Int J STTT (1998) 2: 98–132

DOI 10.1007/s100099800003

The practitioner's guide to coloured Petri nets

Lars M. Kristensen, Søren Christensen, Kurt Jensen

CPN Group, Department of Computer Science, University of Aarhus, Denmark;
E-mail: {lmkristensen,schristensen,kjensen}@daimi.au.dk

Abstract. Coloured Petri nets (CP-nets or CPNs) provide a framework for the design, specification, validation, and verification of systems. CP-nets have a wide range of application areas and many CPN projects have been carried out in industry, e.g., in the areas of communication protocols, operating systems, hardware designs, embedded systems, software system designs, and business process re-engineering. Design/CPN is a graphical computer tool supporting the practical use of CP-nets. The tool supports the construction, simulation, and functional and performance analysis of CPN models. The tool is used by more than four hundred organisations in forty different countries – including one hundred commercial companies. It is available free of charge, also for commercial use. This paper provides a comprehensive road map to the practical use of CP-nets and the Design/CPN tool. We give an informal introduction to the basic concepts and ideas underlying CP-nets. The key components and facilities of the Design/CPN tool are presented and their use illustrated. The paper is self-contained and does not assume any prior knowledge of Petri nets and CP-nets nor any experience with the Design/CPN tool.

Key words: High-level Petri nets – Coloured Petri nets – Practical use – Modelling – Validation – Verification – Visualisation – Tool support

1 Introduction

An increasing number of system development projects are concerned with distributed and concurrent systems. There are numerous examples, ranging from large scale systems, in the areas of telecommunication and applications based on WWW technology, to medium or small scale systems, in the area of embedded systems.

The development of concurrent and distributed systems is complex. A major reason is that the execution of such systems may proceed in many different ways, e.g., depending on whether messages are lost, the speed of the processes involved, and the time at which input is received from the environment. As a result, distributed and concurrent systems are, by nature, complex and difficult to design and test.

Coloured Petri nets (CP-nets or CPNs) [34–37] provide a framework for the construction and analysis of distributed and concurrent systems. A CPN model of a system describes the states which the system may be in and the transitions between these states. CP-nets have been applied in a wide range of application areas, and many projects have been carried out in industry [36] and documented in the literature, e.g., in the areas of communication protocols [21, 30], audio/video systems [9], operating systems [7, 8], hardware designs [22, 62], embedded systems [59], software system designs [46, 61], and business process re-engineering [49, 56].

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. To achieve this, we have combined the strength of Petri nets [50] with the strength of programming languages. Petri nets provide the primitives for describing synchronisation of concurrent processes, while a programming language provides the primitives for defining data types (colour sets) and manipulating data values.

CPN models can be structured into a number of related modules. This is particularly important when dealing with CPN models of large systems. The module concept of CP-nets is based on a hierarchical structuring mechanism, which supports a bottom-up as well as top-down working style. New modules can be created from existing modules, and modules can be reused in several parts of the CPN model. By means of the structuring mechanism it is possible to capture different abstraction

levels of the modelled system in the same CPN model. A CPN model which represents a high level of abstraction is typically made in the early stages of design or analysis. This model is then gradually refined to yield a more detailed and precise description of the system under consideration.

CPN models are executable. This implies that it is possible to investigate the behaviour of the system by making simulations of the CPN model. Very often, the goal of doing simulations is to debug and validate the system design. However, simulations can equally well serve as a basis for investigating the performance of the considered system.

Visualisation is a technique which is closely related to simulation of CPN models. Observation of every single step in a simulation is often too detailed a level for observing the behaviour of a system. It provides the observer with an overwhelming amount of detail, particularly for very large CPN models. By means of high level visual feedback from simulations, information about the execution of the system can be obtained at a more appropriate level of detail. Another important application of visualisation is the possibility of presenting design ideas and results using application domain concepts. This is particularly important in discussions with people and colleagues unfamiliar with CP-nets.

Time plays a significant role in a wide range of distributed and concurrent systems. The correct functioning of many systems crucially depends on the time taken up by certain activities, and different design decisions may have a significant impact on the performance of a system. CP-nets includes a time concept which makes it possible to capture the time taken by different activities in the system. Timed CPN models and simulation can be used to analyse the performance of a system, e.g., investigate the quality of service (e.g., delay), or the quantity of service (e.g., throughput) provided by the system. The time concept of CP-nets is primarily suited for investigating a system by means of simulations. This contrasts analytical approaches to performance analysis [45], and modelling languages aimed at model checking [15, 47] of timed and hybrid systems [3, 42].

The state space method of CP-nets makes it possible to validate and verify the functional correctness of systems. The state space method relies on the computation of all reachable states and state changes of the system, and is based on an explicit state enumeration [28, 29, 35]. By means of a constructed state space, behavioural properties of the system can be verified. Examples of such properties are the absence of deadlocks in the system, the possibility of always reaching a given state, and the guaranteed delivery of a given service. The state space method of CP-nets can also be applied to timed CP-nets. Hence, it is also possible to investigate the functional correctness of systems modelled by means of timed CP-nets.

Design/CPN [10, 51] is a graphical computer tool supporting CP-nets. It consists of three closely integrated components. The CPN editor supports the construction and editing of hierarchical CPN models. Simulation of CPN models is supported by the CPN simulator, while the CPN state space tool supports the state space method of CP-nets. In addition to these three main components, the tool includes a number of additional packages and libraries. One of these is a set of packages, which supports visualisation by means of business charts, message sequence charts (MSC), and construction of application specific graphics.

The Design/CPN tool is used by more than four hundred organisations in forty different countries – including one hundred commercial companies. It is available free of charge, also for commercial use. More than 50 man-years have been invested in the development of CP-nets and Design/CPN.

CP-nets has been developed over the last 20 years. The main developer has been the CPN group at the University of Aarhus, Denmark, headed by Kurt Jensen. We have developed the basic model, including the use of data types and hierarchy constructs, defined the basic concepts such as dynamic properties, and developed the theory behind many of the existing analysis methods. Together with Meta Software Corporation [16], we have played a key role in the development of high-quality tools supporting the use of CP-nets. Finally, we have been engaged in a large number of application projects, many of these in industrial settings. For more information on the group and its work see [27]. More detailed acknowledgements of some of the individual contributions during the first 15 years can be found in the prefaces of [34–36].

Outline. This paper is organised as follows. Section 2 gives an informal introduction to the syntax and dynamic behaviour of CP-nets. In this section, we also introduce the hierarchical structuring mechanism of CP-nets. Section 3 gives an overview of the Design/CPN tool. Section 4 explains how the construction of CPN models is supported by the CPN editor. Section 5 considers simulation of CPN models and the CPN simulator. Section 6 presents the packages supporting visualisation and gives some examples of their use. Section 7 introduces the time concept of CP-nets and explains how this can be used to conduct simulation based performance analysis of systems. Section 8 introduces the CPN state space tool and the state space method of CP-nets. Section 9 sums up the conclusions and explains how to get started using CP-nets and the Design/CPN tool.

2 Coloured Petri nets

This section gives an informal introduction to the syntax and semantics of CP-nets. A small communication protocol is used throughout the section to illustrate the basic concepts of the CPN language. Later, the communication protocol will also be used to introduce the time concept and the state space method of CP-nets.

One should keep in mind that it is very difficult (probably impossible) to give an informal explanation which is totally complete and unambiguous. Thus it is extremely important for the soundness of the CPN modelling language and the Design/CPN tool that the informal description is complemented by a more formal definition. The formal definition of the syntax and semantics of CP-nets can be found in [32, 34]. For the practical use of CP-nets and Design/CPN, however, it suffices to have an intuitive understanding of the syntax and semantics. This is analogous to programming languages, which are successfully applied by users even though they are unfamiliar with the formal, mathematical definitions of the languages.

2.1 Example: a communication protocol

We consider a stop-and-wait protocol from the datalink control layer of the OSI network architecture. The protocol is taken from [2]. The stop-and-wait protocol is quite simple, and does not in any way represent a sophisticated protocol. However, the protocol is interesting enough to deserve closer investigation, and it is also complex enough for introducing the basic constructs of CP-nets.

Figure 1 gives an overview of the protocol considered. The system consists of a Sender (left) transmitting *data packets* to a Receiver (right). The data packets to be transmitted are located in a Send buffer at the sender side. Communication with the receiver takes place on a bidirectional Communication Channel (bottom).

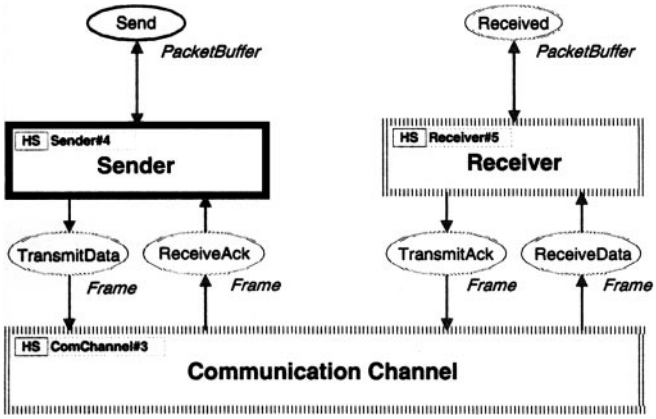

Fig. 1. Stop-and-wait communication protocol

The data packets have to be delivered exactly once and in the correct order into the Received buffer at the receiver side. Obtaining this service is complicated by the fact that the sender and receiver communicate via an unreliable communication channel, i.e., packets may be lost during transmission and packets may overtake each other. One way of achieving the desired service is to use a so-called stop-and-wait retransmission strategy. The idea is that the sender keeps sending the same data packet until a matching *acknowledgement* is received, in which case the next data packet can be transmitted. For simplicity, this particular stop-and-wait protocol uses an unlimited number of retransmissions.

The sender sends a data packet on the communication channel by constructing a *data frame* and putting the data frame in the TransmitData frame buffer. A data frame is a pair consisting of a *sequence number* and a data packet. The channel will then attempt to transmit the data frame, and, if successful, the data frame will be delivered into the ReceiveData frame buffer, where it can be processed by the receiver. The receiver delivers data packets to the upper protocol layers using the data packet buffer Received. The protocol uses sequence numbers to be able to match acknowledgements and data packets, i.e., to be able to deduce which acknowledgements correspond to which data packets, and to be able to deduce whether a given data packet has already been received.

The receiver sends an acknowledgement for a received data frame by constructing an *acknowledgement frame* and putting it in the TransmitAck frame buffer. An acknowledgement frame consists of a sequence number indicating the sequence number of the data packet that the receiver expects next. Similar to data frames, the channel will then attempt to transmit it, and, if successful, it will turn up in the ReceiveAck frame buffer at the sender side, where it can be processed by the sender.

We will return to the inscriptions placed next to the ellipses and in the upper left corner of the boxes later in this section.

2.2 Modelling of states

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

A CP-net is always created as a graphical drawing. Figure 2 shows the CP-net that models the sender in the stop-and-wait protocol. It consists of two parts. The part on the left models the sending of data frames, and the part on the right models the reception of acknowledgement frames. We will explain Fig. 2 in great detail in the following sections which introduce the basic constructs of CP-nets.

Places. The states of a CP-net are represented by means of *places* (which are drawn as ellipses or circles). The sender is modelled using six different places. 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 CP-net (just like the use of mnemonic names in traditional programming). A similar remark applies to the graphical appearance of the places, i.e., the line thickness, size, colour, font, position, etc.

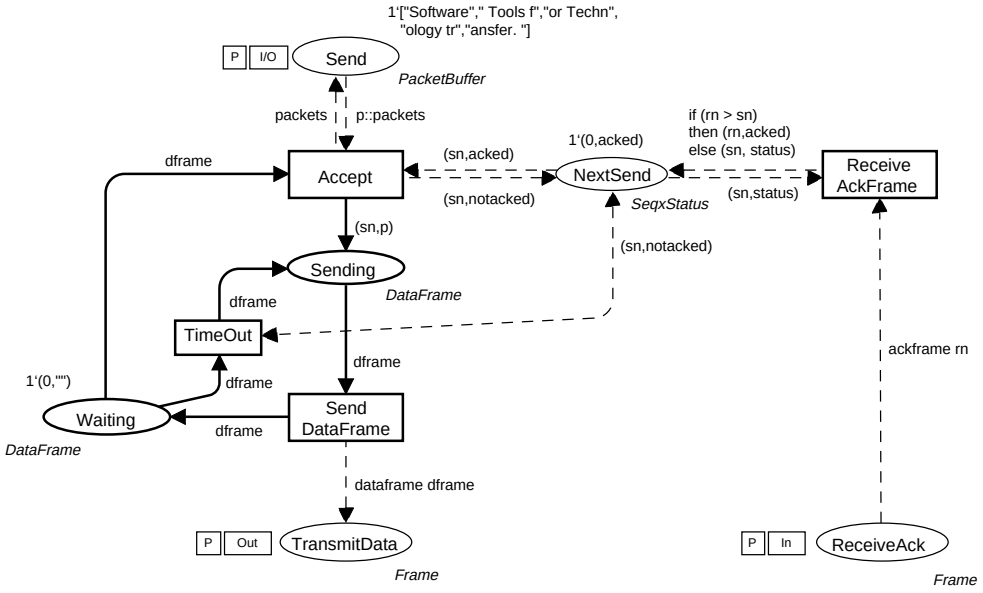

Fig. 2. Sender of the stop-and-wait protocol

It is important to keep in mind that places in contrast to, for example, transition systems, are not used to directly describe the possible states of the CP-net. The state of a CP-net is a so-called marking of the places of the CP-net. We explain the concept of markings after the next paragraph.

The place Send (top left) models the data packet buffer, containing the data packets which the sender has to transmit to the receiver. The places TransmitData and ReceiveAck (at the bottom) model the frame buffers between the sender and the communication channel. The place NextSend (middle) models the internal status of the sender – it will keep track of the sequence number of the data packet to be sent next, and it will indicate whether an acknowledgement has been received for the data packet which is currently being transmitted. The places Sending and Waiting model the two possible positions of the control flow in the sender – either the sender is just about to start Sending a data frame, or the sender is Waiting after having sent a data frame.

Types. Each place has an associated *type (colour set)* determining the kind of data that the place may contain. By convention, the type of a place is written in italics, to the lower left or right of the place. The types are similar to types in a programming language. The types of a CP-net can be arbitrarily complex, e.g., a record where one field is a real, another a text string, and a third a list of integers. Figure 3 shows the *type definitions* used in the stop-and-wait protocol.

We use eight different types to model the sender. The place Send has the type PacketBuffer. This type is defined as a list of Packets representing the possible contents of the packet buffer modelled by the place Send. The type Packet is defined as a string denoting the set of text strings.

```
(* –– data packets –– *)
color Packet = string ;
color PacketBuffer = list Packet ;

(* –– status and sequence numbers –– *)
color Seq = int ;
color Status = with acked | notacked ;

color SeqxStatus = product Seq * Status ;

(* –– data and acknowledgement frames –– *)
color DataFrame = product Seq * Packet ;
color AckFrame = Seq ;

color Frame = union
      dataframe : DataFrame + ackframe : AckFrame ;
```

Fig. 3. Type (colour set) definitions

The type of the place NextSend is SeqxStatus. This type is defined as the product (or pair) of the types Seq and Status. The type Seq is defined as int (integers), and Status is defined as an *enumeration type* containing two possible values: acked and notacked. Hence, the type SeqxStatus contains all pairs in which the first component is an integer (denoting a sequence number) and the second component is either acked or notacked (indicating whether an acknowledgement has been received for the data packet currently being sent).

The places Sending and Waiting have the type DataFrame, which is defined as a product of Sequence numbers and Packets. The places TransmitData and ReceiveAck both have the type Frame, which is defined as the union of type DataFrame and type AckFrame. The type AckFrame is simply a Sequence number.

Markings. A state of a CP-net is called a *marking*. It consists of a number of *tokens* positioned (distributed) on the individual places. Each token carries a *value (colour)*, which belongs to the type of the place on which the token resides. The tokens that are present on a particular place are called the marking of that place. For historical reasons, we sometimes refer to token values as token colours, in the same way as we refer to data types as colour sets. This is a metaphoric picture where we consider the tokens of a CP-net to be distinguishable from each other and hence "coloured" – in contrast to low-level Petri nets which have "black" indistinguishable tokens.

The marking of a place is, in general, a *multi-set* of token values. A multi-set is similar to a set, except that there may be several appearances of the same element. This means that a place may have several tokens with the same token value. As an example, a possible marking of the place TransmitData is the following:

2‘dataframe(0,"CP-nets") + 4‘dataframe(1,"CPN")

This marking contains two tokens with value dataframe(0,"CP-nets") and four tokens with value dataframe(1,"CPN"). By convention, multi-sets are written as a sum (+) using the symbol prime (‘) (pronounced "of") to denote the number of appearances of an element.

Initial marking. A CP-net has a distinguished marking – the *initial marking*, which is used to describe the initial state of the system. The initial marking of a place is, by convention, written on the upper left or right of the place. The place Send has an initial marking consisting of a single token with the value ["Software","Tools f","or Techn","ology Tr","ansfer."], i.e., a list of five packets. The place NextSend initially contains a single token with the value (0,acked), denoting that the packet to be sent first will be assigned sequence number 0, and that an acknowledgement has been received for the previous data packet (since initially there is no previous packet). Initially the sender is in state waiting, as indicated by the initial marking of the place Waiting. Initially, the remaining three places contain no tokens. The specification of the initial marking is therefore (by convention) omitted for these three places.

2.3 Modelling of actions

Transitions. The actions of a CP-net are represented by means of *transitions* (which are drawn as rectangles). As with places, we write the name of the transitions inside the rectangles. The sender consists of four transitions. The transition Accept models the action taken when the next data packet is accepted for transmission. The transition SendDataFrame models the sending of a data frame, and ReceiveAckFrame models the reception of an acknowledgement frame. TimeOut is used to model the occurrence of a timeout so that the data frame can be retransmitted.

Arcs and arc expressions. Transitions and places are connected by *arcs*. The actions of a CP-net consist of *occurrences* of transitions. An occurrence of a transition removes tokens from places connected to incoming arcs (input places), and adds tokens to places connected to outgoing arcs (output places), thereby changing the marking (state) of the CP-net. This is also referred to as the *token game*. As an example, the transition Accept has three incoming arcs and three outgoing arcs. Hence, an occurrence of this transition will remove tokens from the places Waiting, Send, and NextSend, and add tokens to the places Sending, Send, and NextSend.

The exact number of tokens added and removed by the occurrence of a transition, and their data values are determined by the *arc expressions*, which are positioned next to the arcs. How to determine the values of the tokens removed and added will be explained in the next subsection. A double arc, like the dashed arc between TimeOut and NextSend, is shorthand for two opposite directed arcs with identical arc expressions. As we will see in the next section, an action of a CP-net consists, in general, of one or more transitions occurring concurrently.

2.4 Dynamic behaviour

We now describe the dynamic behaviour (operational semantic) of CP-nets. That is, the conditions under which transitions may occur, and the effect of an occurrence of a transition on the marking of the CP-net.

Variables and bindings. To talk about an occurrence of a transition, we need to assign (bind) data values to the (free) *variables* occurring in the arc expressions on the surrounding arcs of the transition. Otherwise, it is impossible to *evaluate* the arc expressions. The transition Accept has four variables: p of type Packet, packets of type PacketBuffer, sn of type Seq, and dframe of type DataFrame. The *variable declarations*, specifying the variables and their types, are shown in Fig. 4. Variables can be assigned data values belonging to the type of the variable.

```
(* —– data packets —– *)
var p : Packet ;
var packets : PacketBuffer ;

(* —– status and sequence numbers —– *)
var sn,rn : Seq ;
var status : Status ;

(* —– data frames —– *)
var dframe : DataFrame ;
```

Fig. 4. Variable declarations for the stop-and-wait protocol

Let us now assume that we assign data values to the variables of the transition Accept by creating the *binding* listed in Fig. 5, where ← should be read "bound to".

p	←	"Software"
packets	←	[" Tools f","or Techn","ology Tr","ansfer. "]
sn	←	0
dframe	←	(0,"")

Fig. 5. A binding of transition Accept

Figure 6 shows, for each surrounding arc of Accept, the multi-set of tokens resulting from evaluating the corresponding arc expressions in the binding listed in Fig. 5. In this case all multi-sets contain a single token. For all arc expressions, the result is straightforward. The only exception is the arc expression p::packets on the incoming arc from Send. The infix operator :: is the basic list constructor. This means that the result of evaluating the expression p::packets in the binding of Fig. 5 is the list obtained by inserting the value "Software", bound to p, at the front of the list bound to packets.

Figure 6 also shows the initial marking of the places surrounding the transition Accept. The marking of each place is indicated next to the place. The number of tokens on the place is shown in the small circle, while the detailed token values are indicated in the dashed box next to the small circle.

A binding of a transition is also written in the form: $\langle v_1 = d_1, v_2 = d_2, \ldots v_n = d_n \rangle$ where v_i for $i \in 1..n$ is a variable, and d_i is the value assigned to v_i. As an example, the binding in Fig. 5 can be written as:

⟨p = "Software", sn = 0, dframe = (0,""),
packets = [" Tools f","or Techn","ology Tr","ansfer. "]⟩

In addition to the arc expressions, it is possible to attach a Boolean expression (with variables) to each transition. The Boolean expression is called a *guard*. It specifies that we only accept bindings for which the Boolean expression evaluates to true. However, none of the transitions of the sender uses a guard.

Enabling. In order for a transition to be *enabled* in a marking, i.e., ready to *occur*, it must be possible to *bind (assign)* data values to the variables appearing on the surrounding arc expressions and in the guard of the transition such that: 1) each of the arc expressions evaluate to tokens which are present on the corresponding input place; and 2) the guard (if any) is satisfied.

Figure 6 tells us that an occurrence of the transition Accept (with the binding in Fig. 5) removes a token with value (0,"") from the place Waiting, a token with value ["Software"," Tools f", "or Techn","ology Tr","ansfer. "] from the place Send, and a token with value (0,acked) from the place NextSend.

This binding of Accept is enabled in the initial marking, since the tokens to which the arc expressions evaluate are present on the corresponding input places.

Obviously, there are many other bindings that we may try for the transition Accept in the initial marking. However, none of these are enabled in the initial marking. This can be seen as follows. The place Send has only one token. Hence we need to bind p to the head of the list, and packets to the tail of the list. The place NextSend also contains a single token with value (0,acked). Hence, we need to bind sn to 0. The place Waiting also contains a single token with value (0,""), hence we need to bind dframe to this value. Therefore the enabled binding is uniquely determined by the tokens residing on the input places.

The possible bindings of data values to the variables of a transition correspond to the possible ways in which a transition can occur. However, as we demonstrated above, only a subset of these will, in general, be enabled in a given marking.

Occurrence. The occurrence of a transition in an enabled binding removes tokens from the input places and adds tokens to the output places of the transition. The values of the tokens removed from an input place are determined by evaluating the arc expression on the corresponding input arc. Similarly, the values of tokens added to an output place are determined by evaluating the arc expression on the corresponding output arc (see Fig. 6). Figure 7 shows the marking of the surrounding places of the transition Accept resulting from the occurrence of this

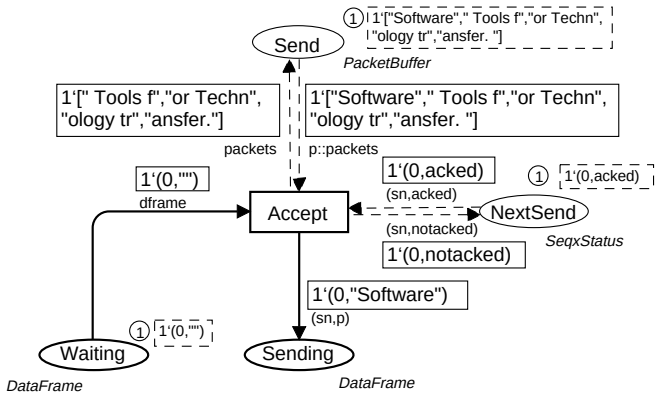

Fig. 6. Evaluation of arc expressions for Accept

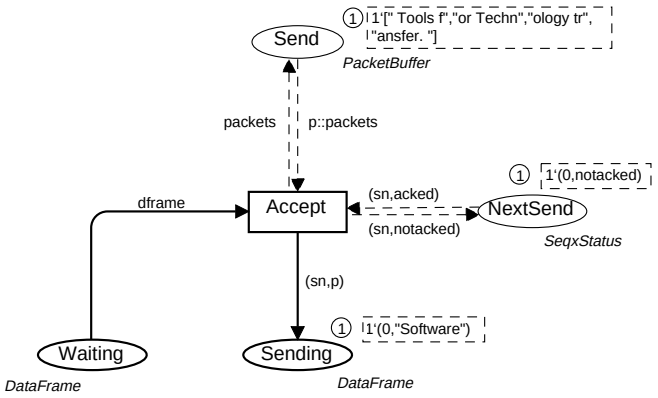

Fig. 7. Marking resulting from the occurrence of Accept

transition in the initial marking with the enabled binding listed in Fig. 5.

Hence, the occurrence of the transition Accept has the effect that the data packet at the head of the list, residing on the place Send, is removed. The status of the sender, modelled by NextSend, is updated so that the token, residing on NextSend, indicates that an acknowledgement has not yet been received (notacked) for the data packet, which is now being transmitted. The control flow of the sender is changed from being in the state waiting to sending, by removing the token, corresponding to the data frame previously being transmitted from the place Waiting, and adding a token to the place Sending corresponding to the newly constructed data frame, which is now being transmitted.

We now briefly sum up the remaining parts of the CP-net in Fig. 2. The transition TimeOut is enabled when the data frame currently being sent is on Waiting, and an acknowledgement has not yet been received (status is notacked) for the data frame currently being sent. An occurrence of TimeOut changes the control in the sender from being waiting to sending, by removing the data frame on Waiting and adding it to the place Sending so that it can be retransmitted. At first glance, it may seem strange that we do not specify the conditions under which retransmissions occur. However, for a lot of purposes this is not necessary. Most CP-nets are used to investigate the logical and functional correctness of a system design. For this purpose it is often sufficient to describe that retransmissions may appear, e.g., because the communication channel is slow. However it may not be necessary, or even beneficial, to consider how often this happens – the protocol must be able to cope with all kinds of communication channels, both those which work so well that there are no retransmissions, and those in which retransmissions are frequent. Later on we will see that CP-nets can be extended with a time concept that allows us to describe the duration of the individual actions and states. This will permit us to investigate the performance of the modelled system, i.e., how fast and effectively it operates. Then we will give a much more precise description of retransmissions (e.g., that they occur when no acknowledgement has been received inside two hundred milliseconds).

The reception of acknowledgement frames is modelled by ReceiveAckFrame. The occurrence of this transition removes an acknowledgement frame from the place ReceiveAck and compares the sequence number rn in the acknowledgement frame with the sequence number sn of the data frame currently being sent. If the sequence number in the acknowledgement is greater than the sequence number of the data frame currently being sent (recall that the receiver sends the sequence number of the data frame it expects next), then the sequence number of the next data packet is updated, and the acknowledgement status is changed to indicate that an acknowledgement has been received for the data frame currently being sent. This is accomplished by the if-then-else construction used on the arc from ReceiveAckFrame to NextSend, which puts a token on NextSend with a value according to the above description.

Occurrence sequences and steps. An execution of a CP-net is described by means of an *occurrence sequence*. It specifies the markings that are reached and the *steps* that occur. Above, we have seen an occurrence sequence of length one. It consisted of a single step, the occurrence of Accept in the binding in Fig. 5 in the initial marking, and leading to the marking in Fig. 7.

In this marking, the transition SendDataFrame is enabled in a binding in which the variable dframe is assigned the value (0,"Software"). Hence, the occurrence sequence can be continued with a step corresponding to the occurrence of this binding of SendDataFrame. This leads to a new marking in which the marking of Send and ReceiveAck remains unchanged, one token with value dataframe(0,"Software") is on TransmitData (corresponding to the data frame being placed in the TransmitData buffer for transmission), and one token with value (0,"Software") is on Waiting.

In general, a step may consist of several enabled binding elements occurring *concurrently*. A *binding element* is a pair consisting of a transition and a binding of its variables. As an example, consider the marking of the sender shown in Fig. 8. This marking is identical to the marking reached after the occurrence of Accept in the initial marking, except that two tokens are residing on the place ReceiveAck; one token with value ackframe(0), and one token with value ackframe(1). Moreover, the token present on place NextSend now indicates that an acknowledgment has been received. In this marking, the three binding elements listed below are enabled.

1. $(\mathsf{SendDataFrame}, \langle \mathsf{dframe} = (0, \text{"Software"}) \rangle)$
2. $(\mathsf{ReceiveAckFrame}, \langle \mathsf{rn} = 0, \mathsf{sn} = 0, \mathsf{status} = \mathsf{acked} \rangle)$
3. $(\mathsf{ReceiveAckFrame}, \langle \mathsf{rn} = 1, \mathsf{sn} = 0, \mathsf{status} = \mathsf{acked} \rangle)$

The binding elements 1 and 2 are *concurrently enabled*, since each binding element can get those tokens that it needs (i.e., those specified by the input arc expressions) – without sharing the tokens with the other binding element. The same is true for the binding elements 1 and 3. However, the binding elements 2 and 3 are *not* concurrently enabled, since they cannot both get the only (0,acked) token on NextSend. These two binding elements are in *conflict*.

In general, it is possible for a transition to be concurrently enabled with itself (using two different bindings or using the same binding twice). Hence, a step, from one marking to the next, may involve a multi-set of binding elements. The marking, resulting from the occurrence of a step consisting of several binding elements, is the same as the marking reached by the occurrence of the individual binding elements in some arbitrary order. This marking is well-defined since the marking resulting from the occurrence of a multi-set of concurrently enabled binding

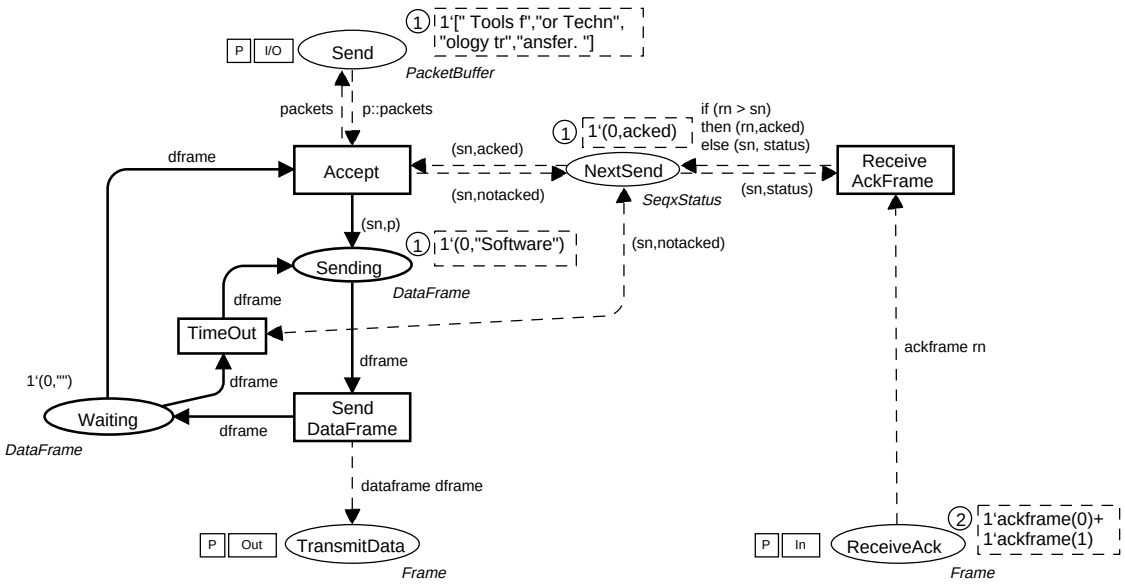

Fig. 8. Marking of the sender

elements, is independent of the order in which the individual binding elements occur.

A *finite occurrence sequence* is an occurrence sequence consisting of a finite number of markings and steps. An *infinite occurrence sequence* is an occurrence sequence consisting of an infinite number of markings and steps. An infinite occurrence sequence corresponds to a non-terminating execution of the system.

2.5 Hierarchical CP-nets

The basic idea underlying hierarchical CP-nets is to allow the modeller to construct a large model by using a number of small CP-nets like the one in Fig. 2. These small CP-nets are called *pages*. These pages are then related to each other in a well-defined way as explained below. This is similar to the situation in which a programmer constructs a large program by means of a set of modules. Many CPN models consist of more than one hundred pages with a total of many hundred places and transitions. Without hierarchical structuring facilities, such a model would have to be drawn as a single (very large) CP-net, and it would become totally incomprehensible.

In the previous section, we described a CPN model for the sender. In this section we describe models for the receiver and the communication channel, and we show how these three submodels can be put together to form a model of the entire stop-and-wait protocol.

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

Substitution transitions and subpages. We start out by working in a top-down manner. The most abstract CP-net describing the stop-and-wait protocol is the one shown in Fig. 1. Here we have three transitions: Sender, Receiver, and Communication Channel. Each of these transitions is marked with an *HS-tag* (in the upper left corner) indicating that it is a *substitution transition* (HS = Hierarchy + Substitution). The dashed boxes next to the HS-tags are called *hierarchy inscriptions* and they define the details of the substitutions.

Each hierarchy inscription specifies the *subpage*, i.e., the CP-net that contains the detailed description of the activity represented by the corresponding substitution transition. The subpage corresponding to the substitution transition Sender is named Sender and it is the CP-net shown in Fig. 2. The Receiver has a subpage named Receiver and the Communication Channel has a subpage named ComChannel. We will return to the latter two pages later in this subsection, where we explain how the receiver and the communication channel are modelled.

Port and socket places. Each subpage has a number of places which are marked with an In-tag, Out-tag, or I/O-tag. These places are called *port places* and they constitute the interface through which the subpage communicates with its surroundings. Through the *input ports*

(marked with an In-tag) the subpage receives tokens from the surroundings. Analogously, the subpage delivers tokens to the surroundings through the *output ports* (marked with an Out-tag). A place with an I/O-tag is both an input port and an output port at the same time. The page Sender has one input port ReceiveAck, one output port TransmitData, and one input/output port Send.

The substitution transition Sender in Fig. 1 has one input place ReceiveAck, one output place TransmitData, and one input/output place Send. These places are called *socket places*. More precisely, ReceiveAck is an *input socket* for the Sender transition, TransmitData is an *output socket*, while Send is an *input/output socket*. To specify the relationship between a substitution transition and its subpage, we must describe how the port places of the subpage are related to the socket places of the substitution transition. This is achieved by providing a *port assignment*. For the Sender subpage, we relate the input port ReceiveAck in Fig. 2 to the input socket ReceiveAck in Fig. 1. Analogously, the output port TransmitData in Fig. 2 is related to the output socket TransmitData in Fig. 1, and the input/output port Send to the input/output socket Send.

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

The communication channel. Let us now consider the communication channel. The communication channel is required to be an unreliable *bidirectional* communication channel. We will now work in a bottom-up manner, modelling an unreliable *unidirectional* channel and then compose two such unidirectional channels to form a bidirectional channel.

The CP-net modelling a unidirectional channel is shown in Fig. 9. It consists of two places Incoming and Outgoing, and a single transition Transmit. The input port place Incoming models the incoming traffic which is to be transmitted on the channel. The output port place Outgoing models the outgoing traffic which has been transmitted successfully on the channel. Both places have the type Frame since the channel will be transmitting frames. An occurrence of the transition Transmit removes a frame from the place Incoming. Depending on the binding of the Boolean variable success (which may either be true or false), the frame is either put on the place Outgoing, corresponding to a successful transmission, or the frame is not put on the place Outgoing, corresponding to a loss of the frame. The constant empty denotes the empty multi-set of tokens.

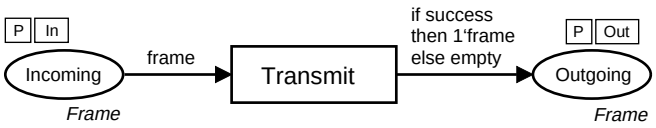

Fig. 9. Unidirectional channel

The CP-net modelling the unreliable bidirectional communication channel is shown in Fig. 10. The CP-net has two substitution transitions: DataChannel and AckChannel.

DataChannel models the traffic in the sender-to-receiver direction. AckChannel models the traffic in the receiver-to-sender direction. Both substitution transitions have the CP-net modelling the unidirectional channel as subpage. This means that we have reused the CP-net of the unidirectional channel. During the execution of the CP-net, we will have two separate *page instances* of the CP-net modelling the unidirectional channel; one instance corresponding to the data channel, and one corresponding to the acknowledgement channel. Each of these page instances will have its own marking which is totally independent of the marking of the other page instances (in a similar way to procedure calls having private copies of local variables).

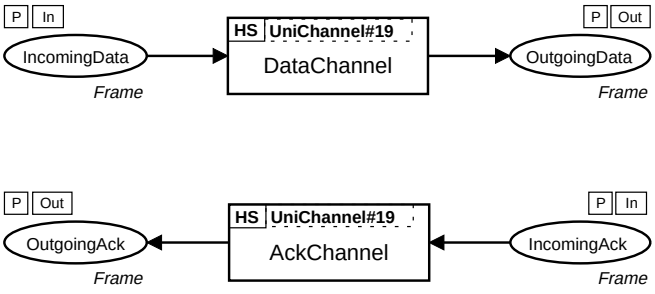

Fig. 10. Bidirectional channel

The port assignment for the socket places in Fig. 10 is as one would expect. The two input socket places IncomingData and IncomingAck are assigned to the port place Incoming in Fig. 9. The two output socket places OutgoingData and OutgoingAck are assigned to the port place Outgoing in Fig. 9.

The CP-net modelling the bidirectional channel is the subpage of the substitution transition Communication Channel in Fig. 1. It is worth observing that the places IncomingData, OutgoingData in Fig. 10 are sockets with respect to the substitution transition DataChannel while they are port places with respect to the substitution transition Communication Channel. A similar remark applies to IncomingAck and OutgoingAck.

The receiver. Figure 11 depicts the CPN model of the receiver. The right hand side models the reception of data

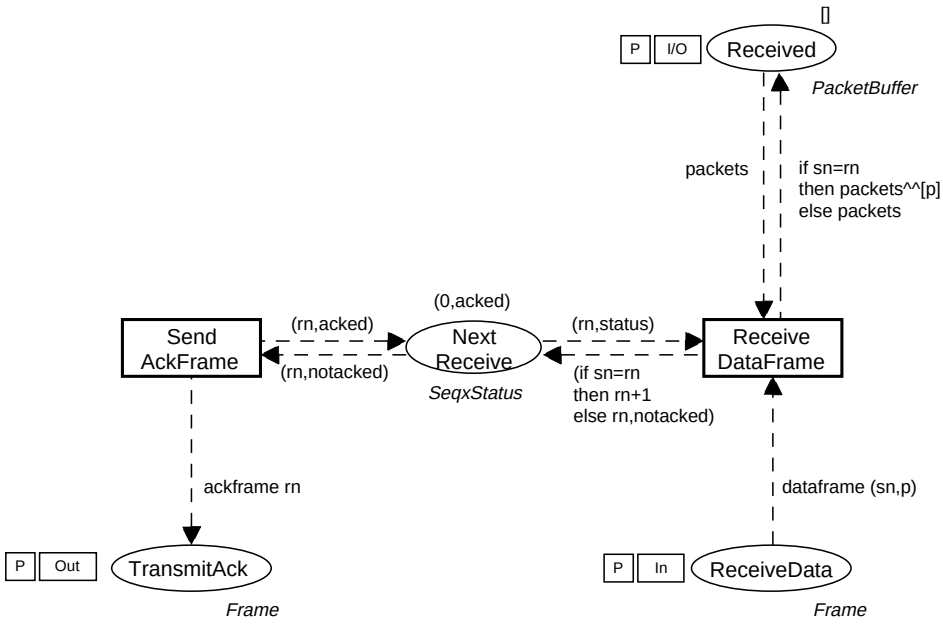

Fig. 11. Receiver of the stop-and-wait protocol

frames while the left hand side models the sending of acknowledgements. The place NextReceive (middle) is similar to the place NextSend at the sender page. It is used to keep track of the state of the receiver. It specifies the sequence number of the data frame which is expected next, and it indicates whether an acknowledgement has been sent for the last received data frame. Initially, the receiver expects the data frame with the sequence number 0, and the previous data frame has been acknowledged (since, initially, there is no such data frame). This is specified by the initial marking (0,acked) of the place NextReceive. The place Received models the data packet buffer on the receiver side. Initially, the buffer is empty as indicated by the initial marking [ ], denoting the empty list. The port places TransmitAck and ReceiveData are like the similarly named places at the sender side in that they model the frame buffers between the receiver and the communication channel.

The transition ReceiveDataFrame models the receipt of data frames. When a data frame is received, it is removed from the place ReceiveData and its sequence number sn is compared with the sequence number rn of the data frame that the receiver expects next. If the sequence numbers match (rn=sn), then the packet p is appended to the list residing on the place Received and the sequence number of the data frame expected next is incremented by one. In any case, the state of the receiver is updated to be notacked.

The transition SendAckFrame models the sending of acknowledgement frames. When the internal state of the receiver is notacked, then an acknowledgement frame with sequence number rn is put on place TransmitAck, and the internal state of the receiver is updated to be acked, indicating that an acknowledgement has been sent.

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

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

There are three different kinds of fusion sets: *global fusion sets* can have members from many different pages, while *page fusion sets* and *instance fusion sets* only have members from a single page. The difference between the last two is the following. A page fusion unifies all the instances of its places (independently of the page instance at which they appear), and this means that the fusion set only has one "resulting place" which is "shared" by all instances of the corresponding page. In contrast, an instance fusion set only identifies the place instances that belong to the same page instance, and this means that the fusion set has a "resulting place" for each page instance. The semantics of a global fusion set is analogous to that

of a page fusion set – in the sense that there is only one "resulting place" (which is common for all instances of all participating pages). To obtain the benefits of modular design and analysis, global fusion sets should be used sparingly.

In the protocol example, we only have three levels in the page hierarchy. However, in practice, there are often up to ten different hierarchical levels. As we saw with the page modelling the bidirectional channel, a subpage may contain substitution transitions and thus have its own subpages. Very often, a page both has transitions and substitution transitions, i.e., some activities are described in full detail, while other activities are described in a coarser way – deferring the detailed description to a subpage.

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

It should be noted that substitution transitions never become enabled and never occur. Substitution transitions work as a macro mechanism. They allow subpages to be conceptually inserted at the position of the substitution transitions – without doing an explicit insertion in the model. In Fig. 1, we have not provided any arc expressions for the arcs that surround the substitution transitions. These are unnecessary since the substitution transitions never become enabled or occur. Nevertheless, they can be very useful in giving the reader of a model a first impression of the functionality of the subpage.

It is important to notice that the hierarchical constructs do not guarantee a formal relationship in terms of behavioural equivalence between arc expressions on arcs surrounding a substitution transition, and the behaviour of the corresponding subpage. This means that the hierarchical concepts of CP-nets offers abstraction [43] at the syntactic rather than at the semantic level.

3 Overview of the Design/CPN tool

In this section, we give an overview of the Design/CPN tool by describing its main components and the general way in which to apply the tool.

The overall architecture of the Design/CPN tool is shown in Fig. 12. Design/CPN consists of two main components: the graphical user interface (GUI) and CPN ML. Together, these two components span the three tightly integrated tools which constitute Design/CPN: the CPN editor, the CPN simulator, and the CPN state space tool.

The GUI is the graphical front-end with which the user interacts. It is built on top of the general graphical package Design/OA [17]. The CPN ML part implements the CPN ML language, which is the programming language used for declarations of variables, declaration of types, and net inscriptions (e.g., arc expressions, guards, etc.) in CP-nets. Figures 3 and 4 from the previous section are examples of type and variable declarations written in CPN ML. The CP-nets from the previous sections also give a number of examples of arc expressions written in CPN ML. In addition to the CPN ML language, the CPN ML component implements the simulation engine and the central algorithms and data structures for the generation and use of state spaces. This means that the CPN ML component is responsible for the calculation of enabled binding elements in the encountered markings of the CPN model.

CPN ML is built on top of the Standard ML (SML) language [48]. The CPN ML language is the SML language extended with some syntactical sugar to simplify the declaration of types, variables, etc. The fact that the inscription language is based on SML has several advantages. First, the expressiveness of SML is inherited by De-

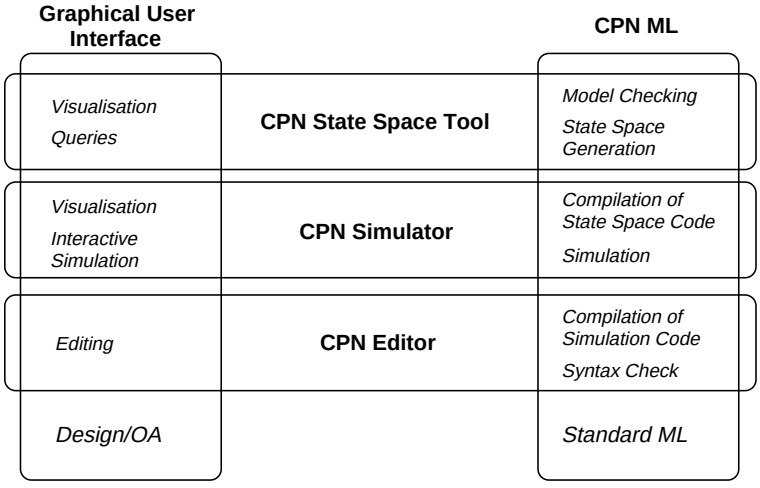

Fig. 12. Architectural overview of Design/CPN

sign/CPN. Second, SML is strongly typed, allowing many modelling errors to be caught early on. A third virtue is that polymorphism, overloading, and definition of infix operators in SML allow net inscriptions to be written in a natural, mathematical syntax. Finally, SML is well documented, tested, and maintained [1, 53, 63]. The choice of SML has turned out to be one of the most successful design decisions for Design/CPN. It has proven advantageous to build Design/CPN upon Design/OA and SML, which are both available on different platforms, since platform dependency is isolated in these building blocks and not in the tool itself.

For the inscriptions of the CP-net (e.g., arc expressions, guards, etc.) only the purely functional part of the SML language is used. This is consistent with the operational semantics of CP-nets, as described in Sect. 2.4. It would not make sense if, for example, evaluation of a guard for a transition might have an impact on the marking of some places.

The usual working style, when applying the tool, is to start out using the editor for constructing a first CPN model of the system under consideration. The hierarchical CP-net from the previous section (Fig. 1) modelling the stop-and-wait protocol, is an example of a *CPN diagram* constructed using the CPN editor. The figures from the previous section are screen dumps from the editor. The user works directly on the graphical representation of the CPN model. In order to be able to simulate the CPN model, it has to constitute a syntactical and type correct CP-net. When reported syntax and type errors (if any) have been corrected, the user can *switch* to the CPN simulator. The switch to the CPN simulator generates the code necessary for simulation of the CPN model. The syntax check as well as the compilation is handled by the CPN ML part of the editor. The CPN editor is described in more detail in Sect. 4.

Once the code for simulation has been compiled, the user is ready to start investigating the behaviour of the system by means of simulations. These first simulations typically have the characteristics of single step debugging in which the token game is observed in great detail, and the user chooses the next binding elements to occur. During such simulations, the markings of the places are shown directly on the CPN diagram similar to Fig. 8. The simulations typically reveal some shortcomings and/or errors in the CPN model which then have to be resolved. Hence, the first phase normally consists of a number of iterations switching back and forth between the editor and the simulator, gradually refining and improving the CPN model. The simulation/execution of the CPN model is driven by the simulator engine of CPN ML. It uses the simulator part of the GUI for visualisation of the token game. The CPN simulator will be considered in more detail in Sect. 5.

Usually, the next phase is to make some lengthy simulations to conduct a more elaborate validation of the design and/or the CPN model. In such lengthy simulations, the visualisation of the token game is typically replaced with some higher and more abstract way of visualising the behaviour of the system. Examples of this include business charts, message sequence charts (MSC), and various kinds of graphics, specific for the application domain. The libraries for visualisation are considered in Sect. 6.

In case the purpose of creating the CPN model is to make performance analysis, the CPN simulator is configured to collect data during the lengthy simulations. The collected data can then be used later in a post-processing phase in which the key figures for the performance of the system are obtained. The time concept of CP-nets and the support for simulation based performance analysis are described in Sect. 7.

A possible next phase is to apply the state space tool to verify and validate the functional correctness of the system. In order to apply the CPN state space tool, the user switches from the CPN simulator to the CPN state space tool. The switch is similar to the one from the editor to the simulator, and consists of a compilation of the necessary internal data structures for working with the state space tool. This compilation is handled by the simulator part of CPN ML. The first phase of applying the state space tool typically consists of making the CPN model tractable for state space analysis. The next step is then to generate the state space (or part of it). Generation is handled by the CPN ML part of the state space tool. The user can now make queries about the behaviour of the system, using the available query languages. Queries can be written as formulas in a state and action oriented variant of the temporal logic CTL [6, 15] or by using a number of available search and traversal functions. The answers to these queries are calculated by the model checker in the CPN ML part of the state space tool. It uses the GUI for visualising information about the state space and for displaying the results of queries. The state space method and the CPN state space tool are considered in more detail in Sect. 8.

The GUI and CPN ML run as two separate communicating processes. Communication between the two processes is needed in order to transform the graphical representation in the GUI into abstract/internal representations used in CPN ML and vice versa. Having the GUI and CPN ML as two separate processes has several advantages. The main advantage comes from the fact that CPN ML handles the syntax check, code generation, simulation, and state space generation which are the parts that require the bulk of the computational resources in terms of memory and speed. The GUI has rather modest requirements with respect to computational resources. Since the two parts are separate communicating processes, the GUI can run on a small workstation, and CPN ML can run in the background on a more powerful workstation. In most of the industrial projects in which the Design/CPN tool has been applied, a workstation equipped with 64 MB internal memory and with a speed equivalent to that of a Sun Ultra Sparc has been

suitable for running the CPN ML part. However, the state space tool often requires additional computational resources in terms of memory.

4 Construction of CPN models

The CPN editor supports construction, editing, and syntax check of CPN diagrams. In this section, we describe the basic functionality of the editor.

4.1 Working with Hierarchical CPN models

In typical industrial applications, a CPN diagram consists of 10–100 pages with varying complexity. A modeller must find a suitable way in which to divide the model into pages. Moreover, the modeller must find a suitable balance between declarations, net inscriptions, and net structure (i.e., places, transitions, and arcs).

The hierarchy page. The editor provides an overview of the pages in the CPN model and their interrelationship by automatically creating and maintaining a so-called *hierarchy page*, which is similar to the project browsers found in many conventional programming environments. The hierarchy page for the stop-and-wait protocol is shown in Fig. 13.

The hierarchy page has a *page node* for each page. An arc between two page nodes indicates that the latter is a subpage of the former, i.e., that the source page contains a substitution transition that uses the destination page as subpage. Each page node is inscribed with text that specifies the page name and the page number. Analogously, each arc has text that specifies the name of the substitution transition in question. As an example, page node SWProtocol represents the page shown in Fig. 1. The page node has three outgoing arcs corresponding to the three substitution transitions Sender, Receiver, and Communication Channel. These three arcs leads to the page nodes representing the pages Sender (shown in Fig. 2), Receiver (shown in Fig. 11), and ComChannel (shown in Fig. 10), respectively. The page node representing page ComChannel has one outgoing arc leading to the page node representing page UniChannel (shown in Fig. 9). This arc corresponds to the two substitution transitions DataChannel and AckChannel.

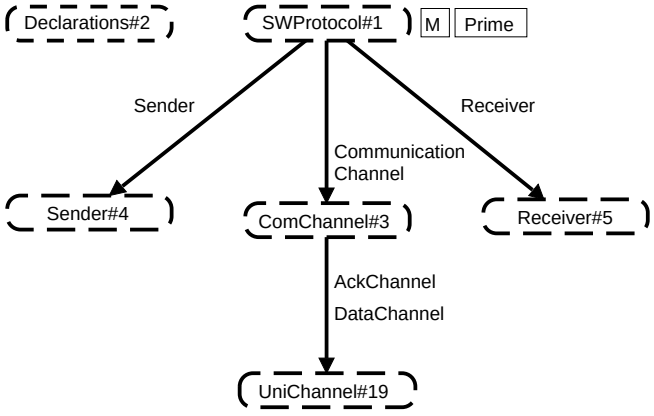

Fig. 13. Hierarchy page for the stop-and-wait protocol

When the user double clicks on a page node, the corresponding page is opened and displayed, and the user can start editing the page. A double click on a substitution transition similarly opens and puts the corresponding subpage in front.

The page SWProtocol has a small Prime tag next to it. This indicates that SWProtocol is a *prime page*, i.e., a page on the most abstract level. A CPN model has a page instance for each prime page. For each substitution transition on a prime page we get a page instance of the corresponding subpage. If these page instances have substitution transitions, we get page instances for these, and so on, until we reach the bottom of the page hierarchy (which is required to be acyclic). The CPN model of the stop-and-wait protocol has six page instances. All pages have one instance, except UniChannel, which has two instances, one for each of the two substitution transitions on page ComChannel.

The CPN model also has a page named Declarations. This page contains the *global declaration node*. The global declaration node contains all declarations of types, constants, functions, and variables, used in the net inscriptions of the CPN model. For the stop-and-wait protocol, the global declaration node contains the type and variable declaration shown in Figs. 3 and 4.

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

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

As may be seen, a lot of rather complex checks, calculations, and manipulations are involved in the Move to Subpage command. However, almost all of these are automatically performed by the editor. The user only selects the subnet, invokes the command, and creates the new

substitution transition. The rest of the work is done by the editor. This is, of course, only possible because the editor recognises a CPN diagram as a hierarchical CP-net, and not just as a mathematical graph or as a set of unrelated objects. Without this property, the user would have to do all the work by means of the ordinary editing operations (copying, moving, and creating the necessary objects). This would be possible – but it would be much slower and much more error-prone.

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

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

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

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

4.2 Flexible graphics

In order to be able to create easily readable CPN models, Design/CPN supports a wide variety of graphical formatting parameters such as shapes, shading, borders, etc. The underlying formal CPN model (in CPN ML of Fig. 12) is unaffected by the graphical appearance. For example, an object created as a place remains a place forever, independent of graphical changes. Flexible graphics in Design/CPN are accompanied by sensible defaults, e.g., the default shape of a place is an ellipse.

Figure 2 illustrates the use of flexible graphics. The main flow of the sender is Accept or TimeOut, Sending, SendDataFrame, and Waiting. This is indicated by the thick border of these places and transitions, and the thick arcs in between. The places modelling buffers and variables in the sender have been given a thin border. All arcs modelling access to these variables and buffers are thin and dashed.

The user can modify the default settings for the graphical appearance of objects using *system defaults* and *diagram defaults*. The scope of diagram defaults is a single diagram. If the user sets the diagram defaults for a specific object type, e.g., places, then all places created in this diagram will be created using the diagram default. The system defaults work across diagrams and determine the initial diagram defaults for new CPN diagrams. The system defaults make it easy for a user to have a certain style in which CPN diagrams are created. The diagram defaults make it simple for a user to modify a CPN diagram created by another user, and still ensure that the graphical appearance of the modified and added objects is consistent with the rest, i.e., the original part of the CPN diagram.

4.3 Syntax and type checking

The editor is syntax directed by enforcing a number of built-in syntax restrictions. This prevents the user from making certain errors during the construction of a model. As an example, it is impossible to draw an arc between two transitions or between two places. However, it is impossible to catch all errors efficiently that way. Hence, there is a syntax and type checker, which can be invoked when the user wants to ensure that the created model constitutes a legal CP-net.

An example of a syntax error would be a missing type of a place. An example of a type error would be an arc expression with a type which is different from the type of the place connected to the arc. Several errors can be reported at the same time during a syntax check.

Reporting errors is based on a hypertext facility. We illustrate this by explaining how syntax errors on page Sender (shown in Fig. 2) is reported. On the hierarchy page (Fig. 13), an *error box* will appear. The error box will contain a text stating that there is an error on page Sender, and there will be a *hypertext link* pointing to the page. Following this link will open page Sender and select another error box with a description of the problem in the form of an error message. As a part of the message another hypertext link is provided, which points to the object, e.g., place or transition, where the error is located.

In many cases, correcting errors only involves local changes. For efficiency reasons, the syntax and type check is incremental. This means that only the modified part of the model is checked again – not the entire model. For example, assume that all five modules of the CPN model depicted in Fig. 13 have been syntax checked. When a syntax error regarding, say, the transition Accept on page Sender has been fixed, only that transition and its surrounding arcs and places are rechecked, not all five modules.

4.4 Textual interchange format

CPN diagrams of hierarchical CP-nets can be imported and exported in a textual format [44]. Figure 14 gives an example of the textual format supported by Design/CPN. It shows how the transition Transmit on page UniChannel in Fig. 9 is represented in the textual format. The transition Transmit is represented using a trans *block*. The *tags* and *attributes* inside this block give information about the name of the transition and its graphical appearance such as the lineattributes, fillattributes, and textattributes. Hence, the textual format gives information about the CP-net itself, i.e., transition, places, arcs and net inscriptions, as well as the graphical layout of the CP-net.

The textual format is based on *SGML (Standard Generalised Markup Language)* [26, 57] on which a number of markup languages such as *HTML (HyperText Markup Language)* are based. This is the reason why the textual format as shown in Fig. 14 looks very similar to HTML. The textual format has been developed with emphasis on conforming to the committee draft [20] of the forthcoming standard for high-level Petri nets.

```
<trans id=id15>
  <text>Transmit</text>
  <lineattr type=Solid thick=2 colour=black>
  <fillattr pattern=None colour=black>
  <posattr x=-14.01390 y=1135.58337>
  <box h=254.00000 w=779.63892>
  <textattr font=Helvetica size=18 just=Centered
            colour=black bold=FALSE italic=FALSE
            underlin=FALSE outline=FALSE
            shadow=FALSE
            condense=FALSE extend=FALSE
            scbar=TRUE>
</trans>
```

Fig. 14. Example of the textual interchange format

5 Simulation of CPN models

The CPN simulator supports execution of CPN models. It provides two fundamental modes of simulation suitable for different purposes as explained below.

5.1 Interactive simulation

As the individual parts of a CP-net are constructed, they are investigated and debugged by means of the CPN simulator, just as a programmer tests and debugs new parts of a program. In the early phases of a modelling process, the user typically wants to make a detailed investigation of the behaviour of the individual transitions or small parts of the model.

For this purpose, the simulator offers an *interactive mode*. Here the user is in full control, sets breakpoints, chooses between enabled binding elements, changes markings of places, and studies the token game in detail. This work mode is similar to single step debugging in an ordinary programming language. The purpose is to see whether the individual net components work as expected. Interactive simulations are, by nature, very slow – no human being can investigate more than a few markings per minute.

As indicated above, the modeller is able to inspect all details of the markings reached. The user can see the set of enabled transitions and select the binding elements to occur. In Fig. 15, a screen dump from an interactive simulation of the stop-and-wait protocol is depicted. It shows the marking of the sender page.

The current marking is indicated: the number of tokens on a place is contained in the small circle next to the place. The absence of a circle indicates that the place has no tokens. The data values of the tokens are shown in the box with a dashed border positioned next to the circle. If desired, the user can hide a box, e.g., if the data values are irrelevant or too big to print. The transition Accept is displayed with a thick border to indicate that it is enabled in the current marking. Usually, enabled transitions and the markings of places are displayed using different colours. This significantly improves the readability compared to the black and white variant shown in Fig. 15.

Interactive simulations are supported in different variants. It is for instance possible to specify that only the token game of certain parts of the CPN model should be observed, and it is possible to control the amount of graphical feedback in each step. Interactive simulations do not require the model to be complete, i.e., the user can start investigating the behaviour of parts of a model and directly apply the insight gained to the ongoing design activities. Often, a model is gradually refined – from a crude description towards a more detailed one.

5.2 Automatic simulation

Later on in a modelling process, the focus shifts from the individual transitions to the overall behaviour of the full model. The *automatic mode* of the simulator is suitable here. In this case, the simulator makes random choices (by means of a random number generator) between enabled binding elements. In this way, it is possible to obtain much faster simulations. This is achieved even for

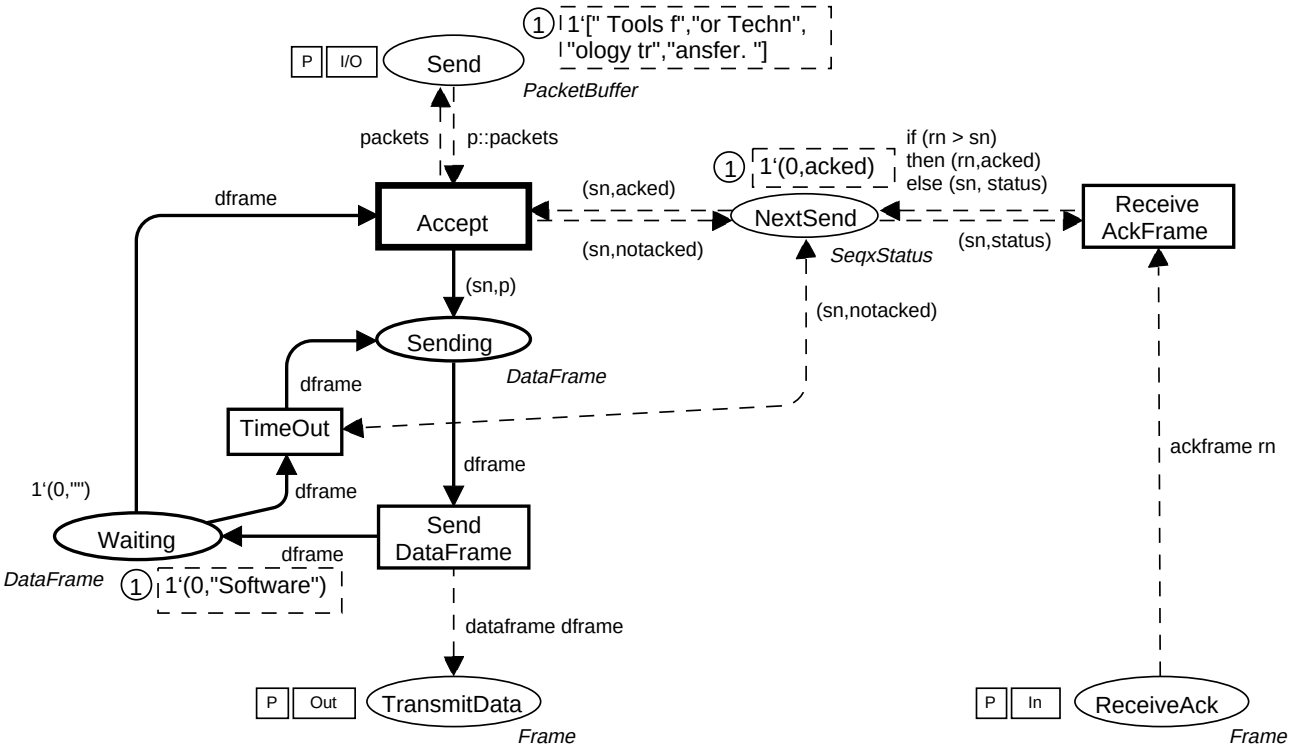

Fig. 15. Snapshot from an interactive simulation

large models, because the enabling and occurrence rules of Petri nets are local. This means that, when a transition has occurred, only the enabling of the nearest transitions needs to be recalculated, i.e., the number of steps per second is independent of the size of the model. A totally automatic simulation is executed with a speed of several thousand steps per second (depending on the nature of the CPN model and the power of the computer on which the CPN simulator runs). The user is in control of automatic simulations by means of *stop options*. Stop options make it possible, for instance, to give an upper limit to the number of steps the simulation should run.

Before and after an automatic simulation, the current marking and the enabled transitions are displayed as described for the interactive mode. However, the token game is not displayed during automatic simulations. Of course, this typically constitutes less information than desired. A straightforward possibility to obtain information about "what happened" is to use the *simulation report*. It is a textual file containing detailed information about all the bindings of transitions which occurred. Figure 16 shows a simulation report of the first 8 steps from an automatic simulation of the stop-and-wait protocol.

The A following the step number indicates that the binding of the transition was executed in automatic mode. The information after the @-sign specifies the page instance, i.e., the part of the CP-net to which the occurring transition belongs. Finally, for each step, the binding in which the transition occurs is shown. As an example, step 3 corresponds to an occurrence of the transition

1	A	Accept@(1:Sender#4) { dframe = (0,""),p = "Software", packets = " Tools f","or Techn","ology tr", "ansfer. "], sn = 0}
2	A	Send@(1:Sender#4) { dframe = (0,"Software")}
3	A	Transmit@(2:UniChannel#19) { frame = dataframe((0,"Software")), success = false}
4	A	TimeOut@(1:Sender#4) { dframe = (0,"Software"),sn = 0}
5	A	Send@(1:Sender#4) { dframe = (0,"Software")}
6	A	TimeOut@(1:Sender#4) { dframe = (0,"Software"),sn = 0}
7	A	Transmit@(2:UniChannel#19) { frame = dataframe((0,"Software")), success = false}
8	A	Send@(1:Sender#4) { dframe = (0,"Software")}

Fig. 16. Partial sample simulation report

Transmit on the second instance of page UniChannel in a binding corresponding to the first data frame being lost.

5.3 Code segments

It is possible to attach a piece of sequential CPN ML code to individual transitions using so-called *code segments*.

When a transition occurs, the corresponding code segment is executed. For example, it may read and write text files, update graphics or even calculate values to be bound to some of the variables of the transition. In this way, the code segments provide a very convenient interface between the CPN model and its environment, e.g., the file system. When we describe visualisation in Sect. 6 and performance analysis in Sect. 7, we will give a number of examples on the use of code segments.

5.4 Integration with the editor

Often, a simulation results in the desire to modify the model. Some of these modifications can be made immediately: it is possible to make minor changes while remaining in the simulator, e.g., to edit an arc expression. Other modifications require more involved rechecking/regeneration of the simulator code, which is only supported in the editor, e.g., it is impossible to add or change a type in the simulator. In Design/CPN, the editor and simulator are closely integrated. Therefore, it is easy and fast to go from the simulator back to the editor, fix a problem, re-enter the simulator, and resume simulation.

6 Visualisation of CPN models

The graphical feedback from interactive simulations and the simulation report from automatic simulations are in many situations a too detailed level of obtaining and interpreting results from simulations. To overcome this problem, the Design/CPN tool includes libraries for *visualisation*. The visualisation is driven by the CPN simulator and makes it possible to create and update various kinds of graphics during the simulation.

There are several reasons why visualising the behaviour of a system during lengthy simulations is beneficial. We can obtain a condensed overview of a lengthy simulation. By extracting the key activities and representing them in a graphical way the simulation results are easily interpreted. We can see whether the CPN model behaves as expected. If it does not, we can see where discrepancies appear. Then we can use interactive simulations or the simulation report to make a closer investigation of these situations. Furthermore, visualisation is able to hide the CP-nets and obtain graphical feedback from simulations which are specific for the application domain. This makes it possible to discuss design ideas and results from simulations with colleagues and other people who have knowledge about the application domain, but who are unfamiliar with CP-nets.

In this section we consider three kinds of graphical feedback from simulations: line and bar charts, message sequence charts, and application specific graphics. The libraries presented are all examples of standard libraries. It is possible for the user, due to the open architecture of Design/CPN, to tailor these libraries to specific domains, or to create new visualisation libraries on top of the CPN simulator.

The visualisation is handled by code segments associated with transitions as discussed in Sect. 5. Typically the user has to provide 2–5 lines of rather straightforward CPN ML code for each of the code segments. It is important to notice that the behaviour of the CPN model is not in any way affected by the additional graphical visualisation, and that new visualisations can be added independently of other means of feedback.

6.1 Line and bar charts

For the stop-and-wait protocol we may use the three charts shown in Figs. 17, 18, and 19. The chart in Fig. 17 is a *line chart* showing how fast the individual data packets are successfully received (as a function of the step number). From the line chart, we can see that the first data packet was received after approximately 10 steps, the second after approximately 30 steps, the third after approximately 50 steps, and so on. The line chart is updated each time a new packet is successfully received. This is done by a few lines of code in the code segment of ReceiveAckFrame.

The chart in Fig. 18 is a *bar chart*. It tells us how many times each of the data frames has been sent and with what result. From the bar chart we see that the data frame with sequence number 0 has been sent 6 times. One of these

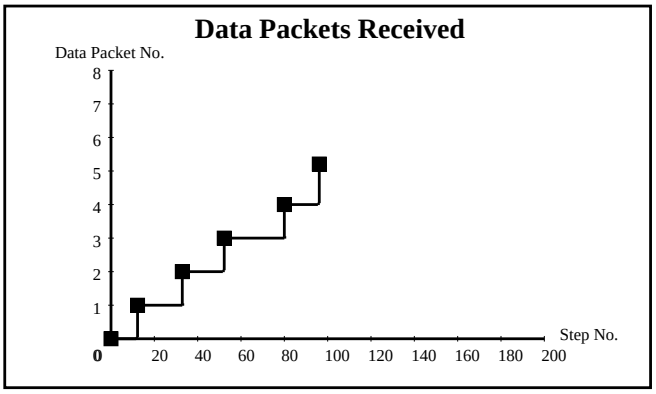

Fig. 17. Line chart for reception of data frames

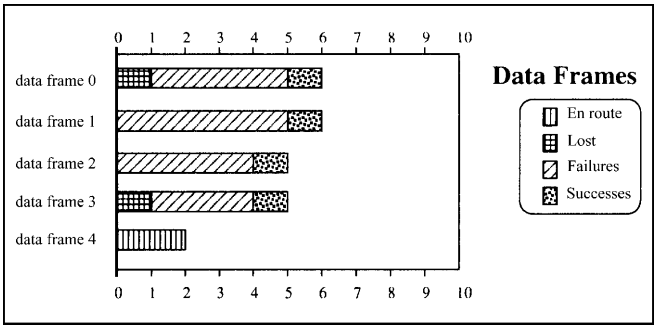

Fig. 18. Bar chart for transmission of data frames

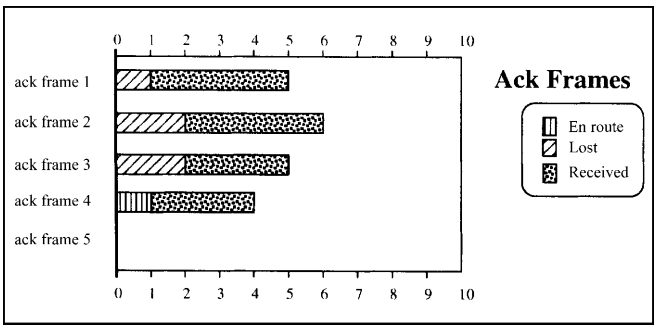

Fig. 19. Bar chart for transmission of acknowledgement frames

was lost, four were received as failures (i.e., out of order) and the last one was successfully received. Analogously, we can see that the data frame with sequence number 1 has been sent six times, while data frame number 2 and 3 have been sent five times each. Finally, we see that data frame number 4 has been sent twice, and that both of these are en route (i.e., on one of the places TransmitData and ReceiveData).

The chart in Fig. 19 is similar to the chart in Fig. 18, except that it shows the progress of acknowledgement frames. The two bar charts are updated periodically, with intervals specified by the modeller, e.g., for each fifty steps. The three charts give us a lot of valuable information about the behaviour of the protocol. As an example, it is straightforward to see that failures (i.e., overtaking) often cause more retransmissions than lost packets. It is also easy to see that we need more than 90 steps to transmit the five packets successfully, while with perfect communication channels (and no overtaking) it is possible to do this in 35 steps.

6.2 Message sequence charts

Message sequence charts (MSCs) [5] – also known as event traces, message flow diagrams, or time sequence diagrams – is a widely used formalism for specifying examples of either normal or exceptional executions of a system being designed. In the Design/CPN tool, MSCs are used to capture a single simulation run, thereby providing a graphical overview of the execution of the system. The exact information contained in the MSCs is determined by the user. For the stop-and-wait protocol an MSC may appear as shown in Fig. 20. Vertical lines in MSCs are typically used to represent processes or components of a system. Small squares on vertical lines and horizontal arrows between vertical lines are typically used to describe actions or events of a system.

An occurrence of the Accept transition is shown by a horizontal arrow between the first and second vertical line. The arrow is labelled with the data packet being accepted for transmission and the sequence number assigned to the data packet. Analogously, the occurrence of the SendDataFrame transition is shown by a horizontal arrow between the second and the third vertical lines. A successful occurrence of Transmit with a data frame is indicated by a horizontal arrow between the third and the fourth vertical lines. However, if the data frame is lost, we only get a small square at the third line. Similarly, an occurrence of the transition TimeOut also results in a small square but at the second line.

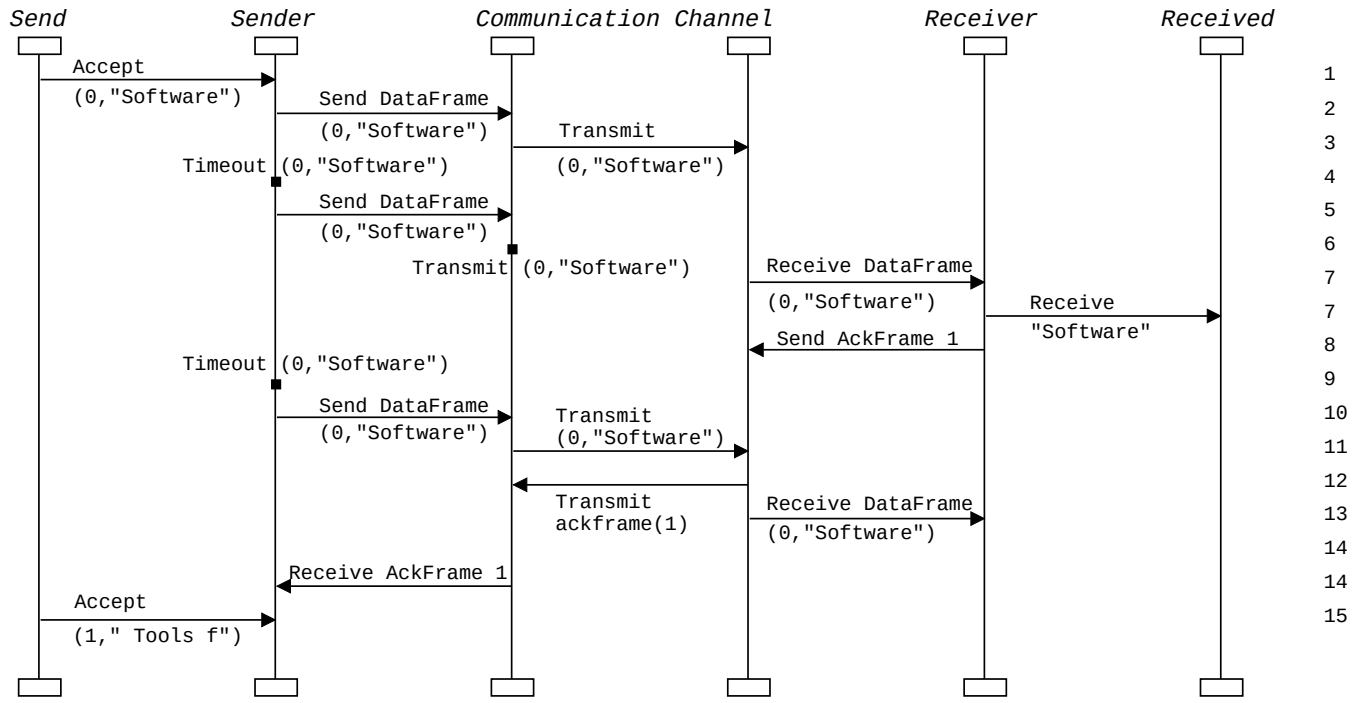

Fig. 20. MSC for stop-and-wait protocol

An occurrence of ReceiveDataFrame is indicated by a horizontal arrow from the fourth to the fifth vertical line. Furthermore, if the data frame being received was the one expected a horizontal arrow is created between the fifth and the sixth vertical line. An occurrence of the transition SendAckFrame is indicated by an arrow from the fifth to the fourth vertical line. The arrow is labelled with the sequence number corresponding to the acknowledgement frame. Occurrences of Transmit with acknowledgement frames are indicated in a similar way as occurrences of Transmit with data frames (but the arrows are now drawn from right to left, while the square dots are positioned at the fourth vertical line). Occurrences of ReceiveAckFrame are indicated by an arrow from the third to the second vertical line with a label corresponding to the sequence number of the acknowledgement. The numbers to the outermost right of the MSC correspond to step numbers of the simulation. In Fig. 20 we have an arrow (or a square) for each step in the simulation. This means that the MSC contains all the information in the simulation report. However, it is much more common only to record a few key activities, e.g., the transmission of data frames and acknowledgement frames.

For creating the MSCs, code segments have been attached to the transitions of the CPN model. Figure 21 shows the code segment of the transition Accept. MSCGraphics is a reference to the message sequence chart, while mkst_col'DataFrame is a predeclared function providing a string representation of the CPN ML value (sn,p). The function MSC.Message is one of the primitives provided by the MSC library for creating a horizontal arrow between two vertical lines in the MSC according to the record given as argument. The entries sender and receiver are used to specify the source and the destination of the horizontal arrow to be created. In this case the arrow is drawn from the first vertical line (identified by "Send") to the second vertical line (identified by "Sender"). The arrow is labelled by the transition name (Accept) and the occuring binding element. The code segments of the other transitions are similar.

```
input (sn,p) ;
action
 MSC.Message (!MSCGraphics)
    {annotation = makestring (step ()),
     label = "Accept"^NEWLINE^
             (mkst_col´DataFrame (sn,p))^NEWLINE,
     sender = "Send",
     receiver = "Sender"
     } ;
```

Fig. 21. Code segment for the transition Accept

6.3 Application specific graphics

The line, bar, and message sequence charts presented in the previous sections are characterised by being rather standard ways of creating graphical feedback. It is however often advantageous to create graphical feedback from simulations which are based on concepts and objects from the specific application domain. The Mimic/CPN library [58] supports such *application specific graphics*. Moreover, the library makes it possible to interact with the graphical feedback and, in this way, to have a high-level way to control and provide input for the simulation. The stop-and-wait-protocol is really too simple to illustrate the power of application specific graphics. Instead we will illustrate how CP-nets and application specific graphics were applied in a project at Deutsche Telekom [4] to the design of services in Intelligent Networks.

Intelligent networks (IN) extends conventional telecommunication networks with the ability to collect and compute information in addition to the conventional transmission and switching functions. The ability to collect and compute information is referred to as network intelligence. Example of IN services are freephone, card calling services, and premium rate services.

Figure 22 illustrates the main components of an IN architecture. The basic idea is to separate the classical switching and transport functionality from the IN service logic functionality. The *service switching point (SSP)* represents physical local telephone exchanges whereas the *service control points (SCP)* represent network nodes containing the IN service logic. From these nodes the local exchanges (SSPs) can be controlled remotely using a so-called signalling system network (SS.7). A local exchange contains the *call control function (CCF)* which represents the functionality contained in a local exchange that is not yet prepared for handling IN calls. The *service switching function (SSF)* enhances the local exchange with the capability of handling IN calls, and makes it possible to communicate with the IN service logic contained in the SCP. The SCP consists of two parts: the *service control function (SCF)* which contains the IN service logic and the *service data function (SDF)* containing the database for the IN.

The goal of separating the IN service logic from the basic switching and transport functionality is to make it possible to introduce new IN services without having to modify the local exchanges (SSPs). The aim of the project

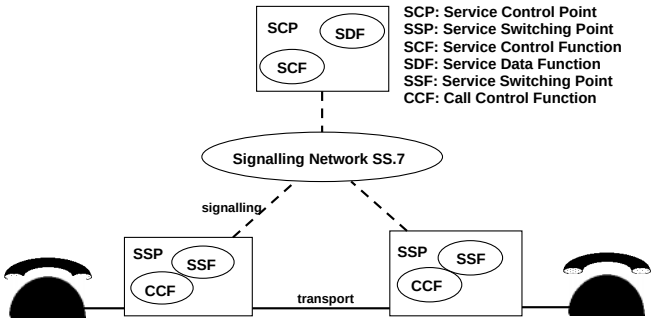

Fig. 22. Intelligent network architecture

at Deutsche Telekom was to investigate how the basic call processing in the SSPs should be augmented to allow this. To investigate this, a CPN model of the IN architecture was constructed. The CPN model consisted of about 40 pages. The CPN model contained a representation of the users of the network, the IN service logic, and the basic call processing at the SSPs.

To provide input and control the simulations, the CPN model was augmented with the application specific graphics shown in Fig. 23. The application specific graphics represent a small IN consisting of one service control point (SCP) connected to two service switching points (SSPs), each of which has three phones attached to it. Having started a simulation, the user can select graphical objects. Depending on the state of the system, different kinds of interaction are possible. If a phone is on-hook, it is possible to select the phone which will result in changing its state to off-hook (e.g., the phone to the upper left). In this state the user can select the phone or its receiver. Selecting the receiver corresponds to an on-hook operation. Selecting the phone will prompt the user for the digits corresponding to the phone number. Figure 23 shows the graphics in a state where two phones 3070 and 7890 are active and have established a connection. In this case the user can choose either to hang-up one of the active phones, or to lift the receiver of one of the passive phones. As can be seen this visualisation is used for controlling the simulation and for testing different scenarios, and it allows users who are unfamiliar with the CP-net to understand the behaviour of the system.

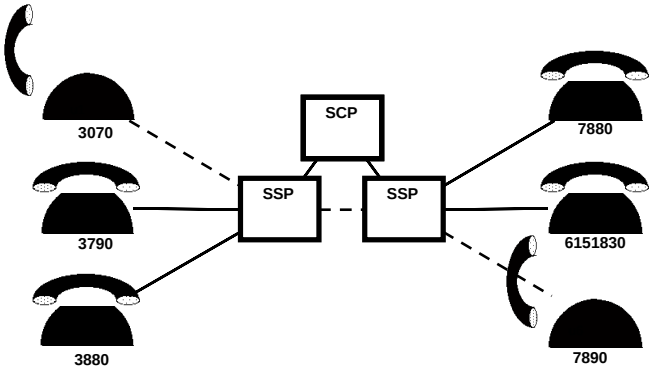

Fig. 23. Example of application specific graphics

The application specific graphics also have a record and replay facility. This makes it possible to record the input provided by the user and later replay these inputs, e.g., after having made modifications to the underlying CPN model. In this way it is possible to create a set of standard scenarios (use cases) for testing the system under consideration.

7 Performance analysis

This section introduces the time concept of CP-nets and explains how this can be used to conduct simulation-based performance analysis of systems. As we proceed with the introduction of the time concept and how it is supported by the Design/CPN tool, we construct a timed CPN model of the stop-and-wait protocol. Also, we analyse the performance of the stop-and-wait protocol in terms of throughput, delay, and usage of channel bandwidth and buffer space.

7.1 Timed CP-nets

The time concept of CP-nets is based on the introduction of a *global clock*. The clock values represent *model time*, and they may either be integers (i.e., discrete) or reals (i.e., continuous). In addition to the token value, we allow each token to carry a *time value*, also called a *time stamp*. Intuitively, the time stamp describes the earliest model time at which the token can be used, i.e., removed by the occurrence of a binding element.

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

Figure 25 shows a timed CP-net for the sender of the stop-and-wait protocol. The net structure is the same as in the untimed CP-net in Fig. 2. However, the net inscriptions and the type declarations (shown in Fig. 24) have been modified with the addition of some timing constructs which will be explained in detail below.

```
(* —— data packets —— *)
color String = string ;
color ArrivalTime = TIME ;

color Packet = product ArrivalTime * String timed ;
color PacketBuffer = list Packet ;

(* —— status and sequence numbers —— *)
color Seq = int timed ;
color Status = with acked | notacked ;

color SeqxStatus = product Seq * Status ;

(* —— data and acknowledgement frames —— *)
color DataFrame = product Seq * Packet ;
color AckFrame = Seq ;

color Frame = union
      dataframe : DataFrame + ackframe  : AckFrame ;
```

Fig. 24. Timed type definitions

From the type definitions it can be seen that type Packet has been modified to be a product of String and ArrivalTime. The latter is used to calculate the delay of the

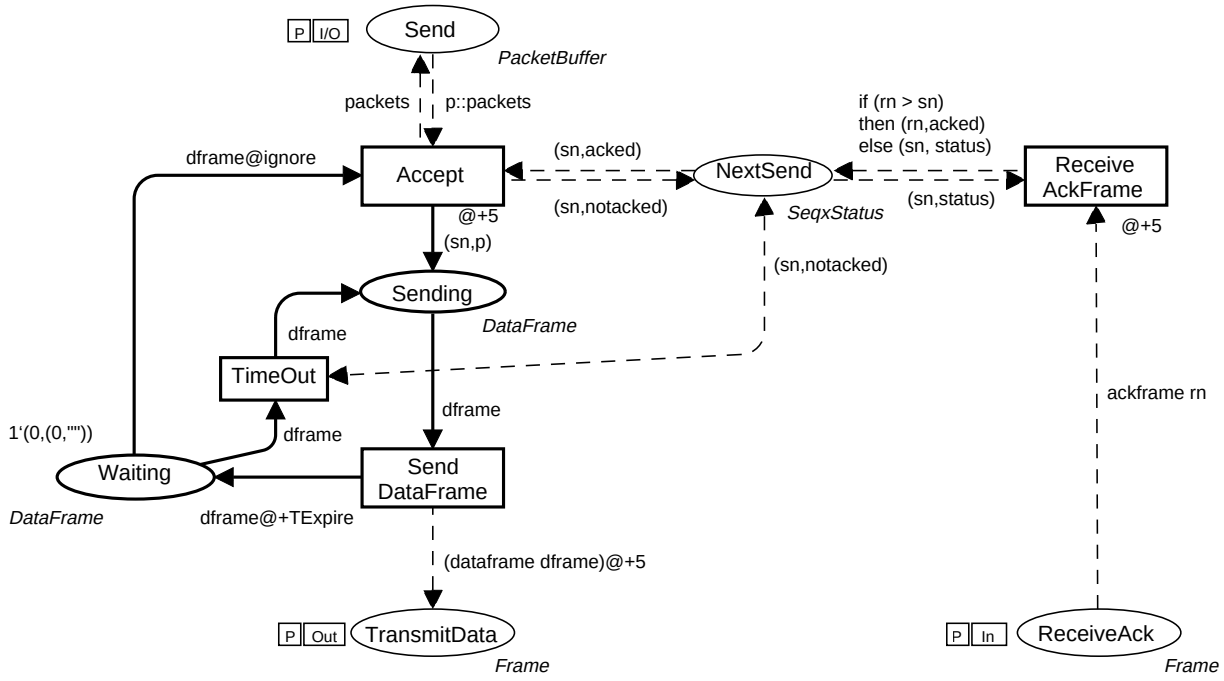

Fig. 25. Timed sender of the stop-and-wait protocol

data packets. Moreover, the types Packet and Seq are now *timed*. By convention, the five structured types PacketBuffer, SeqxStatus, DataFrame, AckFrame, and Frame also become timed – because they contain a component (Packet or Seq) which is timed. This means that the corresponding tokens carry time stamps.

The marking of places with a timed type is a *timed multi-set* of token values. A timed multi-set is similar to a multi-set, except that each element in the multi-set carries a time stamp. As an example, a possible marking of the place TransmitData is the following:

```
2'dataframe(0,(0,"CP-nets"))@[19,45] +
4'dataframe(1,(0,"CPN"))@[7,10,42,42]
```

This marking contains two tokens with value dataframe(0,(0,"CP-nets")) and time stamps 19 and 45. Moreover, there are four tokens with value dataframe(1,(0,"CPN")); one with time stamp 7, one with the time stamp 10, and two with time stamp 42. Similar to ordinary untimed multi-sets, timed multi-sets are written as a sum (+) using the symbol prime (') (pronounced "of") to denote the number of appearances of an element. The @ sign (which is usually read as "at") and the list symbol [ ] is used to specify the time stamps. For the stop-and-wait protocol we use an integer clock, starting at zero. Hence, all time stamps in the initial marking are equal to zero.

To model that an activity/operation takes r time units, we let the corresponding transition create time stamps for its output tokens that are r time units larger than the clock value at which the transition occurs. This implies that the tokens produced are unavailable for r time units.

It can be argued that it would be more natural to delay the creation of the output tokens, so that they did not come into existence until r time units after the occurrence of the transition had begun. However, such an approach would mean that a timed CP-net would get "intermediate" markings which do not correspond to markings in the corresponding untimed CP-net, because there would be markings in which input tokens have been removed but output tokens not yet generated. Hence we would get a more complex relationship between the behaviour of timed and untimed CP-nets. For a timed CP-net we require that each step consists of binding elements which are both colour enabled and ready. As a consequence, the possible occurrence sequences of a timed CP-net always form a subset of the possible occurrence sequences of the corresponding untimed CP-net. This means that we have a well-defined and easy-to-understand relationship between the behaviour of a timed CP-net and the behaviour of the corresponding untimed CP-net.

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

Now let us take a closer look at the transitions in the sender. The transition Accept has a time inscription @+5. Intuitively, the time inscription describes how much time the corresponding operation takes. This time inscription

specifies that the tokens created at Sending, Send, and NextSend when Accept occurs, get time stamps that are 5 time units larger than the time at which the transition occurs. As an example, if the transition Accept occurs in the initial marking at time 0 we get the marking indicated in Fig. 26. In this marking the transition SendDataFrame is not enabled at time 0 since the time stamp of the token on the place Sending is 5 – the transition is colour enabled, but in order for the transition to become ready, the model time has to be increased to 5. This will be the next model time at which a transition becomes enabled. Hence the system is in its initial marking (or state) for zero time units, and in the marking resulting from the occurrence of Accept in the initial marking in five time units.

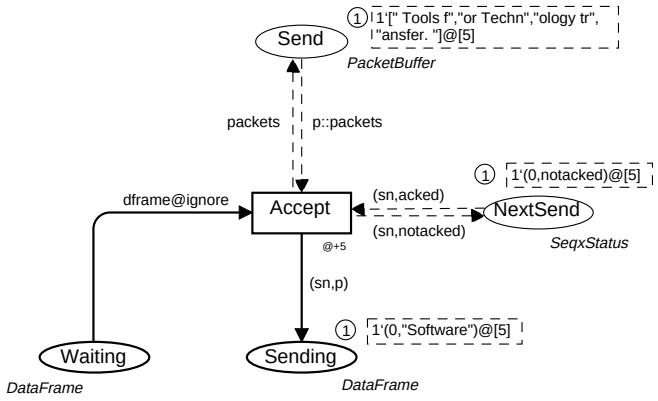

Fig. 26. Marking resulting from the occurrence of Accept

Let us now consider the remaining time inscriptions. The transition ReceiveAckFrame has the same time inscription as the transition Accept. For the transition SendDataFrame the time inscription is not put directly on the transition but associated with the arc expressions of the outgoing arcs. The arc expression on the arc to TransmitData has the time inscription @+5 which specifies that the tokens put on this place are given a time stamp which is 5 time units larger than the current model time. The arc expression on the arc to Waiting is @+TExpire. TExpire is a constant used to specify the time which has to elapse before a data frame is retransmitted. By assigning the token put on the place Waiting, a time stamp which is TExpire time units higher than the current model time, a retransmission of a data frame cannot happen until TExpire time units later. The time inscription on the arc from Waiting to Accept is @ignore. This means that the enabling rule of Accept ignores the time stamp of tokens on Waiting. Hence an occurrence of the transition Accept can remove a token from the place Waiting even though its time stamp is higher than the current model time. This allows us to disable the retransmission timer immediately when an acknowledgement frame arrives for the data frame which we are currently sending.

Let us now consider how the communication channel is modelled in the timed CPN model. In the untimed CPN model of the communication channel the unreliability of the communication channel was modelled such that there were two possible outcomes of a transmission; either the frame was lost or successfully transmitted and there was a 50% chance for each. We now extend the model in such a way that the two probabilities need not be equal and we take into account the transmission time for a frame. This is achieved by the modified CPN model of the unidirectional communication channel shown in Fig. 27 and the declarations shown in Fig. 28.

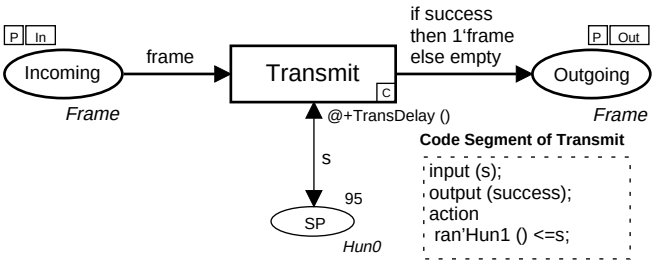

Fig. 27. Timed unidirectional channel

The place SP (success probability) has been added. The marking of this place is used to specify the probability of successful transmission. We use a code segment (see Sect. 5.3) to calculate the binding of the variable success. The types Hun0 and Hun1 contain the integers in the intervals 0..100 and 1..100, respectively. ran'Hun1 draws an element uniformly at random from Hun1. If this integer is smaller than the value bound to s then success will be bound to true. In the above case the initial marking of SP is 95 specifying that the probability of successful transmission is 95%.

The transmission time on the channel is specified by the time inscription @+TransDelay (). TransDelay is a user defined function returning a random element from a normal distribution with mean 10 and deviation 2. This implies that the duration of a transmission varies according to this random distribution. The CPN simulator offers a number of standard statistical functions for specifying other types of delays (e.g., uniform, exponential, and poisson).

The receiver of the timed CPN model is identical to the receiver of the untimed CPN model except that a delay of five time units has been assigned to the transitions ReceiveDataFrame and SendAckFrame.

```
color Bool = bool ;
var success : Bool ;

color Hun0 = int with 0..100 ;
var s : Hun0 ;

color Hun1 = int with 1..100 declare ran ;

fun TransDelay () = normaldist (10.0,2.0) ;
```

Fig. 28. Declarations for communication channel

In general, the time delays may be made much more complex than described above, and they may depend upon the binding in question, i.e., upon the values of the input and output tokens.

7.2 Workload and traffic

When analysing the performance of a system some mechanism is needed to generate the workload or traffic to be used as input for evaluating the performance of the system. In the case of the stop-and-wait protocol, the workload is the data packets presented to the protocol for transmission.

There are a number of techniques which can be applied when generating the workload. One technique is to make a CP-net which is part of the CPN model, and which is responsible for generating the workload (perhaps according to some random distribution). In other cases the workload may be available from a text file (e.g., a server access log file) which can then be read into the CPN model. A third technique is a combination of the two in which a separate CPN model is created which generates the workload (once and for all) and writes it into a text file. This text file can then be used as input for the actual CPN model. Specifying the workload in a file which is read into the model is also sometimes referred to as trace-driven simulation. It is beneficial if reproducibility is needed, for instance, when comparing the performance of different system designs.

For the stop-and-wait protocol we add the relatively simple CP-net shown in Fig. 29 to the CPN model. The place Send is related via port assignments to the place Send on the page modelling the sender. Hence it corresponds to the data packet buffer of the sender. The place Next has the integer type Int, and is used as a counter to enumerate the data packets. The constant NextPacket is used to specify the interval between the arrival of data packets. When the transition GeneratePacket occurs, a new data packet is put at the end of the buffer. The contents of this data packet is a pair, where the first element specifies the arrival time of the packet, while the second element specifies the packet number. The arrival time is the time at which the packet is generated, i.e., the time at which GeneratePacket occurs. It is determined via the function time which returns the current model time. The packet number is determined by the data value of the token on place Next. When GeneratePacket occurs, the value of the token on the place Next is incremented, and given a time stamp which is NextPacket higher than the current model time. This ensures that the next data packet will arrive NextPacket time units later. The arrival time recorded in the data packet will be used later to calculate the delay of the data packet. We use the ignore time inscription on the arc from Send such that the data packets are put in the buffer as soon as they arrive. Hence, for the stop-and-wait protocol we have a simple workload in which data packets arrive for transmission with some fixed input rate.

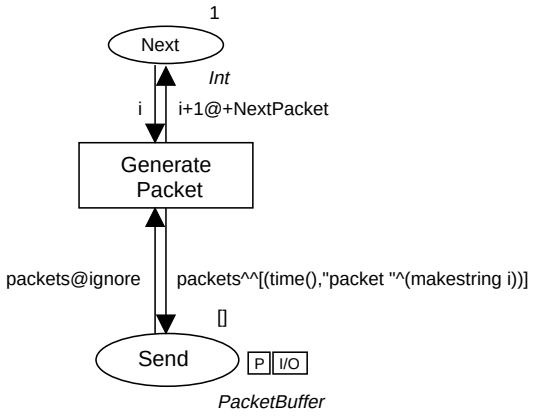

Fig. 29. Workload generator

7.3 Data collection and processing

Simulation based performance analysis typically consists of making a number of lengthy simulations. To make it simple to collect data during such simulations, the CPN simulator implements a number of high-level primitives and data structures [66] for data collection. The collected data are stored in data structures called *data collections*. Data collections offer, in addition to serving as a storage for the collected data, operations for computing, e.g., the average, standard deviation, and minimum and maximum of the observed values. All such key figures can be written to a *performance report*, which provides a quick summary of the observed values and hence the performance of the system. Moreover, the observed values can be written to an *observation log file* which can serve as a basis for processing the collected data further, e.g., in a spreadsheet or for visualisation in a graphical tool.

To collect data only minor modifications are needed to the CPN model. For the stop-and-wait protocol we only had to modify the Packet type by adding an arrival time to be used later to compute the delay of the data packet.

The data to be collected depend on the performance measures of interest. For the stop-and-wait protocol we are first and foremost interested in the throughput and the delay. For analysing the throughput we consider the *interarrival time*, which is the time which elapses between successive deliveries of data packets in the Received buffer. The delay is the time from when a data packet is put in the Send buffer at the sending side until it is delivered in the Received buffer at the receiver side. Therefore, whenever the transition ReceiveDataFrame occurs with the data packet that we expect next, i.e., a data packet is written in the Received buffer, we collect (observe) the time which has elapsed since the delivery of the last data packet, and the time which has elapsed since the data packet was put in the Send buffer.

Figure 30 depicts the transmission delays for data packets in a simulation corresponding to 50,000 time units, with a retransmission interval of 70 time units, and with a workload corresponding to data packets arriving

with an interval of 80 time units. Figure 31 depicts the interarrival times of the packets. The figures have been created by reading the observation log files into the Gnuplot tool [52].

Table 1 shows the part of the performance report containing the key numbers for the observed values for throughput and delay. There have been 625 observations corresponding to the arrival of 625 data packets. The average interarrival time (I-Time) was 79.91 time units, and the average delay was 28.58 time units. *StD* denotes the standard deviation, i.e., the average deviation from the average value. It can also be observed that the minimal and maximal interarrival time of data packets have been 19 and 249 time units, respectively. Similarly, the minimal and maximal delay experienced have been 14 and 189 time units, respectively.

For the stop-and-wait protocol we are also interested in getting information about its usage of buffer space. We want to avoid too many frames accumulating in the frame buffers at the sender and the receiver side. Also, we want to avoid a backlog of unsent data packets accumulating in the Send buffer. The latter could for instance happen if the arrival rate of data packets exceeds the capacity of the protocol. In order to analyse the buffer usage we observe at each step in the simulation the number of tokens on the TransmitData, ReceiveAck, TransmitAck, and ReceiveData places, and the length of the list on the place Send. The key figures for the observed values of our simulation with 50,000 time units are shown in Table 2. The Send row corresponds to the observation on the length of the list on the place Send. The ReceiveData row corresponds to the observation on the number of tokens on the place ReceiveData in the receiver. The TransmitAck corresponds to the observation on the number of tokens on the place TransmitAck in the receiver. Similarly, the ReceiveAck and TransmitData rows correspond to the accordingly named places in the sender.

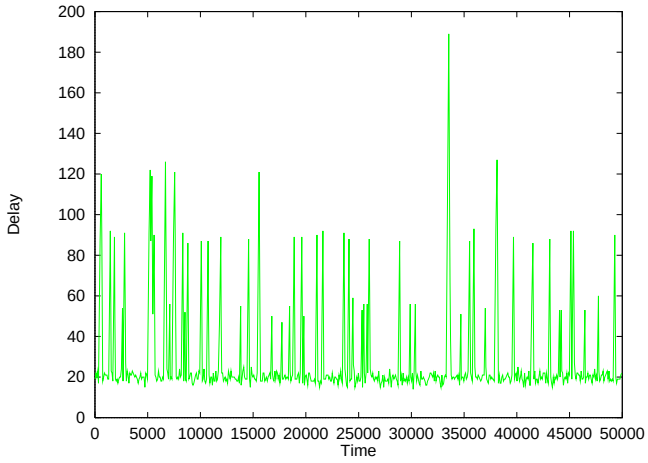

Fig. 30. Packet transmission delay

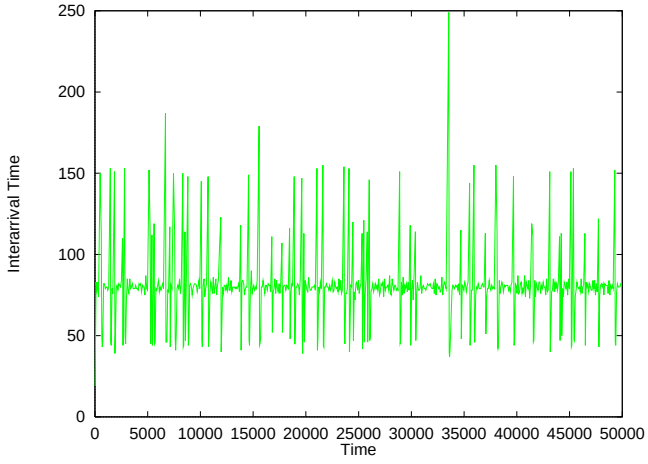

Fig. 31. Interarrival time (I-Time)

Table 1. Transmission and interarrival time

	Updates	Average	StD	Min.	Max.
I-Time	625	79.91	23.35	19	249
Delay	625	28.58	22.77	14	189

Table 2. Analysis of buffer contents

Place	Updates	Average	StD	Min.	Max.
Send	5276	0.07	0.27	0	3
ReceiveData	5276	0.12	0.33	0	1
TransmitAck	5276	0.07	0.25	0	1
ReceiveAck	5276	0.12	0.32	0	1
TransmitData	5276	0.07	0.25	0	1

For each of the data collections there have been 5276 updates corresponding to the 5276 steps of the simulation. From the table it can be seen that the use of buffer space is modest. There has been at most one token on each of the places ReceiveData, TransmitAck, ReceiveAck, TransmitData, and a backlog of at most three unsent data packets. Hence, a workload in which data packets arrive with an interval of 80 time units seems to be well within the range of the capacity of the protocol.

The buffer space analysed above is one measure of the computational resources demanded by the communication protocol. Another measure is the use of bandwidth on the communication channel. The use of bandwidth can be measured by considering the number of occurrences of the Transmit transition on the two instances of page UniChannel. These figures are shown in Table 3. There have been 1342 updates corresponding to 1342 occur-

Table 3. Usage of network bandwidth

Transition	Updates	Time	Frequency
Transmit	1342	50000	0.027

rences in the simulation. This yields a frequency of 0.027, i.e., an average of 37 time units between transmissions.
The purpose of performance analysis is very often to investigate *what-if-scenarios*. To illustrate this we will investigate how the retransmission interval, as specified by the constant TExpire in Fig. 25, influences the performance of the stop-and-wait protocol. The performance of the protocol for six different values of the retransmission interval (TExpire) is listed in Table 4. The I-Time row lists the average interarrival time, the Delay row lists the average delay, and Bandwidth lists the use of bandwidth i.e., the average number of occurrences of the Transmit transition per time unit. RecvAck and ReceiveData are the average number of tokens on the place ReceiveAck in the sender and ReceiveData in the receiver, respectively. From the results it can be seen that it does not pay to decrease the retransmission interval below approximately 30 time units. From that point on, we do not gain any reduction in the delay, but start to make increasing use of the bandwidth and buffer space.

Table 4. Investigation of retransmission parameter

TExpire	70	50	30	20	10	5
I-Time	79.91	79.90	79.91	79.91	79.91	79.84
Delay	28.58	24.74	21.41	20.64	20.19	21.48
Bandwidth	0.03	0.03	0.08	0.13	0.24	0.45
RecvAck	0.12	0.12	0.22	0.23	0.42	0.52
RecvData	0.12	0.12	0.23	0.24	0.50	1.66

8 State space analysis

As explained in Sect. 5 it is customary to first debug and investigate a system by means of simulations. Simulation works in a similar way to program testing and, unless the system is trivial, cannot be used to prove or verify properties about the behaviour of the system. Therefore simulation is often complemented by *state space analysis*.

The basic idea behind a *state space* is to construct a directed graph which has a node for each reachable marking and an arc for each occurring binding element. State spaces are also called *occurrence graphs* or *reachability graphs/trees*. The first of these names reflects the fact that a state space contains all the possible occurrence sequences of the CP-net, while the two latter names reflect that the state space contains all reachable markings of the CP-net. State spaces of CP-nets can be constructed fully automatically, and from a constructed state space it is possible to answer a large set of analysis and verification questions concerning the behaviour of the system. Moreover, if the system does not have a desired property then the state space method can provide debug information in the form of counter examples. The counter examples are typically a node, path, or a subset of the state space demonstrating that the desired property does not hold.

The expressive power of CP-nets inherited from Place/Transition nets [60] and the SML language implies that essentially all interesting verification questions concerning CP-nets are undecidable (see, e.g., [19]). However, in practice, many CP-nets do have a finite state space making verification possible. In addition to this, it is often possible to prove and disprove properties of a system by relying on partial state spaces, i.e., finite subgraphs of the full state space.

The focus and emphasis of this section will be more on state space analysis as seen from the point of view of the user of the CPN state space tool, and less on algorithmic and implementation issues. The reader interested in the underlying algorithms and formal definitions is referred to [11, 33, 35]. As we proceed in this section, we verify the correctness of the stop-and-wait protocol.

8.1 State spaces of CP-nets

Very often, the first step in state space analysis is to make some modifications to the CPN model in order to make it tractable for state space analysis. Tractable typically means obtaining a CPN model with a finite state space of a size which is within the limits of the available computing resources.

For the stop-and-wait protocol we modify the modelling of the frame buffers between the sender, receiver, and the communication channel. In the original CPN model (see Fig. 2) an arbitrary number of frame tokens could be on the places TransmitData and ReceiveAck. This corresponds to the frame buffers being unbounded. To obtain a finite state space we modify the modelling of the frame buffers in such a way that they have a finite capacity (of one). Figure 33 shows the modified CPN model of the sender of the stop-and-wait protocol, and Fig. 32 shows the modified declaration of the Frame type.

```
color Frame = union
      dataframe : DataFrame +
      ackframe  : AckFrame +
      noframe ;
```

Fig. 32. Modified Frame type definition

The modification is that the sender can only put a data frame in the frame buffer TransmitData when it is empty corresponding to a token with value noframe being located at the place. Similarly, when the sender receives an acknowledgement frame, a token with value noframe is returned to the place ReceiveAck indicating that the frame buffer is now empty. This implies that it is no longer possible for frames to overtake each other. The receiver and the communication channel have been modified in a similar fashion as the sender.

The initial part of the state space for the stop-and-wait protocol is shown in Fig. 34. The rounded boxes with thick borderlines are the nodes of the state space. Each

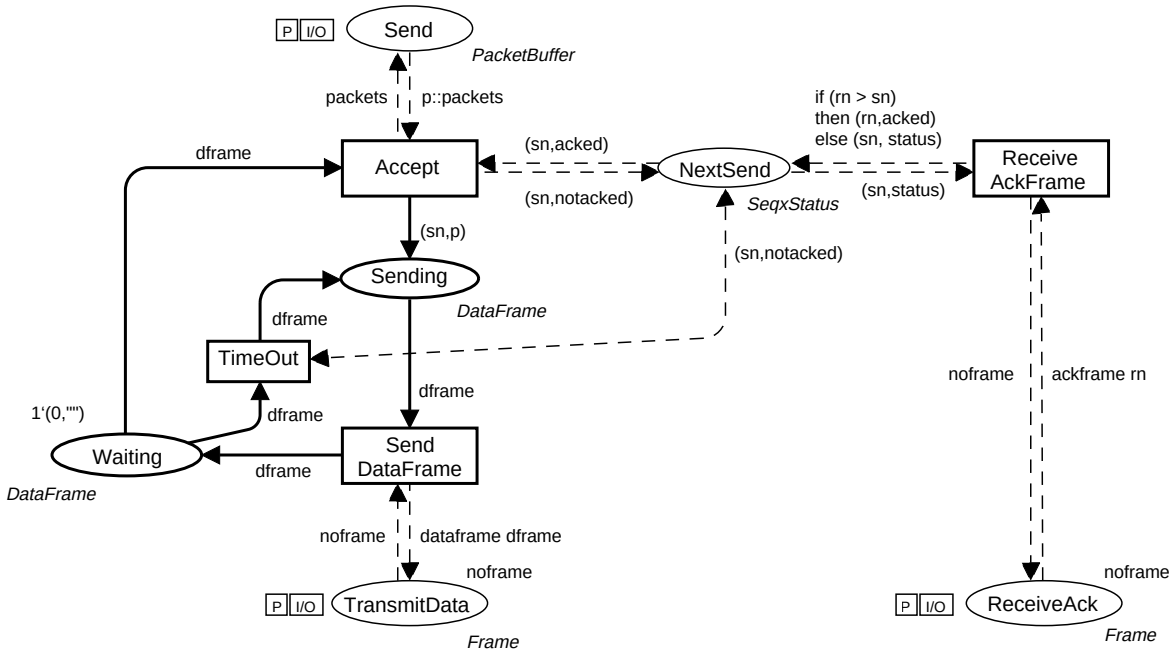

Fig. 33. Modified sender of the stop-and-wait protocol

of them represents a reachable marking. The marking of the individual places is described in the dashed box next to the node. Below, we focus on the marking of places in the sender, and therefore the box lists only the marking of places in the sender. By convention we omit places with an empty marking, and we have also chosen to omit the place Send.

At the top of Fig. 34, we have a node with a thicker borderline. This node represents the initial marking. The text inside the node tells us that this is node number 1 and that it has zero predecessors and one successor (the latter information may be useful when we have drawn only part of a state space). Analogously, we see that node number 2 has three predecessors and one successor. By convention we use M_n to denote the marking of node number n. Each arc represents the occurrence of the binding element listed in the solid box next to the arc. In M_1 the only enabled binding element is transition Accept with the binding listed in the box. When this binding element occurs, we reach marking M_2, in which there is again one enabled binding element. An occurrence of SendDataFrame with the variable dframe bound to (0,"Software") will lead to the marking M_3. In marking M_3 there are three enabled binding elements corresponding to the two enabled bindings of transition Transmit (data frame loss or successful transmission) and the one enabled binding of transition TimeOut. Depending on which of these binding elements occur we end up in marking M_4, M_5, or M_6. Node 6 has a quote (") following the node number. This indicates that successors for this node have not yet been calculated.

From the first part of the state space it can be observed that an occurrence of transition Transmit with the variable success bound to false in M_3 followed by the occurrence of transition TimeOut leads via the marking M_5 to the marking M_2. An occurrence of the two binding elements in the opposite order also leads from M_3 to M_2, but now with M_4 as the intermediate marking. The enabled binding of TimeOut is concurrently enabled with each of the two bindings of transition Transmit. By definition, arcs corresponding to steps with more than one binding element are not part of the state space. Such arcs would give information about the concurrency between binding elements, but they are not necessary for the verification of standard behavioural properties.

8.2 Generation of state spaces

The CPN state space tool supports two modes in which the user can generate state spaces: *interactive* and *automatic*. The relationship between these two modes is similar to the relationship between interactive and automatic simulations.

Interactive generation. In interactive generation the user specifies a marking, and the tool then calculates the enabled binding elements in this marking, and the successor marking resulting from the occurrence of each of these binding elements. The first part of the state space for the stop-and-wait protocol in Fig. 34 was obtained in this manner. Interactive generation is typically used in connection with drawing parts of the state space. We will return to this topic in Sect. 8.5. Interactive generation of the state space has many similarities with single step debugging.

Automatic generation. State spaces are in most cases of a size which makes it impractical to generate the full state

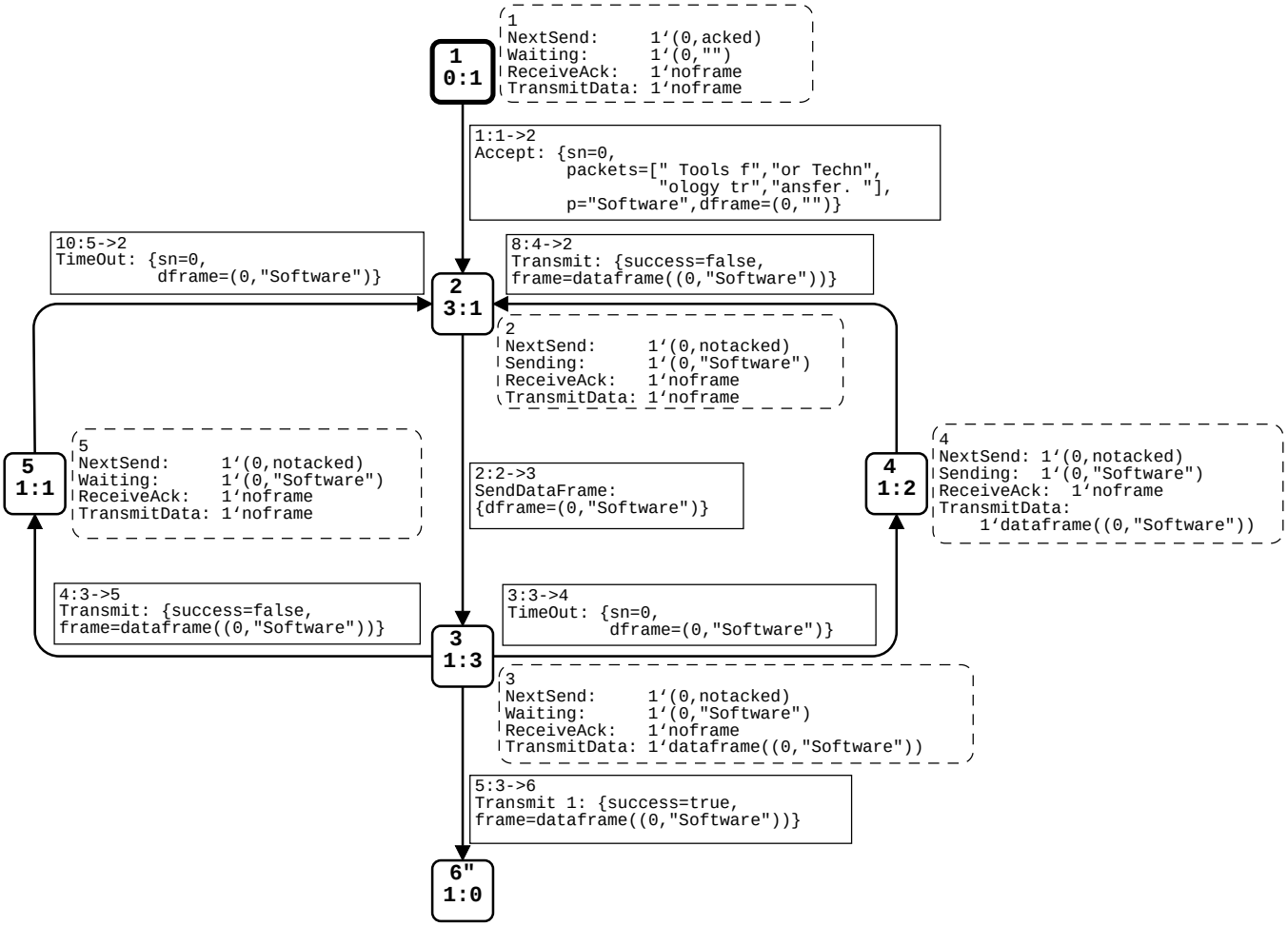

Fig. 34. Initial part of the state space for the stop-and-wait protocol

space in an interactive manner. Therefore it is also possible to generate the state space in an automatic manner. Here, the user simply starts the generation and the state space is then fully automatically calculated by the state space tool without any intervention from the user. The user can control an automatic generation using so-called *stop* and *branching options*. Stop options allow the user to specify conditions under which the calculation of the state space should be stopped. An example of this is to specify an upper bound on the number of nodes that should be generated. The branching options give the user local control over the generation of successors, making it possible to specify that only a certain subset of the successors of a node should be generated. The generation control is particularly important in early phases of state space analysis which is typically concerned with identifying a configuration of the system which is tractable for state space analysis. The stop and branching options also make it possible to obtain a *partial state space*. A partial state space is a subset of the full state space. In many situations it is not necessary to generate the full state space in order to locate errors and check the correctness of a system. A detailed discussion of partial state spaces is beyond the scope of this paper, and we will consider only full state spaces in this section.

The state space of the stop-and-wait protocol has 1220 nodes, 3621 arcs and is generated in 4 seconds on a Sun Ultra Sparc with 300 Mb of memory. This is a rather small state space and typically the state space consists of hundreds of thousands of nodes and arcs. The CPN state space tool has been used to construct state spaces with up to half a million nodes and one million arcs (when the tool runs on a machine with three hundred Megabytes of memory).

8.3 The state space report

When a state space has been generated, the next thing to typically do is to study the *state space report*. This is a text file containing answers to a set of standard behavioural properties which can be used on any CPN model (i.e., generic properties that can be formulated independently of the system under consideration). The state space report can be produced in a few minutes – totally automatically. It contains a lot of highly useful information about the behaviour of the CPN model. By

studying the state space report the user gets a first rough idea, as to whether the CPN model works as expected. If the system contains errors they are often reflected in the state space report. In the following we will go through the various parts of the state space report for the stop-and-wait protocol. We only give an intuitive explanation of the behavioural properties. The reader interested in the formal definitions is referred to [32, 35].

Statistical information. The first part of the state space report contains some statistical information about the size of the state space. The statistical information for the state space of the stop-and-wait protocol is listed in Table 5. We see that the state space has 1220 nodes and 3621 arcs. We have calculated the full state space, and this took only 4 seconds (on a Sun Ultra Sparc with 300 Mb of memory).

Table 5. Statistical information

State Space		SCC Graph	
Nodes:	1220	Nodes:	742
Arcs:	3621	Arcs:	2622
Secs:	4	Secs:	1
Status:	Full		

The statistical part also contains information about the *SCC-graph* [35]. A *strongly connected component (SCC)* is a maximal subgraph in which it is possible to find a path between any pair of nodes, i.e., a part of the state space in which all nodes are mutually reachable from each other. Each node in the state space belongs to exactly one SCC and hence we can use the SCCs to obtain a more compact graph known as the SCC-graph. The SCC-graph has a node for each SCC and an arc for each binding element which leads from one SCC to another. Strongly connected components can be calculated with Tarjan's algorithm [23] which has a time and space complexity that is linear in the size of the state space. Hence, it is cheap to construct SCC-graphs which turn out to be very efficient for the investigation of certain kinds of behavioural properties [34, 35]. From Table 5 we see that the SCC-graph has 742 nodes and 2622 arcs. This means that there are less SCCs (742) than state space nodes (1220), and hence at least one of the SCCs covers more than one state space node. From this we conclude that an infinite occurrence sequence exists, and hence we cannot be sure that the protocol terminates. To guarantee termination one usually limits the number of retransmissions.

Boundedness properties. The second part of the state space report contains information about the *integer* and *multi-set bounds*. Table 6 shows the *upper* and *lower integer bounds*, i.e., the maximal and minimal number of tokens which may be located on the individual places in the reachable markings. The table only lists the places of the receiver and the sender since the remaining places of the CP-net are related via port assignments to places either in the sender or in the receiver. This means that the integer bound for these places can be inferred from those in Table 6. From the table we see that each of the places Sending and Waiting always has either one or no token, while all other places always have exactly one token each. This is not at all surprising, but it is reassuring, since it indicates that the system behaves as expected.

Table 6. Upper and lower integer bounds

Place	Upper Bound	Lower Bound
NextSend	1	1
ReceiveAck	1	1
Send	1	1
Sending	1	0
TransmitData	1	1
Waiting	1	0
NextReceive	1	1
ReceiveData	1	1
Received	1	1
TransmitAck	1	1

Table 7 shows the *upper multi-set bounds* for the places in the sender. By definition, the upper multi-set bound of a place is the smallest multi-set which is larger than all reachable markings of the place. The integer bounds give us information about the number of tokens, while the multi-set bounds give us information about the values which the tokens may carry.

From the multi-set bounds we can, for example, see that the place TransmitData may contain all five different data frames, and a token with value noframe corresponding to the frame buffer being empty. We also see that no other token values are possible for this place. Notice that the upper multi-set bound of TransmitData is a multi-set with six elements, although the upper integer bound tells us that there can never be more than one token on TransmitData at a time. Analogously, each of the places Sending and Waiting may contain all five possible data frames. The place Waiting may in addition contain the data frame (0,"") which is the data frame located on this place in the initial marking.

Also the upper multi-set bounds are as expected, and this further confirms that the protocol works as expected. However, the multi-set bounds also give us some new knowledge. As an example, there are probably many readers who did not notice, until now, that acknowledgement frames never carry the sequence number 0, as can be seen from the upper multi-set bound of the place ReceiveAck.

The *lower multi-set bound* of a place is the largest multi-set which is smaller than all reachable markings of the place. For all places in the stop-and-wait protocol the lower multi-set bound is the empty multi-set. This is

Table 7. Upper multi-set bounds

NextSend	1‘(0,acked) + 1‘(0,notacked) + 1‘(1,acked) + 1‘(1,notacked) + 1‘(2,acked) + 1‘(2,notacked) + 1‘(3,acked) + 1‘(3,notacked) + 1‘(4,acked) + 1‘(4,notacked) + 1‘(5,acked)
ReceiveAck	1‘ackframe(1) + 1‘ackframe(2) + 1‘ackframe(3)+ 1‘ackframe(4) + 1‘ackframe(5)+ 1‘noframe
Sending	1‘(0,"Software") + 1‘(1," Tools f") + 1‘(2,"or Techn") + 1‘(3,"ology tr") + 1‘(4,"ansfer. ")
TransmitData	1‘dataframe((0,"Software")) + 1‘dataframe((1," Tools f")) + 1‘dataframe((2,"or Techn")) + 1‘dataframe((3,"ology tr")) + 1‘dataframe((4,"ansfer. ")) + 1‘noframe
Waiting	1‘(0,"") + 1‘(0,"Software") + 1‘(1," Tools f") + 1‘(2,"or Techn") + 1‘(3,"ology tr") + 1‘(4,"ansfer. ")
Send	1‘["Software"," Tools f","or Techn", "ology tr","ansfer. "] + 1‘[" Tools f","or Techn","ology tr", "ansfer. "] + 1‘["or Techn","ology tr","ansfer. "] + 1‘["ology tr","ansfer. "] + 1‘["ansfer. "] + 1‘[]

because there are no places in the CPN model in which a token with a certain value is permanently present.

Home and liveness properties. The third part of the state space report is shown in Table 8. It provides information about *home* and *liveness properties*. A *home marking* is a marking which is reachable from all reachable markings, i.e., a marking which can always be reached – independently of what has previously happened. We see that the protocol has a single home marking, M_{862}. A *dead marking* is a marking with no enabled transitions. We see that the protocol has a single dead marking, and that the dead marking is identical to the home marking. At first glance, one might think that the existence of dead markings would prevent the existence of home markings. However, by definition, a marking is always reachable from itself, and hence a dead marking may be a home marking (provided that it is the only dead marking). This is the case for the protocol system. We will investigate the home and dead marking in more detail in a moment.

From the liveness properties we also see that there are no dead transitions. Dead transitions are similar to dead code in a programming language and it means that each transition is enabled in at least one reachable marking (which is a rather weak property). We also see that there are no live transitions. A live transition is a transition which can always, no matter what happens, become enabled again. When there are dead markings (as in our protocol), there cannot be any live transitions.

Table 8. Home and liveness properties

Home Markings:	[862]
Dead Markings:	[862]
Dead Transitions Instances:	None
Live Transitions Instances:	None

We now investigate marking M_{862} in more detail. Figure 35 shows marking M_{862}. Similarly to Fig. 34 the dashed box next to the node gives information about the marking of the individual places. The box shows the marking of all places in the sender and in the receiver with a non-empty marking. Inspection of the dashed box reveals that M_{862} is the desired terminating state of the protocol, in which all data packets have been sent, received in the correct order, and the frame buffers are empty. Since M_{862} is the only dead marking this implies that the protocol is partially correct – if the protocol terminates then it terminates in the desired state. Since M_{862} is a home marking the protocol always has a chance of terminating correctly. This means that it is impossible to reach a marking from which we cannot terminate the protocol with the correct result.

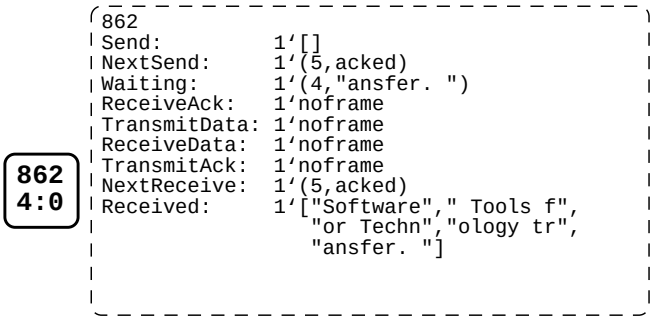

Fig. 35. Home and dead marking

Fairness Properties. The fourth and final part of the state space report is shown in Table 9. It provides information about the *fairness properties*, i.e., how often the individual transitions occur. We see that SendDataFrame, TimeOut, and Transmit 2 are *impartial*. This means that each of them occurs infinitely often in any infinite occurrence sequence. Transmit 2 is the transition on the second instance of page UniChannel. This is the instance which corresponds to the data channel. Hence, in any non-terminating execution of the protocol we keep having timeouts and we keep sending and transmitting data frames. Transition Accept is *fair* which is a weaker fairness property stating that the transition occurs infinitely often

Table 9. Fairness properties

Receive DataFrame	Fair
Send AckFrame	No Fairness
Accept	Fair
Receive AckFrame	No Fairness
Send DataFrame	Impartial
TimeOut	Impartial
Transmit 1	No Fairness
Transmit 2	Impartial

in all infinite occurrence sequences where it is infinitely often enabled.

8.4 Standard queries

In addition to the state space report, the tool also offers a set of *standard query functions* that allow the user to make a more detailed inspection of the standard behavioural properties. Many of these query functions return results which are already included in the state space report as described in the previous subsection. In fact, the state space report is implemented using a subset of the available standard query functions.

As an example, the tool has two functions, UpperInteger and LowerInteger which determine the maximal/minimal number of tokens residing on a set of places (where the state space report only considers individual places). The queries in Fig. 36 use these two functions to determine the maximal/minimal number of tokens on the places Sending and Waiting.

```
fun SendWaitMarking n =
    (Mark.Sender´Sending 1 n) +
    (Mark.Sender´Waiting 1 n) ;

UpperInteger (SendWaitMarking) ;
LowerInteger (SendWaitMarking) ;
```

Fig. 36. Example of standard query

It is easy to use the standard queries. For the queries in Fig. 36, the only thing which the user has to do, is to declare a function to be used as argument to UpperInteger and/or LowerInteger. This function maps a marking M_n (of the entire CPN model) into the multi-set $M_n(\text{Sending}) + M_n(\text{Waiting})$. The result of both queries is 1, and hence we have proved that there always is a token on Sending or a token on Waiting, but not on both places at the same time (see Fig. 33).

Analogously there is a function HomeSpace to determine whether a specified set of markings constitute a home space, i.e., whether it always is possible to reach one of the specified markings – independently of what has previously happened. The argument to this function is a list containing the set of markings under consideration.

The state space tool also provides standard queries to investigate multi-set bounds and to investigate reachability. Finally, there are queries to investigate liveness and fairness properties. These queries can be used on sets of transitions or on sets of binding elements.

8.5 Visualisation of state spaces

Since state spaces often become large, it seldom makes sense to draw them in full. However, the result of queries will often be a set of nodes and/or arcs possessing certain interesting properties, e.g., a path in the state space leading from one marking to another. A good and quick way to get detailed information on a small number of nodes and arcs is to draw the corresponding fragment of the state space. This makes visualisation particularly useful when locating errors in a system under consideration.

The state space can be drawn either in small steps, e.g., node by node or arc by arc, or in fragments using results from, for example, queries as input to a number of built-in drawing functions. Drawing of state spaces is often done in conjunction with interactive generation. Figure 34 was obtained by combining interactive generation with drawing.

To obtain detailed information about the drawn fraction of the state space, it is possible to attach *descriptors* to the nodes and arcs. The boxes positioned next to the nodes and arcs in Fig. 34 are examples of descriptors. For the nodes, the descriptors typically show the marking of certain places. For the arcs, the descriptors typically show the occurring transition and the binding of some or all of its variables. The descriptors have sensible defaults but the user may customise the descriptors by writing a script which specifies the contents and layout of the descriptor. The descriptors thus offer an abstraction mechanism by which the user can define a view of the state space.

To illustrate the use of drawing in combination with debugging we investigate what will happen if we modify the stop-and-wait protocol such that we only send an acknowledgement frame back to the sender when we receive the data frame that we were expecting. The state space for this variant of the protocol has 261 nodes and 609 arcs. It still has one dead marking in which all data packets have been sent, received in correct order, and the frame buffers are empty. However, this marking is no longer a home marking. This indicates that we can end up in a marking in which we cannot terminate the protocol correctly. To investigate this problem, we exploit the constructive feedback from the standard query function for checking whether a marking is a home marking. This feedback tells us that there exists an SCC from which it is impossible to reach the dead marking. By means of a standard query function, we see that this SCC only has eight nodes and that one of these is node 15. Using another query function we can find one of the shortest paths leading from the initial marking (node 1) to the node 15. This path is shown in Fig. 37, which is created by means

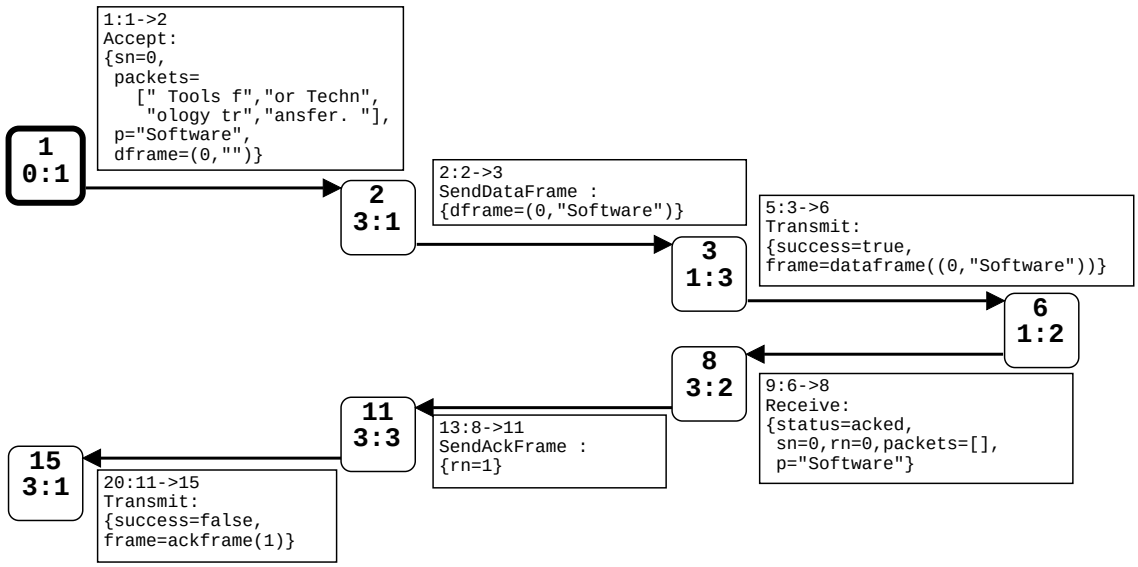

Fig. 37. Occurrence sequence as debugging information

of the built-in drawing functions. From the descriptors of the arcs on this path it is easy to identify the problem. To begin with, we successfully transmit the first data packet corresponding to the first three arcs in the path. Then we attempt to transmit the acknowledgement frame but it is lost. Hence, when we lose the acknowledgement frame we end up in a situation in which we cannot terminate the protocol correctly. The only thing that can happen from that point on is that we keep retransmitting the first data frame. These retransmissions cannot trigger an acknowledgement frame which would allow us to start transmitting the next data frame. The latter can easily be seen by drawing the eight nodes in the SCC together with the arcs that interconnect them.

8.6 Integration with simulation

During a modelling and design process, the user quite often switches between state space analysis and simulation. To support this, the state space tool is tightly integrated with the simulator, making it possible to transfer markings between the simulator and the state space tool.

When a marking is transferred from the state space into the simulator, it is displayed on the CPN diagram in the usual way. The user can inspect the marking of the individual places, and the enabling of the individual transitions. It is also possible to start a simulation from the transferred marking. As an example we could have checked that M_{862} was the desired dead marking by transferring the marking from the state space tool into the simulator and inspect the marking of the individual places. Figure 38 shows the top level page of the stop-and-wait protocol after M_{862} has been transferred into the simulator.

Transferring the current marking of the simulator into the state space is supported as well. A typical use of this is to investigate all possible markings reachable within a few steps from the current simulator marking. In this case, the user transfers the simulator marking into the state space tool and all successor markings can be found and drawn as illustrated above.

8.7 Advanced queries

The state space report and the standard queries are good for providing a rough picture of the behaviour of the system. However, they also have some limitations. First of all, many interesting properties of systems cannot easily be investigated using standard queries. Second, for debugging systems, more elaborate queries are often needed for locating the source of the problem. Therefore, a more general query language implemented on top of Standard ML (SML) is provided. It provides primitives for traversing the state space in different ways and thereby writing non-standard and model dependent queries. On top of

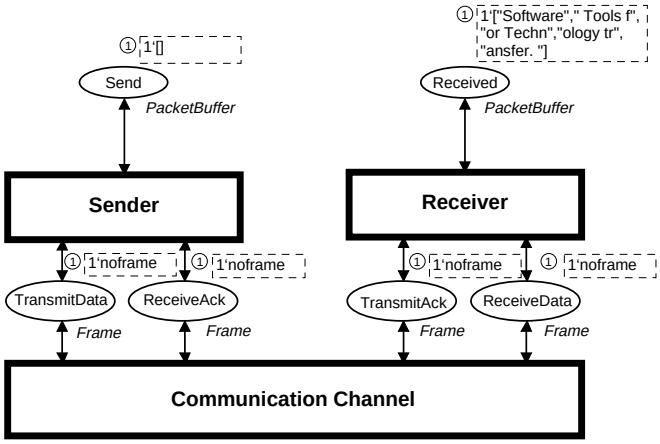

Fig. 38. Home and dead marking of the protocol

this general query language, a variant of the temporal logic CTL is provided [6, 15]. In this variant of CTL, it is possible to formulate queries about binding elements as well as markings corresponding to Petri nets being both action and state oriented.

Similar to standard queries, a non-standard query typically consists of a few lines of SML code. To illustrate this we will use the query language to investigate a system dependent property of the stop-and-wait protocol. We want to check that the sender is at most one data packet ahead of the receiver, i.e., that the stop-and-wait discipline is observed by the protocol. This can be proved by the non-standard query listed in Fig. 39. SWPredicate is a predicate which, given a marking, checks that the sum of the lengths of the lists on the places Send and Received differs by at most one from the total number of data packets to be sent. PredAllNodes is a predeclared query function, which returns all the nodes in the state space that satisfy a specified predicate. In the query we negate the SWPredicate and hence check the state space for nodes which violate the predicate. The operator o is the built-in infix operator in SML for functional composition.

```
val Length = length o ms_to_col ;
val total = Length (Mark.Sender´Send 1 1)

fun SWPredicate n =
  let
    val sent = total – (Length (Mark.Sender´Send 1 n)) ;
    val received = (Length (Mark.Receiver´Receiver 1 n))
  in
    ((sent = received) orelse (sent = received + 1))
  end ;

val nodelist = PredAllNodes (not o SWPredicate) ;
```

Fig. 39. Example of non-standard query

For the state space of the stop-and-wait protocol the result of the query is the empty list. Hence all nodes in the state space satisfy the SWPredicate, and we have proved that the stop-and-wait discipline is observed.

8.8 Advanced state space analysis

In this section, we have demonstrated that the state space method is an effective way to investigate the dynamic behaviour of a system. The construction and analysis of state spaces are totally automatic. From the state space report the user gains a lot of knowledge about the dynamic behaviour of the system. The state space method, however, also has some drawbacks.

The most serious drawback of the state space method is the so-called *state explosion problem*. For many systems, the state space becomes so large that it cannot be fully constructed even for small configurations of the system. A number of techniques exist for alleviating the state explosion problem. They do this by constructing a condensed state space from which it is still possible to verify the same kind of behavioural properties as with full ordinary state spaces. Condensed state spaces are typically orders of magnitude smaller than the full state space. Some of the techniques exploit the symmetry [14, 18, 31, 33, 35, 40] and equivalence [35, 39] which are present in many concurrent and distributed systems. Other techniques exploit the independency and concurrency between binding elements [24, 25, 41, 54, 55, 64], or take advantage of the hierarchical structure of the CPN model [13, 65]. The reader is encouraged to consult the papers cited above for a more detailed presentation and discussion of these techniques, their use, and how they are supported by the CPN state space tool.

Another drawback is the fact that a state space is always constructed for a particular initial marking, which often corresponds to only one out of many different possible system configurations. We have seen above that the protocol works when we have five data packets – but will it also work for more packets? In theory we cannot be totally sure. However, in practice the situation is not that bad, since most errors in system designs tend to manifest themselves readily in small configurations. Hence, it is very likely that our protocol will also work with more than five packets.

9 Conclusions

Coloured Petri nets have been developed over the past 20 years at the University of Aarhus, Denmark. The development has been balanced between the development of theory, supporting computer tools, and experiments in the form of practical case studies. All three parts have had a mutual beneficial influence on each other. The development of theory has been driven by the idea of combining the concurrency theory of Petri nets with the practical applicability of high-level programming languages. The theory has been important for developing computer tools founded on a solid and sound theory. The development of computer tools has been necessary to be able to make practical experiments with the aim of assessing the practical applicability of the developed theory. The practical case studies have led to new theoretical developments. A number of the case studies have been conducted by external users. This has provided valuable input for the further development of CP-nets and their supporting tools, and has demonstrated the practical applicability of CP-nets.

In this paper we have demonstrated the application of CP-nets for modelling and analysis of a rather simple communication protocol. It is obvious that compared to a simple communication protocol considerably more effort has to be invested when modelling and analysing industrial-sized systems. The feasibility of doing this has been demonstrated in a large number of dif-

ferent projects. This includes the projects described in the subsequent papers of this special section on coloured Petri nets. Other examples are projects within the areas of communication protocols [21, 30], audio/ video systems [9], operating systems [7, 8], hardware designs [22, 62], embedded systems [59], software system designs [46, 61], and business process re-engineering [49, 56] and the industrial projects listed on the web site [51]. The reader is referred to the individual papers for an elaborate assessment of the resources needed to apply CP-nets on industrial-sized systems.

As part of larger case studies (see e.g., [9, 59]), knowledge has also been gained on the resources which people without prior knowledge of CP-nets need to invest in order to be able to start applying CP-nets. The experiences have been encouraging. Typically, only about one or two weeks of intensive training are needed in order to be able to construct and simulate the first CPN models. CP-nets and the Design/CPN tool have been used in a course on distributed systems [12] at the University of Aarhus. After a period of about four weeks (with 15 hours of work per week) the students are able to make a CPN model of a non-trivial protocol stack and validate its correctness by means of simulations.

A comprehensive web site has been set up [51] with resources and technical support on the use of CP-nets and the Design/CPN tool. It contains manuals, tutorials, examples, answers to frequently asked questions (FAQs), a list of known problems, tips and tricks, etc. Moreover, it explains how to get a free license for the Design/CPN tool. The tool is currently available for Sun Solaris, HP Unix, and Linux platforms. A mailing list has been established as a forum for discussion among users of the Design/CPN tool.

The first workshop on practical use and application of CP-nets and Design/CPN (CPN'98) [38] was held in Aarhus in June, 1998 with the participation of 40 users from 10 different countries. The workshop contained presentations of regular papers, tutorials with practical exercises, and discussions and presentations of future plans for Design/CPN. Selected papers from the workshop are published in this special section of the STTT journal. The second workshop CPN'99 will be held in Aarhus in October, 1999.

The first version of Design/CPN appeared in 1989. Based on the knowledge gained from the practical use of the tool, the internal data structures and algorithms of the CPN simulator have been re-designed and are currently in the process of being implemented. The first results show that the new simulator runs orders of magnitude faster than the current version of the CPN simulator. The new simulator also contains an improved interface for declarations of types, variables, and functions. This new interface reduces the turn-around time between the editor and simulator when working with large-scale CPN models. The first version of this new simulator is released in 1999. A similar development is planned for the CPN state space tool. This work is expected to significantly push the borderline of systems that can be analysed by state spaces.

The area of automatic code generation from CP-nets is also a topic of ongoing work. Until now CP-nets have mostly been applied for the design and analysis of systems, and the transition to the final implementation has been done manually. Some first experiments in the area of automatic code generation were reported in [59]. The promising work of this project is currently being continued.

Acknowledgements. Many students and colleagues – in particular at the University of Aarhus and Meta Software – have influenced the development of CP-nets, their analysis methods, and their tool support. The development has been supported by several grants from the Danish Natural Science Research Council. A more detailed description of individual contributions can be found in the prefaces of [34–36]. For the present paper, we are grateful for the comments and suggestions provided by the reviewers.

References

1. Appel, A.W., MacQueen, D.B.: Standard ML of New Jersey. In: Maluszyński, J., Wirsing, M. (eds.): Third International Symposium on Programming Languages Implementation and Logic Programming. LNCS 528. Berlin, Heidelberg, New York: Springer-Verlag, 1991
2. Bertsekas, D., Gallager, R.: Data Networks. Prentice-Hall, 1992
3. Campos, S., Clarke, E.: Analysis and Verification of Real-time Systems using Quantitative Symbolic Algorithms. Software Tools for Technology Transfer, To appear.
4. Capellmann, C., Christensen, S., Herzog, U.: Visualising the Behaviour of Intelligent Networks. In: Margaria, T., Steffen, B., Ruckert, R., Posegga, J. (eds.): Services and Visualisation, Towards User-Friendly Design. LNCS 1385. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 174–189
5. ITU (CCITT). Recommendation z.120: Msc. Technical report, International Telecommunication Union, 1992
6. Cheng, A., Christensen, S., Mortensen, K.H.: Model Checking Coloured Petri Nets Exploiting Strongly Connected Components. In: Spathopoulos, M.P., Smedinga, R., Kozák, P. (eds.): Proceedings of the International Workshop on Discrete Event Systems, WODES96. Institution of Electrical Engineers, Computing and Control Division, Edinburgh, UK, 1996
7. Cherkasova, L., Kotov, V., Rokicki, T.: On Net Modelling of Industrial Size Concurrent Systems. In: Ajmone-Marsan, M. (ed.): Proceedings of ICATPN'93. LNCS 691. Berlin, Heidelberg, New York: Springer-Verlag, 1993
8. Cherkasova, L., Kotov, V., Rokicki, T.: On Scalable Net Modeling of OLTP. In: Proceedings of the 5th International Workshop on Petri nets and Performance Models, Toulouse, France. IEEE Computer Society Press, 1993
9. Christensen, S., Jørgensen, J.N.: Analysis of Bang and Olufsen's BeoLink Audio/Video System Using Coloured Petri Nets. In: Azéma, P., Balbo, G. (eds.): Proceedings of ICATPN'97. LNCS 1248. Berlin, Heidelberg, New York: Springer-Verlag, 1997
10. Christensen, S., Jørgensen, J.B., Kristensen, L.M.: Design/CPN - A Computer Tool for Coloured Petri Nets. In: Brinksma, E. (ed.): Proceedings of TACAS'97. LNCS 1217. Berlin, Heidelberg, New York: Springer-Verlag, 1997, pp. 209–223
11. Christensen, S., Kristensen, L.M.: State Space Analysis of Hierarchical Coloured Petri Nets. In: Farwer, B., Moldt, D., Stehr, M.-O. (eds.): Proceedings of Workshop on Petri nets in System Engineering – Modelling, Verification, and Validation. Department of Computer Science, University of Hamburg, 1997, pp. 32–43, Report no. 205

12. Christensen, S., Mortensen, K.H.: Teaching Coloured Petri Nets - A Gentle Introduction to Formal Methodsin a Distributed Systems Course. In: Azema, P., Balbo, G. (eds.): Proceedings of ICATPN'97. LNCS 1248. Berlin, Heidelberg, New York: Springer-Verlag, 1997, pp. 290–309
13. Christensen, S., Petrucci, L.: Modular State Space Analysis of Coloured Petri Nets. In: De Michelis, G., Diaz, M. (eds.): Proceedings of ICATPN'96. LNCS 935. Berlin, Heidelberg, New York: Springer-Verlag, 1996
14. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting Symmetries in Temporal Model Logic Model Checking. Formal Methods in System Design 9, 1996
15. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite State Concurrent Systems using Temporal Logic. ACM Transactions on Programming Languages and Systems 8(2): 244–263, 1986
16. Meta Software Corp. `http://www.metasoftware.com`
17. Meta Software Corporation. Design/OA. Meta Software Corporation, 150 Cambridge Park Drive, Cambridge MA 02140, USA
18. Emerson, E.A., Prasad Sistla, A.: Symmetry and Model Checking. Formal Methods in System Design 9, 1996
19. Esparza, J.: Decidability and Complexity of Petri Net Problems – An Introduction. In: Reisig, W., Rozenberg, G. (eds.): Lectures on Petri nets I: Basic Models. LNCS 1491. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 374–428
20. Billington, J., et al.: High-level Petri Nets – Concepts, Definitions and Graphical Notation. Technical report, International Standards Organization and International Electrotechnical Committee, October 1997, Commitee Draft 15909 ISO/IEC JTC1/SC7/WG11 N-3
21. Floreani, D.J., Billington, J., Dadej, A.: Designing and Verifying a Communications Gateway Using Coloured Petri Nets and Design/CPN. In: Billington, J., Reisig, W. (eds.): Proceedings of ICATPN'96. LNCS 1091. Berlin, Heidelberg, New York: Springer-Verlag, 1996
22. Genrich, H.J., Shapiro, R.M.: Formal Verification of an Arbiter Cascade. In: Jensen, K. (ed.): Proceedings of ICATPN'92. LNCS 616. Berlin, Heidelberg, New York: Springer-Verlag, 1992
23. Gibbons, A.: Algorithmic Graph Theory. Cambridge University Press, 1985
24. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Proceedings of CAV'90. LNCS 531. Berlin, Heidelberg, New York: Springer-Verlag, 1990, pp. 175–186
25. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems, An Approach to the State-Explosion Problem. LNCS 1032. Berlin, Heidelberg, New York: Springer-Verlag, 1996
26. Goldfarb, C.F., Rubinsky, Y.: The SGML handbook. Clarendon Press, Oxford, UK, 1990
27. The CPN Group. `http://www.daimi.au.dk/CPnets`
28. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice-Hall International Editions, 1991
29. Huber, P., Jepsen, A.M., Jepsen, L.O., Jensen, K.: Reachability Trees for High-level Petri nets. Theoretical Computer Science 45: 261–292, 1986, Also in Jensen, K., Rozenberg, G. (eds.): High-level Petri nets. Theory and Application. Berlin, Heidelberg, New York: Springer-Verlag, 1991, pp. 319–350
30. Huber, P., Pinci, V.O.: A Formal Executable Specification of the ISDN Basic Rate Interface. In: Rozenberg, G. (ed.): Proceedings of ICATPN'91, 1991
31. Ip, C.N., Dill, D.L.: Better Verification Through Symmetry. Formal Methods in System Design 9, 1996
32. Jensen, K.: An Introduction to the Theoretical Aspects of Coloured Petri Nets. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.): A Decade of Concurrency. LNCS 803. Berlin, Heidelberg, New York: Springer-Verlag, 1994, pp. 230–272
33. Jensen, K.: Condensed State Spaces for Colored Petri Nets. Formal Methods in System Design 9, 1996
34. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts. Monographs in Theoretical Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, 2nd corrected printing 1997, ISBN: 3-540-60943-1
35. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods. Monographs in Theoretical Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, 2nd corrected printing 1997, ISBN: 3-540-58276-2
36. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use. Monographs in Theoretical Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, 1997, ISBN: 3-540-62867-3
37. Jensen, K.: An Introduction to the Practical Use of Coloured Petri Nets. In: Reisig, W., Rozenberg, G. (eds.): Lectures on Petri nets II. LNCS 1492. Berlin, Heidelberg, New York: Springer Verlag, 1998, pp. 237–292
38. Jensen, K. (ed.): Workshop on Practical Use of coloured Petri nets and Design/CPN, number 532 in DAIMI PB. Department of Computer Science, University of Aarhus, 1998, Also available via `http://www.daimi.au.dk/CPnets/workshop98/papers/`
39. Jørgensen, J.B., Kristensen, L.M.: Verification of Coloured Petri Nets Using State Spaces with Equivalence Classes. In: Farwer, B., Moldt, D., Stehr, M.O. (eds.): Proceedings of Workshop on Petri nets in System Engineering – Modelling, Verification, and Validation. Department of Computer Science, University of Hamburg, 1997, pp. 20–31, Report no. 205
40. Jørgensen, J.B., Kristensen, L.M.: Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries. To Appear in IEEE Transactions on Parallel and Distributed Systems, 1999
41. Kristensen, L.M., Valmari, A.: Finding Stubborn Sets of Coloured Petri Nets Without Unfolding. In: Desel, J., Silva, M. (eds.): Proceedings of ICATPN'98. LNCS 1420. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 104–123
42. Larsen, K.G., Steffen, S., Wiese, C., Alur, R., Henzinger, T.A.: Special Section on Timed and Hybrid Systems. Number 1-2 in Software Tools for Technology Transfer. Berlin, Heidelberg, New York: Springer-Verlag, 1997
43. Loiseaux, G., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S.: Property Preserving Abstractions for the Verification of Concurrent Systems. Formal Methods in System Design 6, 1995
44. Lyngsø, R.B., Mailund, T.: Textual Interchange Format for High-level Petri Nets. In: Jensen, K. (ed.): Workshop on Practical Use of Coloured Petri nets and Design/CPN, number 532 in DAIMI PB. Department of Computer Science, University of Aarhus, 1998, pp. 47–65, Also available via `http://www.daimi.au.dk/CPnets/workshop98/papers/`
45. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Frabceschinis, G.: Modelling with Generalized Stochastic Petri nets. Series in Parallel Computing. Wiley, 1995
46. McLendon, W.M., Vidale, R.F.: Analysis of an Ada System Using Coloured Petri Nets and Occurrence Graphs. In: Jensen, K. (ed.): Proceedings of ICATPN'92. LNCS 616. Berlin, Heidelberg, New York: Springer-Verlag, 1992
47. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, 1993
48. Milner, R., Harper, R., Tofte, M.: The Definition of Standard ML. MIT Press, 1990
49. Mortensen, K.H., Pinci, V.: Modelling the Work Flow of a Nuclear Waste Management Program. In: Valette, R. (ed.): Proceedings of ICATPN'94 815. Berlin, Heidelberg, New York: Springer-Verlag, 1994
50. Murata, T.: Petri Nets: Properties, Analysis and Application. In: Proceedings of the IEEE, Vol. 77, No. 4. IEEE Computer Society, 1989
51. Design/CPN Online. `http://www.daimi.au.dk/designCPN/`
52. The Gnuplot Tool. Available online: `http://www.cs.dartmouth/ gnuplot_info.html.`
53. Paulson, L.C.: ML for the Working Programmer. Cambridge University Press, 2 edition, 1996
54. Peled, D.: All from One, One for All: On Model Checking Using Representatives. In: Proceedings of CAV'93. LNCS 697. Berlin, Heidelberg, New York: Springer-Verlag, 1993, pp. 409–423

55. Peled, D.: Combining Partial Order Reductions with On-the-fly Model Checking. Formal Methods in System Design, 1996
56. Pinci, V.O., Shapiro, R.M.: An Integrated Software Development Methodology Based on Hierarchical Coloured Petri Nets. In: Rozenberg, G. (ed.): Advances in Petri nets. LNCS 524. Berlin, Heidelberg, New York: Springer-Verlag, 1991, pp. 227–252
57. Information Processing: Text and office systems: Standard Generalized Markup Language (SGML). Technical report, ISO standard 8879, International Organisation for Standardization, Geneve, Switzerland, 1986, Amendment from 1988 exists.
58. Rasmussen, J.L., Singh, M.: Mimic/CPN. A Graphical Simulation Utility for Design/CPN. User's Manual. Available from `http://www.daimi.au.dk/designCPN`
59. Rasmussen, J.L., Singh, M.: Designing a Security System by Means of Coloured Petri Nets. In: Billington, J., Reisig, W. (eds.): Proceedings of ICATPN'96. LNCS 1091. Berlin, Heidelberg, New York: Springer-Verlag, 1996
60. Reisig, W.: Petri nets, volume 4 of EACTS Monographs on Theoretical Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, 1985
61. Scheschonk, G., Timpe, M.: Simulation and Analysis of a Document Storage System. In: Valette, R. (ed.): Proceedings of ICATPN'94. LNCS 815. Berlin, Heidelberg, New York: Springer-Verlag, 1994
62. Shapiro, R.M.: Validation of a VLSI Chip Using Hierarchical Coloured Petri Nets. Journal of Microelectronics and Reliability, Special Issue on Petri nets, 1991
63. Ullman, J.D.: Elements of ML Programming. Prentice-Hall, 1997
64. Valmari, A.: A Stubborn Attack on State Explosion. In: Proceedings of CAV'90. LNCS 531. Berlin, Heidelberg, New York: Springer-Verlag, 1990, pp. 156–165
65. Valmari, A.: Compositionality in State Space Verification Methods. In: Billington, J., Reisig, W. (eds.): Proceedings of ICATPN'96. LNCS 1091. Berlin, Heidelberg, New York: Springer-Verlag, 1996
66. Wells, L., B. Lindstrøm. The Design/CPN Performance Tool Manual. Department of Computer Science, University of Aarhus, 1998. On-line version: `http://www.daimi.au.dk/designCPN/`