Lecture Notes in Computer Science, vol. 4024 pp. 431–440. Springer, Berlin (2006)
41. Yakovlev, A., Gomes, L., Lavagno, L.: Hardware Design and Petri Nets. Springer, Berlin (2000)