

Simulation Based Performance Analysis in Design/CPN

Bo Lindstrøm and Lisa Wells

Department of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C
Denmark
E-mail: {blind,wells}@daimi.aau.dk

Abstract. This paper describes the design of facilities for doing simulation based performance analysis. The performance facilities have three main components: functions for generating random numbers from different distributions, statistical variables for collecting different data while simulating, and reporting facilities for generating output from the statistical variables. We also describe the integration of the performance facilities into the Design/CPN tool. To illustrate the usability of the performance facilities, we give a non-trivial example of simulation based performance analysis by analysing a multiaccess protocol.

Keywords. Performance analysis, Coloured Petri Nets, simulation, random distributions, networks and multiaccess protocol.

1 Introduction

Most applications of Coloured Petri Nets (CP-nets) [Jen92,Jen94,Jen97] are used to investigate the logical correctness of a system. This means that we consider the dynamic properties and the functionality of the system. However, CP-nets can also be used to investigate the performance of a system, e.g., the maximal time used for the execution of certain activities and the average waiting time for certain requests. To perform this kind of analysis we often use timed CP-nets. A timed CP-net is a CP-net extended with a *global clock* (continuous or discrete) which represents the model time. Each token is now allowed to carry a *time stamp*. A binding element is *ready* when the time stamps of the removed tokens are less than or equal to the current model time. A binding element is *enabled* if it is colour enabled (enabled in an ordinary CP-net) and ready. The global clock advances when none of the colour enabled binding elements are ready. See Chap. 5 in [Jen94] for further details.

While the Design/CPN tool [Jena] supports state space analysis [CK97], timed simulations and functional analysis [CJ97], it lacked integrated support for performance analysis of a CPN model. Previously, all collection of data had to be explicitly defined and coded by the user. This, in turn, meant that the user had to be familiar with untimed statistical variables¹ and the use of code segments. An untimed statistical variable is a data type with which it is possible to collect some values and later on extract different statistical information such as sum or average [Jenb]. Examples of CPN models with explicitly coded performance analysis can be found in Chap. 2, 4, 12 and 15 in [Jen97].

The aim of this paper is to present the performance facilities [LWb] which have been integrated into the Design/CPN tool and which remedy the above shortcomings of the tool. These performance facilities provide distribution functions² for generating random numbers and high-level support for collecting statistics during simulation. Finally it is possible to create reports containing these statistics.

¹ Originally called statistical variables.

² In this paper a *distribution function* refers to a function returning a sample (integer or real) from the given probability distribution function.

When constructing a CPN model of real world phenomena, we are often interested in using random numbers from a specific distribution due to our knowledge of the behaviour of the modelled phenomena. The implemented distribution functions allow the user to draw random samples from several different kinds of distributions. These functions allow the user to construct, e.g. delay or input to the CPN model with values from a specific distribution. For example, when constructing a CPN model of the arrival of packages to a network we often assume that this arrival is Poisson distributed.

During simulation of a CPN model, we are often interested in evaluating the performance of the CPN model. To do this it can be necessary to extract different values from the markings of the CPN model during simulations. By using the statistical variables which are part of the performance facilities it is possible to collect these values from the CPN model while simulating. An example illustrating the use of statistical variables is a CPN model simulating a physical network. In such a CPN model we could be interested in knowing the average number of packages on the network per time unit. Hereby, we get a performance measure of the load of the network.

The performance facilities provide an easy way to specify the values from a marking and a binding element to be examined during a simulation. This can be done by using some of the predefined functions which measure, for example, number of tokens on a place or average length of the lists on a place. The user is also able to make his own functions to extract a value from a marking and a binding element of a CP-net to update the statistical variable. Other facilities such as inserting, deleting and initialising a statistical variable and maintaining a log file with the observed values during a simulation are also available.

This paper is organised as follows: In Sect. 2 we first give an overview of the incorporation of the performance facilities into the Design/CPN tool. Section 3 contains a model which will serve as an example throughout the rest of the paper. In Sect. 4 we briefly describe the distribution functions. The statistical variables are described in Sect. 5. In Sect. 6 we explain how to use the performance facilities. Finally in Sect. 7 we draw some conclusions including ideas for future work.

2 Performance Analysis in Design/CPN

In this section the incorporation of the performance facilities into the Design/CPN tool is described. The overall structure of the facilities is illustrated in Fig. 1. The items (except the item *Simulator*) illustrate the new performance facilities in the Design/CPN tool. The rectangles illustrate the main components of the tool while the dashed ellipses indicate the existing libraries which have been extended and integrated into the tool. The solid line ellipses indicate output produced by the performance facilities.

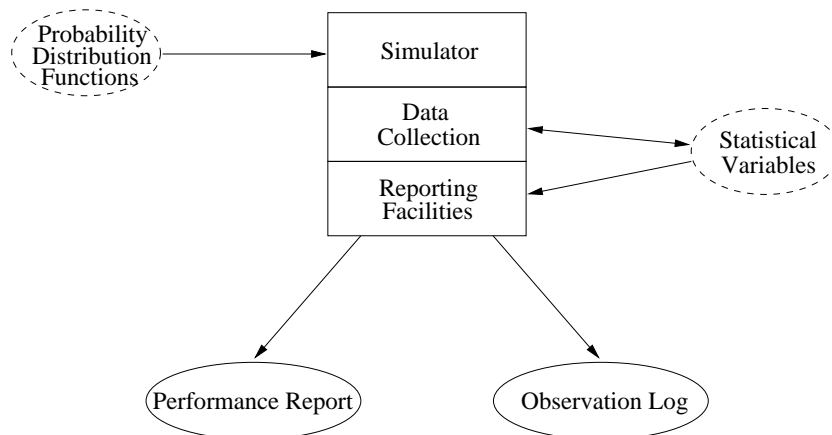


Fig. 1. Overview of the Tool.

The execution of the model in the Design/CPN tool using the performance facilities is as follows: the *Simulator* simulates the CPN model possibly using the *Probability Distribution Functions* for generating random numbers. At the same time, the *Simulator* does *Data Collection* and updates the *Statistical Variables* with the new observed values. Finally, the *Reporting Facilities* dump all of the observed values in a detailed *Observation Log* file. At any point in the simulation, the user can also request a *Performance Report* to be generated using the *Reporting Facilities*. A performance report shows the current status of the statistical variables, e.g. sum, sum of the squares, average and variance. In this way a *Performance Report* gives a more abstract view of the observed values than the view of an *Observation Log* file.

The performance facilities are mainly implemented in Standard ML [AM91]. The graphical interface is currently being implemented in C.

3 Example: Multiaccess Protocol

This section contains a non-trivial example of a CPN model that uses the performance facilities of Design/CPN. The example is a model of a protocol from one layer of a network architecture. The model presented here will be used in the following sections to illustrate how the performance facilities can be used. A detailed description of the protocol can be found in [BG92], and a brief outline of the protocol follows in Sect. 3.1. Section 3.2 contains detailed description of the model.

3.1 Aloha Protocol

The protocol to be modelled is the Aloha protocol which is a multiaccess protocol from the medium access control (MAC) layer of a network architecture. Multiaccess communication is communication between several sources, or nodes, across a shared communication medium, e.g. communication via an Ethernet or a satellite system. The purpose of the MAC layer is to allocate use of the shared communication medium among the competing nodes.

The protocol is used to coordinate communication between $m \geq 1$ transmitting nodes. For the sake of simplicity, we will assume that all messages are sent to one receiver and that the receiver is responsible for providing feedback. The nodes communicate by sending packets via a shared communication channel. The following assumptions are made about the system:

1. *Slotted system.* All packets have the same size, and each packet can be transmitted in one time unit, in the following referred to as a slot.
2. *Poisson arrivals.* Packets to be transmitted arrive at each node according to independent Poisson processes. Let $\frac{\lambda}{m}$ be the arrival rate for each node.
3. *Collision or perfect reception.* A *collision* occurs if two or more nodes send packets in a given time slot. In this case, the receiver obtains no information about the contents or senders of the packets. If only one node transmits a packet in a slot, then it is correctly received by the receiver.
4. *0,1,e Immediate feedback.* At the end of each slot, each node receives feedback from the receiver indicating whether 0 packets, 1 packet, or more than one (*e* for error) were transmitted in that slot.
5. *Retransmission of collisions.* Each packet that collides with another must be retransmitted. A packet is retransmitted until it is successfully received. A node is said to be *backlogged* if it has a packet that must be retransmitted.
6. *Buffering.* Each node has a buffer containing packets to be transmitted to the receiver. These packets have been received from the datalink control (DLC) layer.

The purpose of the protocol is to coordinate and make effective use of the communication channel. This is achieved by minimising both the number of collisions and the number of slots in which no packets are sent, or alternatively, by maximising the number of slots in which exactly one packet is sent. When a collision occurs, the transmitting nodes should not automatically retransmit their packets in the following slot because the packets would obviously collide once again. Thus, the basic idea of the Aloha protocol is to determine when backlogged and unbacklogged nodes should transmit packets. If a node is not backlogged, and its buffer is not empty, then it transmits a packet at the beginning of the next slot. Backlogged nodes retransmit in the following slots with some fixed probability $q_r > 0$ until the packet is successfully transmitted.

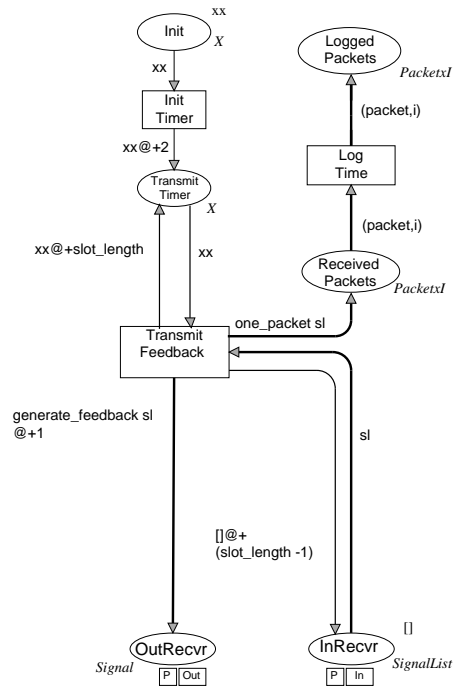


Fig. 3. Receiver in Aloha protocol.

States models the state of a node: whether it is *ready* (unbacklogged), *backlogged*, or in the process of *sending* a packet. The number of backlogged nodes in the system is calculated from the marking of this place. The values on *Retransmit* are compared to q_r , and are thus used to determine whether a backlogged node can retransmit in a given slot.

Whether or not a node can transmit a packet in the current time slot is determined by its state, its buffer, and possibly by the retransmission probability. If there are packets on *DLCtoMAC* for node *node*, then they can be added to the end of *node*'s buffer. At the beginning of a slot, a node is either *ready* or *backlogged*. If a node's buffer is empty, then it must be *ready*, however, the transition *Transmit Packet* is not enabled, so no packets are sent from that node. When a node's buffer is not empty, and the node is *ready*, then the packet at the front of the buffer is automatically transmitted, and the node is transferred to *sending* state. In the last case, a *backlogged* node is allowed to send a packet only if the value from the place *Retransmit* is less than the predetermined retransmission probability which can be found on the place *QR*. When a *backlogged* node is allowed to retransmit a packet, its state is also changed to *sending*, while a *backlogged* node that cannot retransmit remains *backlogged*. The function *temp_state*³ will change a node's state to *sending* when necessary. Each time a node attempts to transmit a packet, a token is removed from *Retransmit*, and then a new token with a time stamp which corresponds to the beginning of the next slot is returned to the place. In this way, nodes are prevented from attempting to transmit more than once per slot by the new time stamp.

Once all packets have been received from the DLC layer, and all nodes with packets have attempted to transmit a message, the receiver (Fig. 3) becomes active. The place *InRecvr* contains a list of the packets that were transmitted in the beginning of the current slot. The receiver uses this list to determine which type of message to broadcast to all nodes. If the list is empty, then the receiver sends a *zero* message indicating that no packets were received. A collision occurred if the list has length ≥ 2 , in which case, the receiver broadcasts an *e* (for error) message to all nodes. Finally, when the list contains only one packet, then the receiver broadcasts a *one* and saves the

³ This function can be found on the arc from *Transmit Packet* to *States* in Fig. 2.

packet. The packet is saved in order to register delay. Again, a time stamp is used (on the token on *Transmit Timer*) to restrict the receiver to broadcasting messages exactly once per slot.

The nodes (Fig. 2) become active again after the receiver has broadcasted the appropriate messages. These messages can be found on place *InNode*. Recall that the nodes can be in *sending*, *backlogged* or *ready* states. If a node is either *ready* or *backlogged*, then it is only necessary to remove its feedback message from *InNode* because it has not attempted to send a packet in the current slot. If a node is in the *sending* state and the feedback from the receiver is *one*, then the packet at the head of the node's buffer was successfully transmitted, which means that it can be removed from the buffer. At the same time, the node's state must be changed to *ready*, indicating that it can transmit a packet in the next slot. Feedback *e* comes in response to a collision, so every node that is *sending* must become *backlogged*, and no changes are made to buffers because no packets were successfully transmitted. The function `end_state`⁴ changes a node's state accordingly.

4 Distribution Functions

In this section we give a short description of the implemented distribution functions⁵. When modelling a physical phenomenon we often know that this particular phenomenon has a specific behaviour. This knowledge leads to a need for different functions that return random numbers from different distributions. By using such functions we are able to construct a more precise model of the physical phenomenon which can lead to better results when simulating and analysing the model. An example of a phenomenon which we know has such a specific behaviour can be the delay between busses arriving at a bus stop.

While analysing the built-in random number generator in Design/CPN, it became clear that this random number generator is not completely satisfactory. It is shown in [Dri] that this random number generator produces numbers that can not be considered especially random. This led to constructing a new random number generator which has been proven to be better than the old one.

The distribution functions have then been implemented on top of the new random function. This random function gives samples from a standard uniform distribution. The different distribution functions are then implemented by applying different procedures to this random function.

The implemented distribution functions are: Bernoulli, binomial, chi-square, discrete uniform, Erlang, exponential, normal, student and Poisson distribution. See [Rip87] for further information about the distributions. A detailed description of the analysis, implementation and interface can be found in [Dri].

To illustrate the use of the distribution functions we refer to the example model in Sect. 3.2, where two distribution functions have been used to generate random values. The first one is used to generate the packets from the DLC layer arriving at the nodes. These packets are placed on the input place *DLCToMAC* in Fig. 2. The packets are generated by the function `generate_packets` (see below) using the Poisson distribution function with arrival rate $\frac{\lambda}{m}$ for each node (in accordance with assumption 2, Sect. 3.1).

```

fun generate_packets 0 = empty
  | generate_packets node =
    let
      val num = poisson((!lambda)/(real(!m)))
    in
      num'(node, (pack,time())) + generate_packets (node-1)
    end;

```

⁴ This function can be found on the arc from *Check Feedback* to *States* in Fig. 2.

⁵ Theo van Drimmelen [Dri] is solely responsible for the design and implementation of the distribution functions presented in this section.

As mentioned previously, the values of the tokens on the place *Retransmit* are used to determine whether a backlogged node can retransmit a packet in a given slot, therefore new values must be generated for each node every time it attempts to send a packet. Each time the transition *Transmit Packet* occurs it is necessary to find a new random value in the interval (0,1). These values are generated using the uniform distribution function, and the code segment for *Transmit Packet* sets $pr = \text{uniform}(0.0, 1.0)$, where pr can be found on the arc from *Transmit Packet* to *Retransmit*.

5 Statistical Variables

This section describes the concept of statistical variables. Statistical variables are data structures providing the ability both to collect values from a simulation and to access statistics about these values during the simulation of a model. The values accumulated in a statistical variable can be integers or reals. Two types of statistical variables with different behaviour are available: timed and untimed.

The original implementation and design of untimed statistical variables was done by Alain Karsenty (see [Jenb]). We have modified the implementation of the untimed statistical variables to be more time-effective. Furthermore, we have added the timed statistical variables.

The values and statistics that can be accessed in both types of statistical variables are: minimum and maximum observed value, number of observations, sum, sum of squares, average, sum of squares of deviation, standard deviation and variance. Timed statistical variables include additionally: time of first update, time of last update and time interval (which indicates how much time has elapsed since the statistical variable was first updated).

Updates and accesses of a statistical variable can be freely intermixed. A statistical variable may also be reinitialised at any time, allowing a new set of values to be accumulated.

In our implementation of statistical variables [LWa], we do not save all the individual values but instead only calculate the values needed to be able to access the above mentioned statistics. This approach saves a lot of memory compared to accumulating all the values used for calculating the statistics.

Untimed Statistical Variables.

Figure 4 shows how an untimed statistical variable is updated with different observed values. If

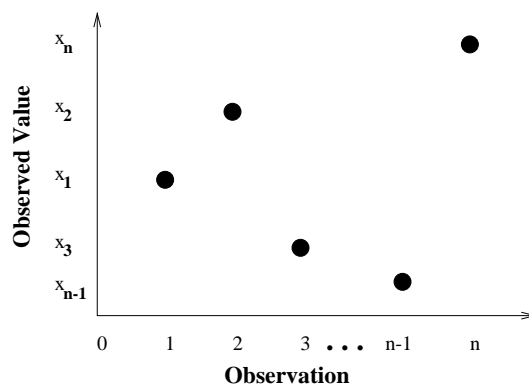


Fig. 4. Observed values for untimed statistical variables.

we update an untimed statistical variable with the the same value twice then the value influences the statistics twice, as expected. The sum is calculated in the following way:

$$Sum_n = \sum_{i=1}^n x_i$$

Timed Statistical Variables.

Timed statistical variables differ from untimed statistical variables in that an interval of time is used to weight each observed value. Assume that at time t_n , a timed statistical variable is updated with a new value x_n . At precisely time t_n , x_n has no influence on sum, sum of the squares, average, sum of the squares of deviation, standard deviation or variance – the weight is zero, but for all time $t > t_n$, x_n will influence these values. As an example, consider the graph in Fig. 5.

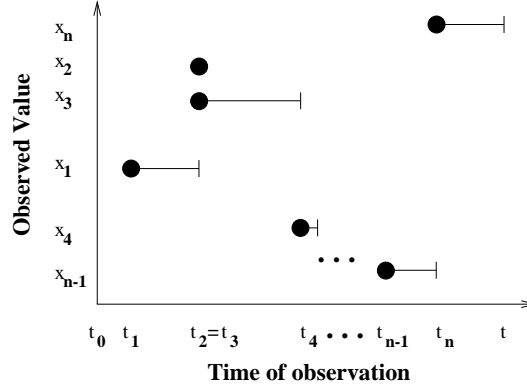


Fig. 5. Observed values for timed statistical variables.

Assume that a timed statistical variable was created at the time t_0 and that the circles indicate when the timed statistical variable was updated. The variable was last updated at time t_n with value x_n . Let us look at how the sum and the average of the values are calculated at time $t \geq t_n$.

$$Sum_t = (\sum_{i=1}^{n-1} x_i * (t_{i+1} - t_i)) + x_n * (t - t_n) \text{ if } t \geq t_n$$

$$Average_t = \frac{Sum_t}{t - t_1} \text{ if } t \geq t_n$$

With timed statistical variables, it is possible for a value to exist for zero time, see for example Fig. 5 where the value x_2 at time t_2 exists for zero time because it is followed by an update with the value x_3 at the same model time $t_3 = t_2$. In this situation we note that, in contrast to the above mentioned functions, the maximum, minimum and number of observations take into account all the values with which the statistical variable has been updated, i.e. including the ones which have existed for zero time (e.g. the value x_2 in Fig. 5). This is due to the fact that the functions maximum, minimum and number of observations are the only ones not weighted with the time elapsed since the last update.

Technical Remark. Consider a timed simulation with integer time. If the time advances with value one for each update of a timed statistical variable, then the statistics for the timed and untimed statistical variables are equal, assuming that we access the statistics one time unit after the last update. The reason is that in this situation the weight is one for each value in the timed statistical variable.

6 How to use the Performance Facilities

In this section we describe the performance facilities provided for defining and manipulating statistical variables. The statistical variables described in Sect. 5 are incorporated into the Design/CPN tool. The performance facilities provide an easy way for defining when and how the statistical

variables should be updated. Reporting capabilities and maintenance of log files are additional components in the performance facilities. It is possible to maintain several different statistical variables (timed or untimed with integer or real values) – each being updated by extracting different values from markings and binding elements during a simulation. It is also possible to create and delete statistical variables after having simulated some steps. Furthermore, a statistical variable can be reinitialised at any time during a simulation.

When using the performance facilities, it is necessary to define two functions for each statistical variable. Definitions and examples of these two functions follow. Sections 6.1 and 6.2 introduce predicate functions and observation functions, respectively. In Sect. 6.3 observation log files are discussed. Finally, Sect. 6.4 contains an explanation of the performance report.

6.1 Predicate Function

A predicate function determines how often a statistical variable is updated. This function is a mapping from a marking and a list of binding elements to a boolean value. A predicate function is evaluated after each step of the simulator. If it evaluates to *true*, then the corresponding statistical variable is updated with an observed value, otherwise no update is made. If a function that always returns *true* is used as a predicate, then the statistical variable will be updated after each step in the simulation. When creating a statistical variable and later examining the performance report generated by the reporting facilities, it is important to be aware of when and how often an update of the statistical variable is performed.

The timed statistical variables have an additional note. An implication of the definition of a timed statistical variable (see Sect. 5) is that only the value observed at the time of the last occurrence of a transition before the time advances influences the values (sum, sum of the squares, etc.) of the *timed* statistical variables. As mentioned above, there are a few exceptions which are influenced: number of observations, minimum and maximum. This means that one has to be very careful when designing the predicate function.

We now take a closer look at some of the predicate functions that were used with the model presented in Sect. 3.2. The function `isSubSlot4` is a predicate function which returns *true* whenever the model time has reached the last subplot of a slot. The transition *Check Feedback* in Fig. 2 is enabled only in the last subplot, so it makes sense to record the number of backlogged nodes right after the states of the nodes have been updated to reflect whether or not they are backlogged.

```
fun isSubSlot4 (ogrec:CPN'OGrec, bes: Bind.Elem list) =
  ((time() mod 4) = 3);
```

The function `Transmit_Feedback_Occurred` is another example of a predicate function. This particular predicate function is interesting because it returns *true* only when a certain transition has occurred, i.e. the return value depends on the binding element from the current step. The local function `Transmit1` evaluates to *true* only when the transition *Transmit Feedback* occurs and *sl* is a list with one element. This predicate function is used when calculating delay of a packet, since it evaluates to *true* precisely when one packet has been successfully transmitted.

```
fun Transmit_Feedback_Occurred (ogrec:CPN'OGrec, bes:Bind.Elem list) =
  let
    fun Transmit1 [] = false
      | Transmit1 ((Bind.MAC_Receiver'Transmit
                    (1,{sl=((node_pack _)::[]),...})):rest) = true
      | Transmit1 (_:rest) = (Transmit1 rest)
  in
    Transmit1 bes
  end;
```

6.2 Observation Function

An observation function is used to calculate the values with which the corresponding statistical variable is updated. This function is a mapping from a marking and a binding element to either an integer or a real. The return value of an observation function is used to update a statistical variable, provided that the corresponding predicate function evaluates to *true* after a step in a simulation.

It is important to remember that a predicate function determines how often a statistical variable is updated. Consider an observation function that returns a value that is solely dependent on the marking of a particular place. If the predicate function returns *true* after each step, then the statistical variable will be updated with the observed value even though the marking of the place from which the observed value has been extracted may not have changed.

Three predefined observation functions for performance measures are currently available in Design/CPN. The first one, *Number_of_Tokens*, records how many tokens are on a place. The other two are concerned with list length. For a place that has a colour set which is a list, *Average_List_Length* calculates the average list length given a multi-set of lists from that place. *Combined_List_Length* will return the sum of the lengths of lists.

For each statistical variable, the user can either use one of the predefined observation functions or define his own function to extract a specific value from a marking and a binding element. There are no limitations on how a value is computed. For example, the user-defined function can be a function which calculates the average length of lists from a number of places.

Another example showing the generality of the observation function could be a statistical variable counting the number of tokens that arrive to a place. In this way it is possible to get the average number of arrivals per step (or time-unit, if simulating with time and having chosen a timed statistical variable). This could be done by letting the observation function extract from the binding element the number of tokens arriving to the particular place.

The function `Backlogged_nodes` is an observation function that is used in the example model. In this function, the local variable *mark* refers to the marking of the place *States*. The local function `countBacklogged`⁶ determines how many nodes are *backlogged* for a given marking from the place *States*. In order to keep track of the number of backlogged nodes, it is only necessary to define `Backlogged_nodes` and `isSubSlot4` for updating a timed statistical variable. In other words, a statistical variable is updated with the number of backlogged nodes each time a final subplot is reached in the simulation.

```
fun Backlogged_nodes (ogrec:CPN'OGrec, be: Bind.Elem) =
  let
    val mark = OEMark.MAC_Node'States 1 ogrec
    fun countBacklogged empty = 0
      | countBacklogged ((coef,(node,backlogged),ts) !!! rest) =
        coef + (countBacklogged rest)
      | countBacklogged ((-,_,_) !!! rest) = (countBacklogged rest)
  in
    countBacklogged mark
  end;
```

6.3 Observation Log

It is possible to associate an observation log file with each statistical variable. This log file will be updated each time the statistical variable is updated. For timed simulations, it is possible to record the time of the update as well as the new value. In this way we get a file with all the observed values which can be examined after the simulation. This file could be used, for example,

⁶ The multi-set operator `!!!` is used in the internal representation of timed multi-sets in Design/CPN.

to draw a graph of the values with which the statistical variable has been updated (relative to the time of the update – if using a timed simulation).

The current observation log file format can be plotted by Gnuplot [Gnuplot]. Figure 6 is a Gnuplot graph created from observation log files produced when tracking buffer length for nodes 1 and 2 in the model. As one would expect, the buffer lengths continued to grow since the nodes were frequently backlogged and, therefore, unable to send packets and reduce the number of packets in their buffers.

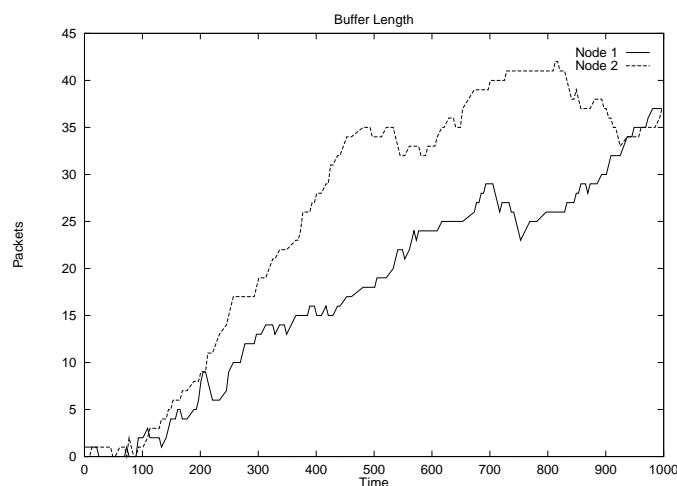


Fig. 6. Buffer length from log files.

6.4 Performance Report

After simulating a CPN model and thereby collecting some statistics, it is possible to print a report containing the different values of the statistical variables, e.g sum, last value, average, etc. This is done by specifying which of these values the user wants to be included in the report. It is also possible to specify both how many decimal positions should be shown and how wide the columns should be.

An example of a performance report which can be generated from the statistical variables can be found in Table 1. The first column, **Observation function**, is always part of the performance report. The values in this column are the names of the observation functions that were used to update the statistical variables. Each untimed statistical variable can produce nine values (sum, average, ...) while timed statistical variables can produce twelve (see Sect. 5). However, one is not always interested in knowing all of the values, so it is possible to indicate which subset of the values should be contained in the report. Here, for example, we chose to include number of updates, sum, average, and maximum, while disregarding the rest.

Contents of Statistical Variables. In this particular model, ten statistical variables were used. Three of these (Lines 7-9, Table 1) track the predefined performance measures that are available, the other seven record values that are of specific interest in this model. Line 7 in Table 1 indicates that a total of 303 packets were recorded on place *OutNode*. Here it is interesting to recall how a timed statistical variable is updated. More than one step in a timed simulation can occur before the time advances, this means that only the last value inserted before the time changes will have any influence on the sum, sum of the squares, etc (as discussed in Sect.5). Thus, if at a given

TIMED STATISTICAL VARIABLES					
Observation function	#Updates	Sum	Average	Max	
1 Backlogged_nodes	2000	3379	3.39	4	
2 Backlogged_packets	1460	74836	74.84	142	
3 Buffer_length_node_1	303	17739	17.76	37	
4 Buffer_length_node_2	303	25795	25.82	42	
5 Buffer_length_node_3	303	15223	15.24	37	
6 Buffer_length_node_4	303	15533	15.55	32	
7 Number_of_Tokens_OutNode	4122	303	0.30	4	
8 Combined_List_Length_InRecvr	4122	303	0.30	4	
9 Average_List_Length_InRecvr	4122	303.00	0.30	4.00	

UNTIMED STATISTICAL VARIABLES					
Observation function	#Updates	Sum	Average	Max	
10 Packet_delay	109	30958	284.02	642	

Table 1. Performance Report

time there were one, two and finally three tokens on a place, only the value 3 will have influence on the sum, sum of the squares, etc. In this model, there are no tokens on *OutNode* for three out of four subslots (time units). Since the statistical variable which records the number of tokens on this place is updated after every step, many zeros will be added to the total sum of tokens. This accounts for the relatively low (0.30) average number of tokens on the place.

Two statistical variables were used in this model to collect data about the average list length and combined list length of lists on the place *InRecvr*. Lines 8 and 9 in Table 1 show the values accumulated in these variables. Since there is always exactly one list on this place, the values for combined list length and average list length are necessarily the same.

The seven other rows in Table 1 correspond to the model specific performance measures. The number of backlogged nodes in the system can be found in Line 1. On average there were 3.39 (out of a possible 4) backlogged nodes, which indicates that the parameters chosen in this model of the Aloha protocol do not produce a very effective means for allocating use of the communication channel.

Since there were so many backlogged nodes, there must have been backlogged packets in the system. The total number of backlogged packets at a given time is the cumulative number of packets in the buffers belonging to the nodes. One might think that it would be possible to track this value using the predefined performance measure *Combined_List_Length*. However, this is not possible because the lists corresponding to buffers are one element in a product (see Sect. 3.2), and *Combined_List_Length* can only be used when the tokens on a place are lists. The length of a single node's buffer can be found in Lines 3-6. It is not surprising that the average buffer lengths are quite high (between 15.24 and 25.82), knowing that the nodes were frequently backlogged and that new packets continued to arrive in each slot. The total number of backlogged packets was updated in the first subslot of each slot, while buffer lengths were updated in the second subslot of each slot. It is possible that a different number of simulation steps occur in different subslots (time units), and this accounts for the discrepancy between the number of updates for *Backlogged_packets* and *Buffer_length_node_x*.

The final performance measure of interest is the average delay for packets which were successfully transmitted. The delay was measured in time units. This value was recorded in an untimed statistical variable, and the results can be found in Line 10. The appropriate statistical variable was updated every time a packet arrived on the place *Received Packet* in Fig. 3.

7 Conclusion

The motivation for writing this paper came from our work with developing the performance facilities. Our original focus was on extending the untimed statistical variables to timed versions.

While working with the statistical variables, it seemed like a logical next step to provide a better, high-level interface to the statistical variables. Previously, it was frequently necessary to add code segments and/or extra places and transitions to a CPN model in order to extract the desired statistics. It is obviously time consuming to add these extra structures. Adding new structures can also lead to more error-prone CPN models. By using the performance facilities described in this paper, the means for collecting data is not directly present in the CPN model, i.e. very little, if anything needs to be added to the model. Thus, it is possible to keep the semantics of the model and the collection of data from the model separate. We think that this is clearly an improvement.

Once it was possible to easily create and update statistical variables independently of a CPN model, it seemed useful to have a reporting facility which could present the results in a simple report. Combining these capabilities with an existing Design/CPN library of distribution functions resulted in a performance library which a user could include and use when simulating models in Design/CPN. The interface to this library consisted solely of fairly low-level function calls. A graphical user interface to the new capabilities, in the form of menus and dialog boxes, would obviously provide a higher-level interface. This interface has not been completed, but it is in the process of being implemented. During development of the library, the advantages of integrating it into the Design/CPN tool became obvious, and the result is the new performance facilities in Design/CPN described in this paper.

Some improvements to the design of the performance facilities are interesting for further research. One of the areas is the predicate function. A possible improvement could be to give the user the ability to specify a list of transitions for each statistical variable. Then we could check if one of these transitions has occurred after each step. If this is the case, then we could update the statistical variable with the observed value, otherwise not. This would be another step towards providing a high-level interface to the performance facilities. The issue of performance analysis based on occurrences of transitions also deserves more attention in future work.

Another possible extension of the performance facilities is the ability to export the defined statistical functions and the corresponding predicate and observation functions to a file. The facility of being able to export these definitions in a library of statistical variables would be useful. In this way it would be possible to import and re-install the runtime table containing these statistical variables when restarting the simulator. This feature is also in the process of being implemented, but it too is not quite finished yet. One limitation to having a run-time table is that no places or transitions referred to in the predicate or observation functions may have been removed or renamed since the definitions were saved.

It is still possible to use the statistical variables independently of the performance facilities, but then the user is completely responsible for updating the statistical variables and extracting the appropriate values at the end of a simulation. The user is also required to open, close and update files if he is interested in maintaining a log file for each statistical variable.

It has been our experience that the performance facilities provide a user-friendly interface for collecting data, calculating statistics and reviewing the collected data. In conclusion, we do believe that these performance facilities can help doing performance analysis of CP-nets created in Design/CPN.

Acknowledgements. We would like to thank Lars Michael Kristensen for his help in writing this paper. We thank the CPN group at the University of Aarhus for comments both on the design of the performance facilities and on this paper.

References

- [AM91] Appel, A.W. and MacQueen, D.B. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Third International Symposium on Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
- [BG92] Bertsekas, D., Gallager, R.: *Data Networks*. Prentice Hall, 1992.
- [CJ97] Christensen, S. and Jørgensen, J.B. Analysing Bang & Olufsen's BeoLink® Audio/Video System Using Coloured Petri Nets. In P. Azema and G. Balbo, editors, *Proceedings of the 18th Inter-*

- national Conference on Application and Theory of Petri Nets, Toulouse, France*, volume 1248 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
- [CK97] Christensen, S. and Kristensen, L.M. State Space Analysis of Hierarchical Coloured Petri Nets. In: B. Farwer, D. Moldt and M-O. Stehr (Eds): *Proceedings of Workshop on Petri Nets in System Engineering (PNSE'97) Modelling, Verification, and Validation, Hamburg, Germany*, Publication No. 205, University Hamburg, Fachberich Informatik, pp. 32-43, 1997.
- [CJK97] Christensen, S., Jørgensen, J.B. and Kristensen, L.M. Design/CPN - A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Twente, The Netherlands* volume 1217 of *Lecture Notes in Computer Science* pages 209-223. Springer-Verlag, 1997. Also available as DAIMI PB-511, ISSN 0105-8517, February 1997.
- [Dri] Drimmelen, T. Implementation of Statistical Functions in Design/CPN. Online: <http://www.daimi.aau.dk/designCPN/libs/pdf/>
- [Gnuplot] Online: http://www.cs.dartmouth.edu/gnuplot_info.html
- [Jena] Jensen, K. et al *Design/CPN Online*, Department of Computer Science, University of Aarhus, Denmark. Online: <http://www.daimi.aau.dk/designCPN/>.
- [Jenb] Jensen, K. et al *Design/CPN Reference Manual*, Department of Computer Science, University of Aarhus, Denmark. Online: <http://www.daimi.aau.dk/designCPN/man/>
- [Jen92] Jensen, K. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
- [Jen94] Jensen, K. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1994.
- [Jen97] Jensen, K. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 3, Practical Use*. Monographs in Theoretical Computer Science, chapter 2, pages 21-37. Springer-Verlag, 1997.
- [LWa] Lindstrøm B., Wells, L. The Design/CPN Statistical Variable Library, Department of Computer Science, University of Aarhus, Denmark. Online: <http://www.daimi.aau.dk/designCPN/libs/>
- [LWb] Lindstrøm B., Wells, L. User Manual for the Performance Analysis Package, Department of Computer Science, University of Aarhus, Denmark. Online: <http://www.daimi.aau.dk/designCPN/>
- [Rip87] Ripley, B.D. *Stochastic Simulation*. Wiley series in probability and mathematical statistics. Applied probability and statistics, 1987.