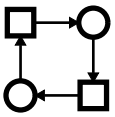

# DESIGN/CPN

CPN Group
Department of Computer Science
University of Aarhus
DENMARK

E-mail: designCPN-support@daimi.au.dk
WWW: www.daimi.au.dk/designCPN

# Design/CPN

## Performance Tool Manual

Version 1.0

Bo Lindstrøm and Lisa Wells

Design/CPN Manual Series September 1999

# Contents

# 1 Introduction

This manual describes the Design/CPN Performance Tool for facilitating simulation based performance analysis of Coloured Petri Nets. The performance tool is fully integrated in Design/CPN. It is assumed that the user is familiar with both Coloured Petri Nets (CP-nets or CPN) [6, 7] and the Design/CPN tool [2, 3, 8].

In this context, performance analysis is based on analysis of data extracted from a CPN model during simulation. The performance tool provides random number generators for a variety of probability distributions and high-level support for both data collection and for generating simulation output. The random number generators can be used to create more accurate models by modelling certain probability distribution aspects of a system, while the data collection facilities can extract relevant data from a CPN model. Note that the tool does not undertake any performance analysis, rather it provides a means for collecting data which then can be analysed by the user.

Section 1.1 gives a general description of the performance tool. This includes motivation for developing the tool, a description of the functionality of the tool, and a description of the integration of the tool in Design/CPN. Section 1.2 introduces a small CPN model which will be used to illustrate the use of the performance tool. Finally, Sect. 1.3 will give a brief overview of the output that the performance tool provides.

## 1.1 Overview of the Design/CPN Performance Tool

Most applications of CP-nets are used to investigate the logical correctness of a system. This means that focus is on 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 one often uses timed CP-nets[1].

While the Design/CPN tool [2] supports state space analysis [1, 3], timed simulations and functional analysis, it lacked integrated high-level 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 variables[2] and the use of code segments. An untimed statistical variable is a data type with which it is possible to store collected values and later to extract different statistical information such as sum or average [3]. This manual describes a performance tool which remedies the above shortcomings of Design/CPN.

During simulation of a CPN model, one can be interested in evaluating the performance of the system. To do this it is necessary to extract different values from the markings or binding elements encountered during simulation of the CPN model. A *data collector* is a central concept in the performance tool. A data collector determines when and how data is extracted from the CPN model and how to deal with this data. A data collector is defined in Sect. 6.1.

The performance tool provides an easy way to refer to the values from markings and binding elements to be examined during a simulation. The user must write his own functions for extracting a value from a marking and a binding element of a CP-net. The user is not required to recall the exact syntax for the functions because it is possible to generate templates for the necessary functions. Data collection occurs during simulation, and the extracted value is then added to the appropriate data collector. Other facilities such as creating and initialising a data collector and producing a performance

[1] See Chap. 5 in [7] for a definition of timed CP-nets

[2] Originally called statistical variables.

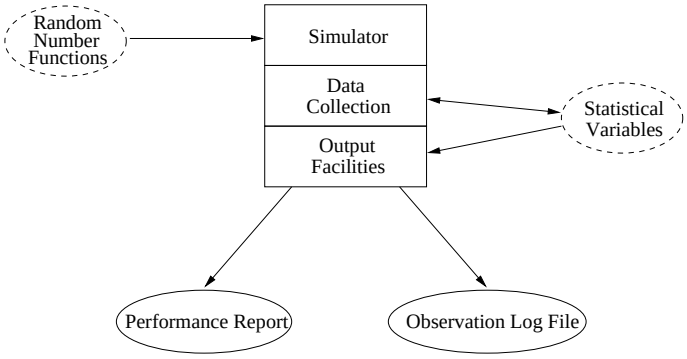


Figure 1: Overview of the tool.

report during simulation are also available.

Figure 1 illustrates how the performance tool is integrated in Design/CPN. The items (except the item *Simulator*) illustrate the components of the new performance tool within Design/CPN. 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 tool.

The execution of the model in Design/CPN using the performance tool is as follows: the *Simulator* simulates the CPN model possibly using the *Random Number 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 *Output Facilities* dump all of the observed values in a detailed *Observation Log File*. At any stop point in the simulation, the user can also generate a *Performance Report* using the *Output 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 provided by an *Observation Log File*.

## 1.2 Example: Ferris Wheel

In this manual a CPN model of a Ferris wheel is used to illustrate the use of the performance tool. The Ferris wheel is an amusement park ride. In this system, the Ferris wheel has a capacity of four customers. When new customers arrive they receive a numbered ticket, and they join a queue where they wait until it is their turn to ride the Ferris wheel. The inter-arrival time for customers is exponentially distributed. Figure 2 depicts the system to be modelled.

What kind of performance measures are interesting for this system? One obvious choice would be to consider the length of the queue. How does it grow and shrink? Another interesting performance measure is the average amount of time each customer waits in the queue.

The system is modelled by the CPN model shown in Figs. 3 and 4. The global declaration node for the model can be seen in Fig. 3. The value `WheelSize` indicates the capacity of the Ferris wheel; in this case it is 4. The variable `interArrival` is used to model the time between customer arrivals. Note that `interArrival` is a reference variable. A reference variable is used to make it possible to change the value of the variable without having to switch between the editor and the simulator or the performance tool in Design/CPN. The colour set `Int` is used to model the time used for various activities. The colour set `IntT` is a timed colour set which is used to count customers. The colour set

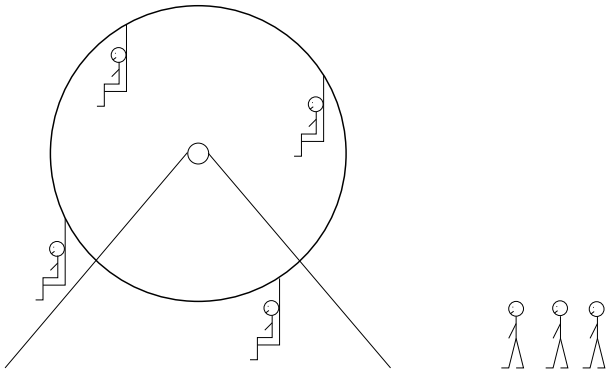

Figure 2: Ferris wheel.

`Ticket` models a ticket. A ticket consists of a ticket number and a time stamp which indicates when the ticket was given to a customer. The colour set `Customer` models the customers. Note that the colour set `E` is also a timed colour set. The functions `discExp` and `discNorm` are used to generate random numbers from discrete exponential and normal distributions.

```
(* Capacity of the wheel *)
val WheelSize = 4;

(* Average inter-arrival time, in seconds*)
val interArrival = ref 90.0;

(* — Colour sets — *)
color Int = int;
color IntT = int timed;
color Ticket = product Int * Int;
color Customer = with custom;
color CustomerxTicket = product Customer * Ticket;
color E = with e timed;

(* — Variables — *)
var nextCust,arrivalTime,runTime,loadTime,n,m : Int;
var noOfCust : IntT;

(* — Functions — *)
fun intTime() = IntInf.toInt(time());
fun round x = floor(x+0.5);
fun discNorm (x,y) = round(normal (x,y));
fun discExp x = round(exponential(x));
```

Figure 3: Global declarations.

Figure 4 contains the complete net structure of the model. The name of the page containing the net structure is *FerrisWheel*. The left-hand side of the net models the arrival of customers, while the rest of the net models the Ferris wheel. When the transition *Customer Arrives* occurs, a new customer token is added to the place *Waiting Customers*. Upon arrival, each customer is paired with a numbered ticket which contains a time stamp. The time stamp is equal to the current model time, and it is created

by means of the function `intTime`, on the arc between *Customer Arrives* and *Waiting Customers.* The place *Waiting Customers* represents the queue of customers. Customers are loaded for the next run of the Ferris wheel when the transition *LoadNext Customer* occurs. The arcs between *Next Ticket* and *LoadNext Customer* ensure that customers are loaded into the Ferris wheel in the same order in which they joined the queue.

When customers are loaded onto the wheel, they are counted (see the inscriptions on the arcs between *LoadNext Customer* and *Loaded Customers*). The guard of the transition *Start Wheel* ensures that the Ferris wheel is not started until either there are no more customers in the queue or the Ferris wheel is filled to capacity. Once the Ferris wheel is started, it is running – modelled by the place *Running.* After the wheel stops (*Stop Wheel* occurs), then new customers can be loaded, and the wheel can be started again.

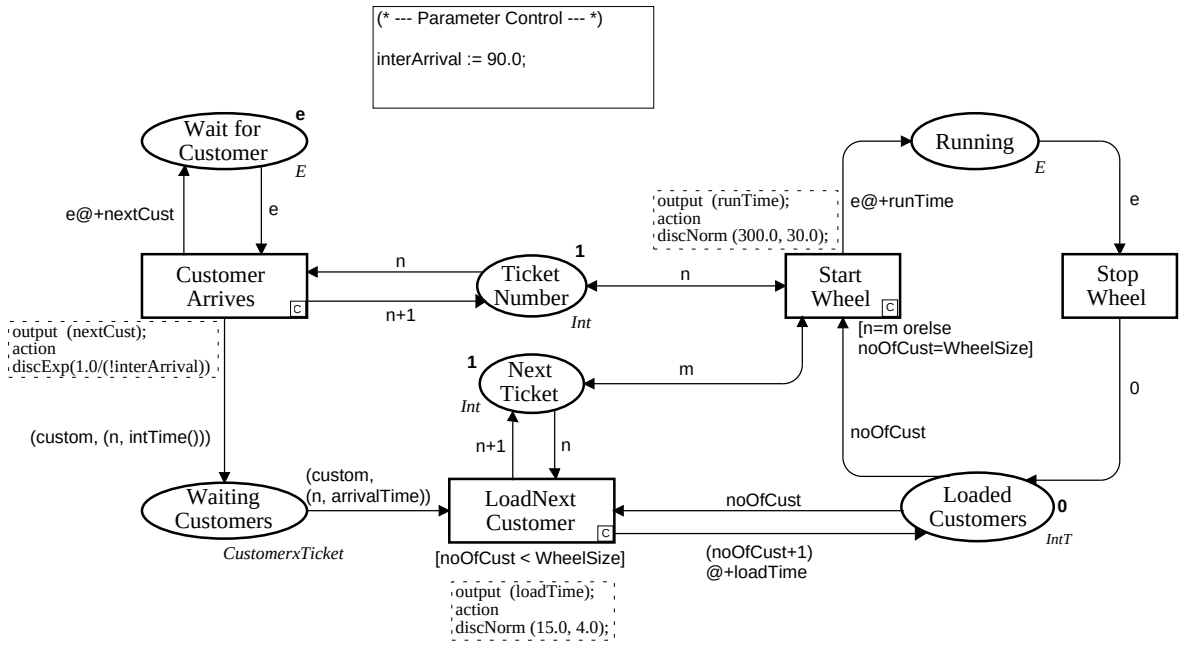


Figure 4: Ferris wheel net.

The model is an integer-timed CPN model, and time is measured in seconds. The `discExp` function is used to model the inter-arrival time for customers, and it can be found in the code segment associated with *Customer Arrives.* The inter-arrival time for customers is exponentially distributed, which is modelled using the `exponential` random number function, and the mean inter-arrival time is `interArrival` seconds. The value pointed to by the variable `interArrival` is used as the parameter for the `discExp` function. By changing the value and reevaluating the auxiliary node (the box with the text `Parameter Control`), one can change the inter-arrival rate for customers (`interArrival`) without having to switch to the editor.[3] The `discNorm` function in the code segment for *LoadNext Customer* calculates how much time is needed for loading each customer. The average time needed for loading a customer is 15 seconds while the variance is 4 seconds. A similar approach is used to calculate how much time is used for running the Ferris wheel – as shown in the code segment for *Start Wheel.*

[3]This is particularly useful when one wishes to make a number of simulations using different parameters.

## 1.3 Performance of Ferris Wheel Model

This section will give a brief introduction to the output that can be created by the performance tool. Data was collected during two simulations of the Ferris wheel model. This data will be shown in two different forms, and the data will be briefly analysed.

In the previous section two performance measures for the system were mentioned: queue length and average waiting time. The necessary functions were defined for collecting data concerning these measures. These functions will be discussed throughout the manual and can be found in Appendix B. The model was then simulated. Three simulations were made to see how changing the parameter `interArrival` affects the performance of the system. The parameter `interArrival` affects how often a new customer arrives at the Ferris wheel. Three simulations were made: one with `interArrival=60.0`, one with `interArrival=90.0`, and one with `interArrival=120.0`.

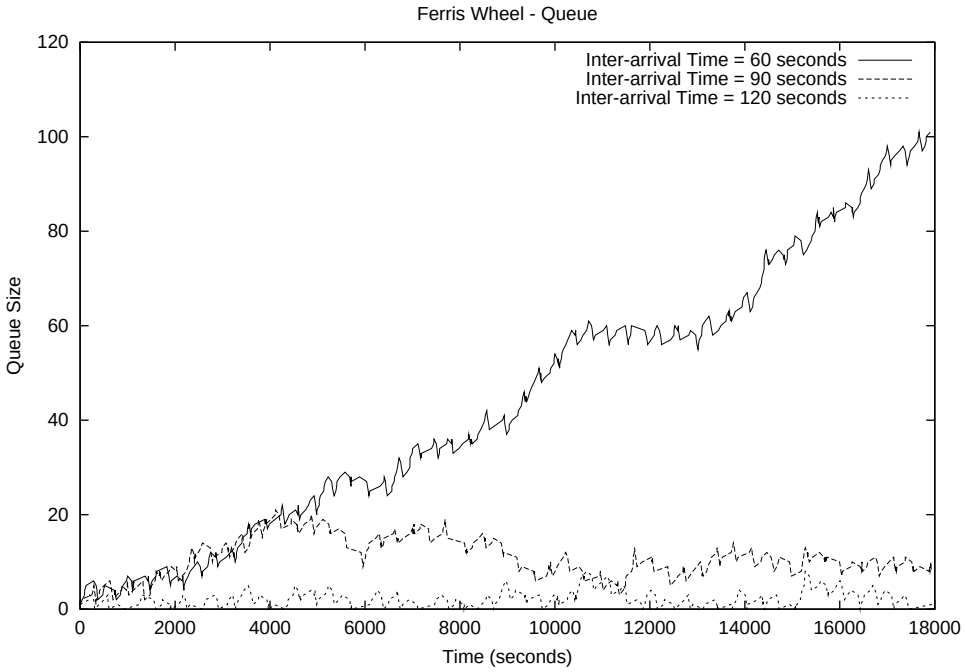


Figure 5: Customer queue for different arrival rates.

Figure 5 shows a graph which indicates how changing the inter-arrival time between customers will effect the queue length. When `interArrival` is 120.0 seconds the size of the queue is never larger than eight. Reducing `interArrival` to 90.0 seconds produces a queue size that fluctuates, but does not continually grow. However, when `interArrival=60.0` the queue grows constantly. This graph was plotted using data from *observation log files* which were created using the output facilities of the performance tool. Section 7.2 describes observation log files. The graph was made using the **gnuplot** program [5]. The script that was used to generate the graph can be found in Appendix C.

Examples of the other type of output that can be generated in the performance tool can be seen in Figs. 6 and 7. These *performance reports* contain statistics about the data that has been collected concerning queue length and average delay. This type of report gives a high-level view of the data since

that can be calculated are: first and last value observed, variance,

statistics about the data rather than a *list* of each value that was observed. In addition to the statistics shown, the other

standard deviation, sum of squares, and sum of squares of deviation. In these reports one can see that when `interArrival=120.0`, the average waiting time for each customer was 3.232 minutes[4], compared to 44.526 minutes when `interArrival=60.0`. Similarly for the average length of the queue. Performance reports are described in detail in Sect. 7.1.

TIMED STATISTICS

| Name | Count | Sum | Average | Minimum | Maximum |
|---|---|---|---|---|---|
| CustQueue | 311 | 34944 | 1.932 | 0 | 8 |

UNTIMED STATISTICS

| Name | Count | Sum | Average | Minimum | Maximum |
|---|---|---|---|---|---|
| CustWait | 155 | 501 | 3.232 | 0 | 10 |

Current step: 418
Current time: 18085

Figure 6: Performance report, `interArrival` = 120.0.

TIMED STATISTICS

| Name | Count | Sum | Average | Minimum | Maximum |
|---|---|---|---|---|---|
| CustQueue | 493 | 800337 | 44.355 | 1 | 101 |

UNTIMED STATISTICS

| Name | Count | Sum | Average | Minimum | Maximum |
|---|---|---|---|---|---|
| CustWait | 196 | 8727 | 44.526 | 4 | 92 |

Current step: 592
Current time: 18044

Figure 7: Performance report, `interArrival` = 60.0.

[4]See Appendix B for an explanation of how customer wait time is measured in minutes rather than seconds.

# 2 Simulation with the Design/CPN Performance Tool

This section will give a general introduction to simulating CPN models using the performance tool. Section 2.1 describes how to generate the necessary ML code for the performance tool. Section 2.2 introduces the performance page and node. Finally, Sect. 2.3 describes how to simulate models and collect data using the performance tool.

## 2.1 Generation of Performance Code

Before data can be collected from a CPN model, it is necessary to generate the *performance code*, i.e. the ML code which is used to extract data from the CPN model. The performance code is generated in a way which is similar to the switch from the editor to the simulator.

To use the performance tool the following steps must be performed (in the specified order) in either the editor or the simulator:

1. Make sure that you are using Design/CPN version 3.2 (or later) and the corresponding ML image.

2. Use **General Simulation Options** to indicate whether you want your simulation to be with or without time. To choose the settings that you want, it may be first necessary to use **Simulation Code Options**.

3. Check the box **OG Tool Violations** in **Syntax Options**.

4. Use **Enter Perf** (in the **File** menu) to enter the performance tool. This will create the performance code and may take some time. When **Enter Perf** terminates, the **Perf** menu (Fig. 8) is added to the menu bar (at the rightmost end). This menu contains all of the commands which are specific for the performance tool.

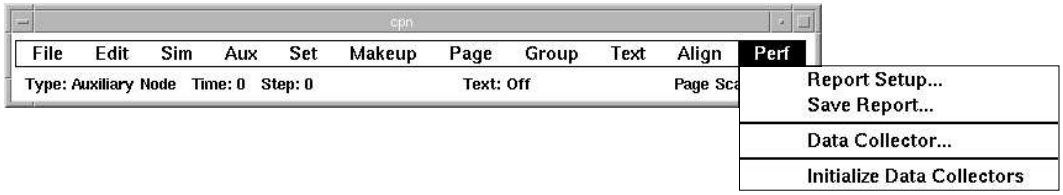


Figure 8: Perf menu.

Each of the menu items in the **Perf** menu will be discussed in detail in later sections. **Report Setup** is described in Sect. 7.1.1, and **Save Report** follows in Sect. 7.1.2. The items **Data Collector** and **Initialize Data Collectors** are introduced in Sects. 6.3 and 6.4, respectively.

**Note:** Invoking **Enter Perf** from the editor is a shortcut for switching to the simulator and then switching to the performance tool.

## 2.2 Performance Page and Performance Node

In order to collect data in the performance tool, a number of functions (from now on referred to as *performance functions*) must be provided by the user. These functions are described in Sects. 6.2.1 – 6.2.3. The performance functions cannot be defined in the global declaration node because they are dependent on the performance code that is generated during the switch to the performance tool. As a result, two new objects have been defined in Design/CPN: the *performance page* and the *performance node*.

**Performance Page**
The performance page is simply a page that has the name **Performance_Page**. When the switch is made to the performance tool, such a page must exist. If it does not exist, a new one will be created automatically. This page may contain at most one auxiliary node. If no auxiliary node is found on the page when a switch is made, then a new one must be created. If more than one node exist, then the user will be warned and will then have to delete all but one of the auxiliary nodes and reinvoke **Enter Perf**.

**Performance Node**
The one auxiliary node that can be on the performance page will be referred to as the performance node. The user must define the necessary performance functions in the performance node in order to ensure that data will be collected as desired.

When the switch is made to the performance tool, the performance node is automatically evaluated (equivalent to choosing **ML Evaluate** in the **Aux** menu). After the performance node has been evaluated, the data collectors[5] that have been defined in the performance node are installed and are ready to extract data from the model during simulation. Invoking **Initial State** in the **Sim** menu will delete any existing data collectors and reevaluate the performance node, in addition to returning to the initial state. Reevaluating the performance node simply reinstalls all the data collectors that are defined in the performance node. If the performance node is evaluated by invoking **ML Evaluate** (instead of using **Initial State**), then old data collectors will not be deleted, and therefore, multiple copies of a data collector may exist.

**Important:** No syntax check is made for the performance node. The user is responsible for checking if any error has occurred by checking the auxiliary region that appears next to the performance node when **ML Evaluate** is invoked. Additionally, the switch to the performance tool is made even if an error is found in the performance node.

## 2.3 Simulation and Data Collection

After the switch has been made to the performance tool, the model is ready for simulation. In order to collect data, an **Automatic Run** (in the **Sim** menu) must be started. Data will be collected and stored during the simulation. At any stop point in the simulation, it is possible to create a performance report (see Sect. 7).

**Details and Limitations**

[5] Data collectors will be defined in Sect. 6.

When you modify a CPN diagram, it is necessary to regenerate the performance code. When a modification is made while being in the simulator, it is sufficient to invoke **Reswitch** and then **Enter Perf**. This means that all data collectors will be deleted, and the performance node will be evaluated again to reinstall the data collectors.

## 2.4 Limitations in the Current Version

In the current version of the performance tool it is only possible to collect data during an **Automatic Run**, i.e. data will not be collected during an **Interactive Run**.

# 3 Random Number Functions

Random number functions allow the user to draw random samples from different distributions. By using such functions it is easier to construct a more precise model of some physical phenomenon than having to model the distribution explicitly in the CPN model. This in turn can lead to better results when simulating and analysing the performance of the model. The random number functions can also be used in the standard simulator, i.e. one is not limited to using these functions only in the performance tool.

The following subsections present each of the available random number functions. The interface for each function is presented along with a short description and an example of practical use. Finally, the following are also shown for each function: mean value, variance, and the density function which indicates how likely a value is to be drawn from the corresponding distribution. Further details on the implementation of the random number functions and the underlying pseudo random number generator can be found in [4].

## 3.1 Bernoulli

Interface: `fun bernoulli (p:real) : int`

Where $0 \leq p \leq 1$. The value returned is either 0 or 1. The function gives a drawing from a Bernoulli distribution with a probability of $p$ for success, i.e. success $= 1$.

> **Example 1** *Throw a die and observe if a six was thrown. This experiment has a Bernoulli distribution with parameter $p = \frac{1}{6}$ for success.*

Mean: $p$
Variance: $p(1-p)$

The density function, $f_X(x)$, is given by:

$$f_X(x) = \begin{cases} 1-p & \text{x=0} \\ p & \text{x=1} \\ 0 & \text{elsewhere} \end{cases}$$

### 3.2 Binomial

Interface: `fun binomial (n:int, p:real) : int`

Where $n \geq 1$ and $0 \leq p \leq 1$. Returns a drawing from a binomial distribution with $n$ experiments and probability $p$ for success.

The binomial distribution is related to the Bernoulli distribution in the following way: the sum of $n$ Bernoulli drawings with parameter $p$ has a binomial distribution with parameters $p$ and $n$.

> **Example 2** *Throw a die* 100 *times and observe how many times a six was thrown. This process has a binomial distribution with parameters* $n = 100$ *and* $p = \frac{1}{6}$.

Mean: $np$
Variance: $np(1-p)$

The density function, $f_X(x)$, shown in Fig. 9 is given by:

$$f_X(x) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x} & \text{x=0, 1, 2, ..., n} \\ 0 & \text{elsewhere} \end{cases}$$

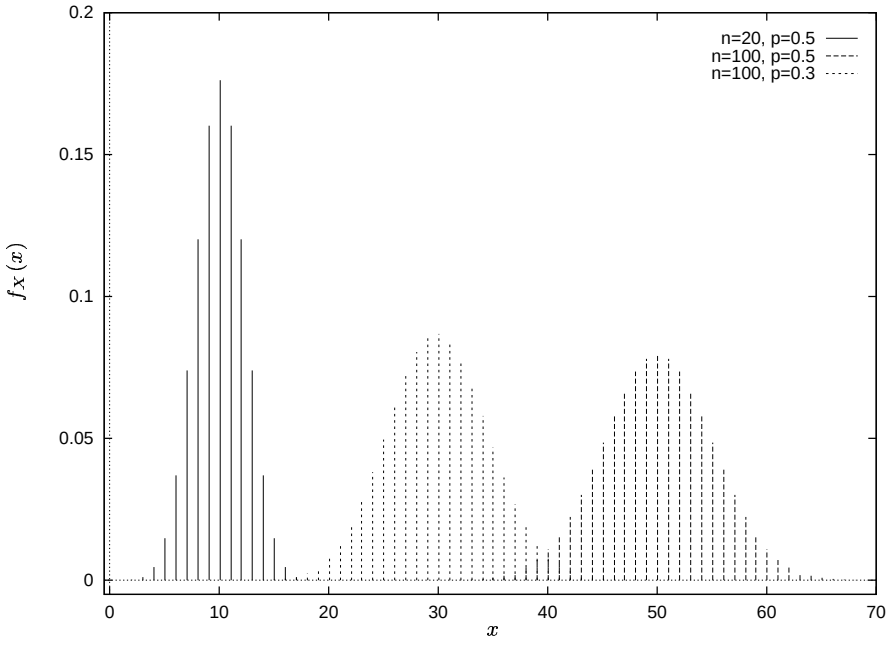


Figure 9: Binomial densities.

### 3.3 Chi-square

Interface: `fun chisq (n:int) : real`

Where $n \geq 1$. Returns a drawing from a chi-square distribution with $n$ degrees of freedom.

The sum of the squares of $n$ independent normally distributed random variables with mean $0$ and standard deviation $1$ is a chi-squared distribution with $n$ degrees of freedom.

> **Example 3** *An example for this distribution has been omitted due to the fact that it is rarely found in nature. Instead, the distribution is used when doing statistical tests.*

Mean: $n$
Variance: $2n$

The density function, $f_X(x)$, shown in Fig. 10 is given by:

$$f_X(x) = \begin{cases} \frac{1}{2^{n/2}\Gamma(n/2)} x^{(n/2)-1} e^{-x/2} & \text{x} > 0 \\ 0 & \text{elsewhere} \end{cases}$$

where $\Gamma(\alpha) = \int_0^\infty x^{\alpha-1} e^{-x} dx, \ \alpha > 0$

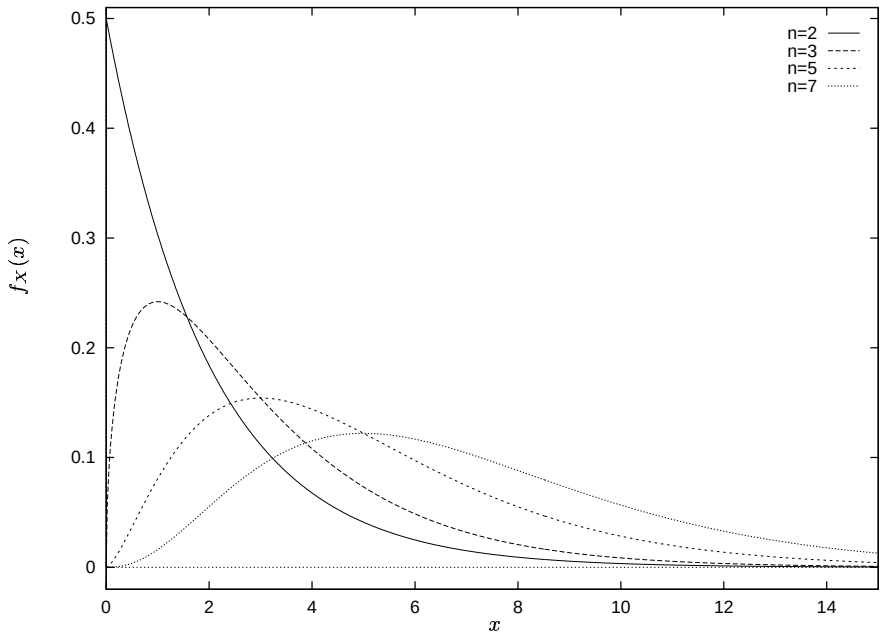


Figure 10: Chi-square densities.

### 3.4 Discrete Uniform

Interface: `fun rint (a:int, b:int) : int`

Where $a \leq b$. Returns a drawing from a discrete uniform distribution between $a$ and $b$ ($a$ and $b$ included).

> **Example 4** *Throwing a die has a discrete uniform distribution with parameters $a = 1$ and $b = 6$. The function $rint$ will then return the number of eyes on the die.*

Mean: $\frac{a+b}{2}$

Variance: $\frac{(b-a+1)^2-1}{12}$

The density function, $f_X(x)$, shown in Fig. 11 is given by:

$$f_X(x) = \begin{cases} \frac{1}{b-a+1} & \text{x = a, a+1, ..., b} \\ 0 & \text{elsewhere} \end{cases}$$

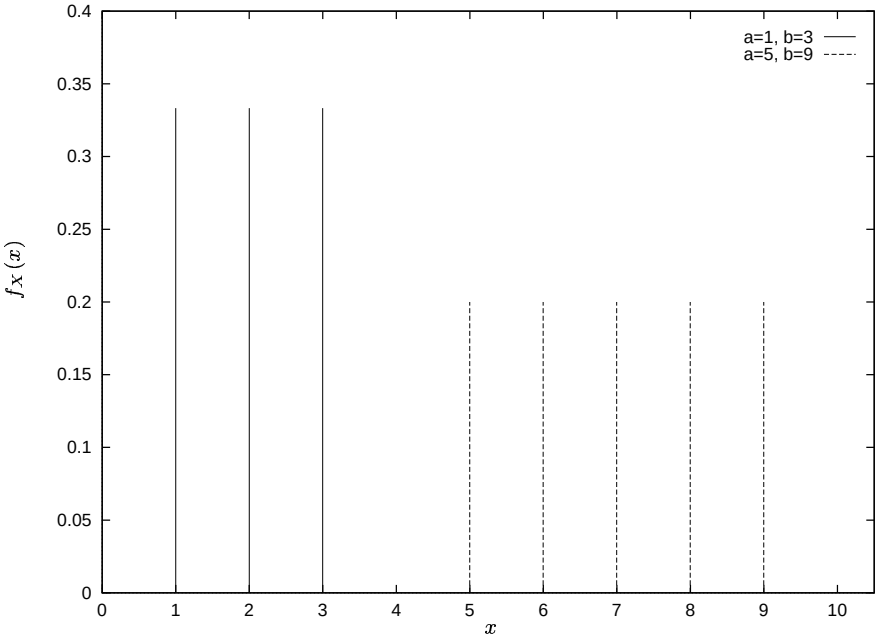


Figure 11: Discrete uniform densities.

### 3.5 Erlang

Interface: `fun erlang (n:int, l:real) : int`

Where $n \geq 1$ and $l > 0$. Returns a drawing from an Erlang$_n(l)$ distribution with intensity $l$.

A drawing from an $Erlang$ distribution can be derived by addition of $n$ drawings from a exponential distribution.

> **Example 5** *A shop gives each* $100$*th customer a present. The arrival time between customers is exponentially distributed with intensity* $l = 50$ *per hour. The sums of the arrival times between customers is Erlang distributed with parameters* $n = 100$ *and* $l = 50$.

Mean: $\frac{n}{l}$
Variance: $\frac{n}{l^2}$

The density function, $f_X(x)$, shown in Fig. 12 is given by:

$$f_X(x) = \begin{cases} \frac{l^n}{(n-1)!} x^{n-1} e^{-xl} & x > 0 \text{ and n=1, 2, 3, ...} \\ 0 & \text{elsewhere} \end{cases}$$

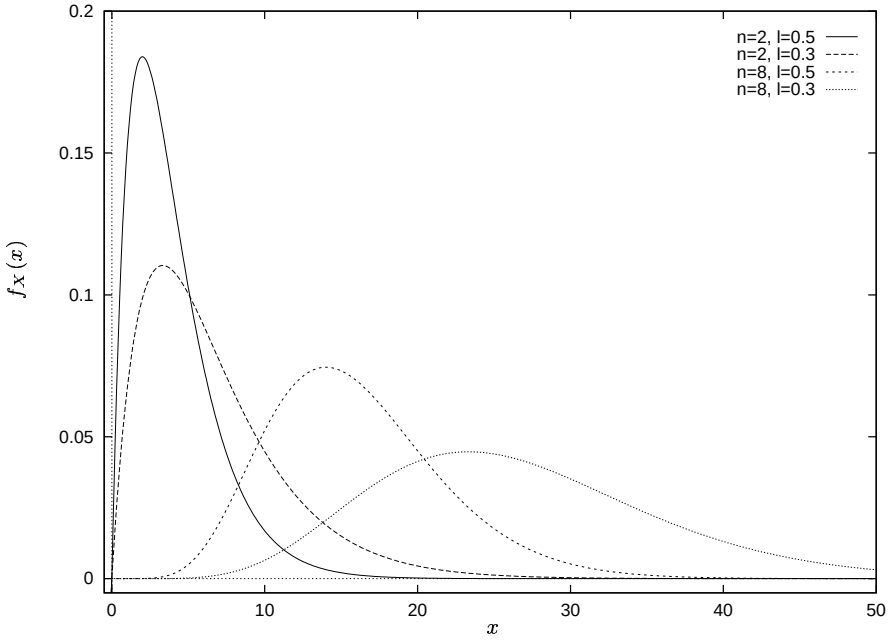


Figure 12: Erlang densities.

### 3.6 Exponential

Interface: `fun exponential (r:real) : real`

Where $r > 0$. Gives a drawing from a exponential distribution with intensity $r$.

> **Example 6** *Customers arrive at a post office for service. The time between two arrivals has a mean of 4 minutes. The inter-arrival time has a exponential distribution with parameter* $r = \frac{1}{4}$.

Mean: $\frac{1}{r}$
Variance: $\frac{1}{r^2}$

The density function, $f_X(x)$, shown in Fig. 13 is given by:

$$f_X(x) = \begin{cases} re^{-rx} & \text{x} \geq 0 \\ 0 & \text{elsewhere} \end{cases}$$

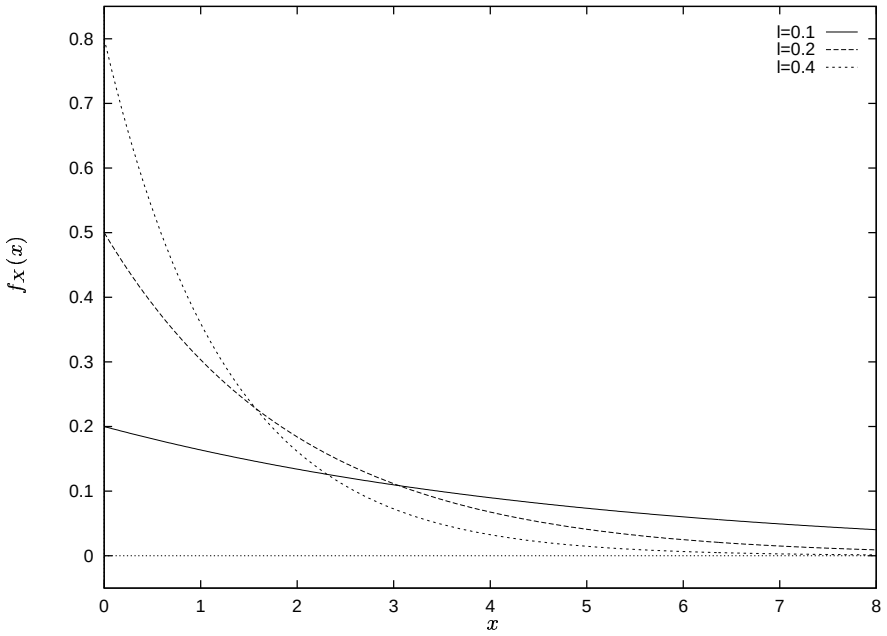


Figure 13: Exponential densities.

### 3.7 Normal

Interface: `fun normal (n:real, s:real) : real`

Returns a drawing from a normal distribution with mean $n$ and variance $s$.

> **Example 7** *A factory produces chocolate in packages of* $500$ *grams. The amount of chocolate in each package has a normal distribution with mean* $n = 505$ *grams and variance* $s = 3$ *grams. Each package with a weight less than 500 grams will be rejected in the control procedure.*

Mean: $n$
Variance: $s$

The density function, $f_X(x)$, shown in Fig. 14 is given by:

$$f_X(x) = \frac{1}{\sqrt{2s\pi}} e^{-\frac{(x-n)^2}{2s}}$$

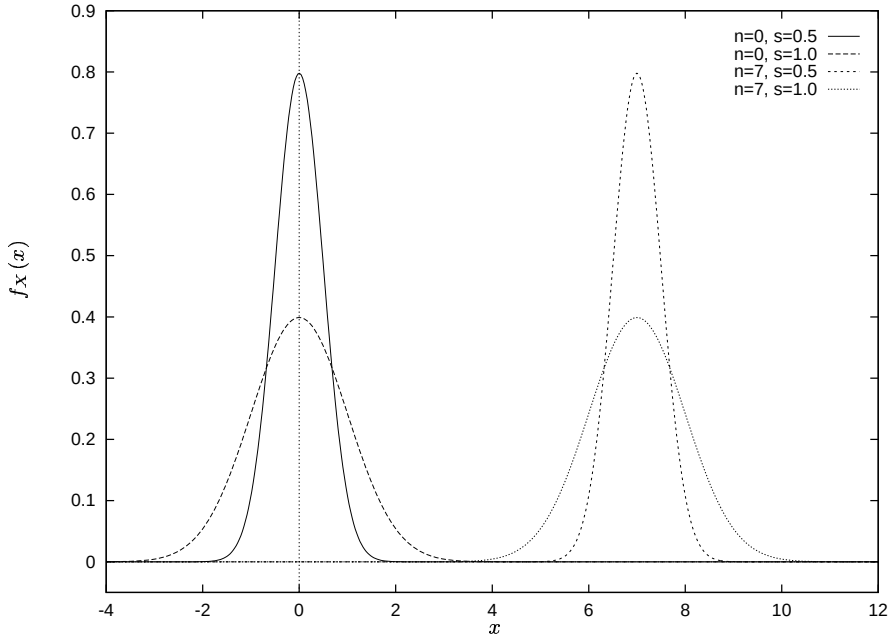


Figure 14: Normal densities.

### 3.8 Poisson

Interface: `fun poisson (m:real) : int`

Where $m > 0$. Returns a drawing from a Poisson distribution with intensity $m$.

> **Example 8** *A company has a network with a certain load. Each second an average of* 100 *packets is sent to the network. The number of packets arriving to the network per second is Poisson distributed with an intensity* $m = 100$.

Mean: $m$
Variance: $m$

The density function, $f_X(x)$, shown in Fig. 15 is given by:

$$f_X(x) = \begin{cases} \frac{m^x}{x!}e^{-m} & \text{x = 0, 1, 2, ..., n} \\ 0 & \text{elsewhere} \end{cases}$$

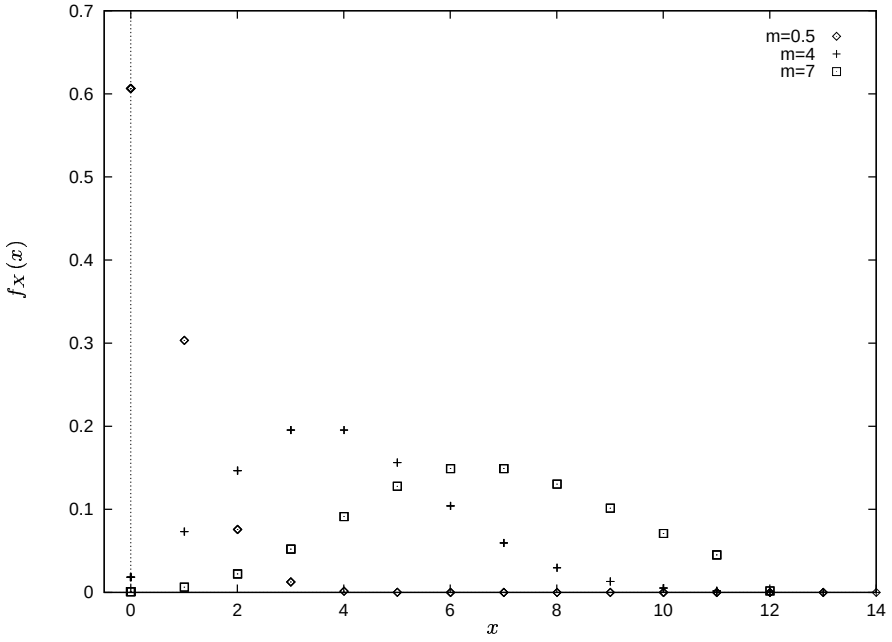


Figure 15: Poisson densities.

### 3.9 Student

Interface: `fun student (n:int) : real`

Where $n \geq 1$. Returns a drawing from a Student distribution (also called t distribution) with $n$ degrees of freedom.

Note that as $n$ increases, the Student density approaches the normal density in Sect. 3.7. Indeed, even for $n = 8$ the Student density is almost the same as the normal density.

> **Example 9** *An example for this distribution has been omitted due to the fact that it is rarely found in nature. Instead, the distribution is used when doing statistical tests.*

Mean: $0$
Variance: $\frac{1}{n-2}$

The density function, $f_X(x)$, shown in Fig. 16 is given by:

$$f_X(x) = \frac{\Gamma[(n+1)/2]}{\sqrt{n\pi}\,\Gamma(n/2)} \left(1 + \frac{x^2}{n}\right)^{-\frac{(n+1)}{2}}$$

where $\Gamma(\alpha) = \int_0^\infty x^{\alpha-1}e^{-x}dx, \alpha > 0$.

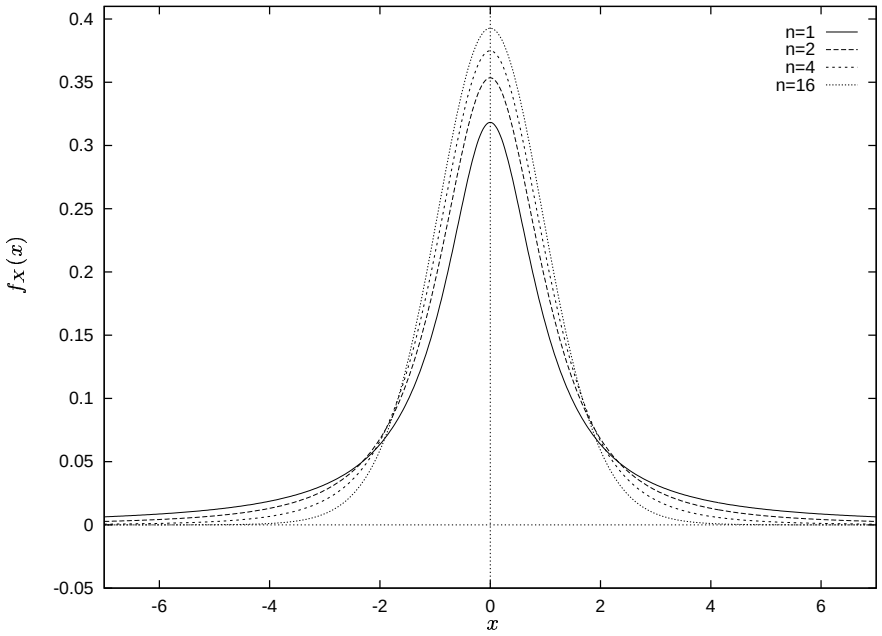


Figure 16: Student densities.

### 3.10 Continous Uniform

Interface: `fun uniform (a:real, b:real) :  real`

Where $a \leq b$. Returns a drawing from a uniform distribution between $a$ and $b$.

> **Example 10** *A person is asked to choose a real number between* 1 *and* 10. *This random variable is uniformly distributed with parameters* $a = 1$ *and* $b = 10$.

Mean: $\frac{a+b}{2}$

Variance: $\frac{(b-a)^2}{12}$

The density function, $f_X(x)$, shown in Fig. 17 is given by:

$$f_X(x) = \begin{cases} \frac{1}{b-a} & \text{a < x < b} \\ 0 & \text{elsewhere} \end{cases}$$

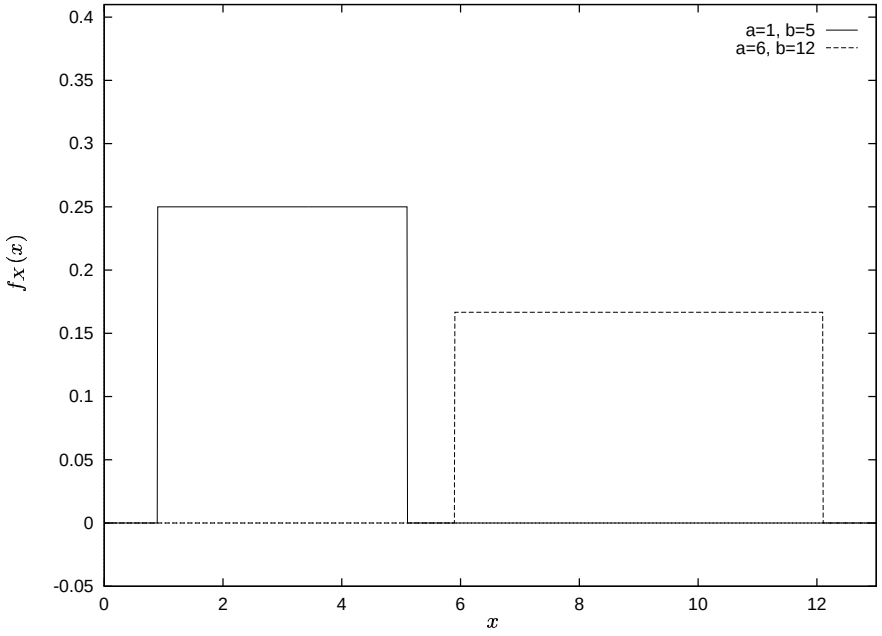


Figure 17: Uniform densities.

# 4 Binding Elements and Markings

In the performance tool it is possible to refer to each state of a model during simulation, i.e. one can refer to each encountered *marking* during simulation. It is also possible to inspect how a state was reached from the previous state, i.e. one can inspect each occurring *binding element* during simulation. Thus, values can be extracted from the markings and binding elements for updating the data collectors. Two different ML structures are available: the structure `PerfMark` containing a marking function for each place, and the structure `Bind` containing a binding constructor for each transition in the CP-net. These functions and constructors can be used to determine when to extract data and which data to extract. The `PerfMark` and `Bind` structures are analogous to the `Mark` and `Bind` structures which are described in the *Design/CPN Occurrence Graph Manual* [8].

## 4.1 Binding Elements

To denote binding elements the following constructors are available:

```
con Bind.<PageName>'<TransName>:  Inst * record -> Bind.Elem
```

where the second argument is a record specifying the binding of the variables of the transition. The type of this argument depends upon the transition.

For example in the Ferris wheel model:

```
Bind.FerrisWheel'Customer(1, {nextCust=75,n=x})
```

matches the binding element where the transition *Customer Arrives* (with the name *Customer*) on the *first* instance of page *FerrisWheel* has the variable `noOfCust` bound to 75. The variable `x` is included to be able to inspect the value bound to the variable `n`.

It should be noted that `Bind.<PageName>'<TransName>` is a constructor. This means that it can be used in pattern matching. For example:

```
Bind.FerrisWheel'Customer _
```

will match with every occurrence of each instance of the transition *Customer Arrives* (with name *Customer*).

```
Bind.FerrisWheel'Customer(1,{nextCust=75,...})
```

will match with each occurrence of the *first* instance of the transition *Customer Arrives* where the variable `nextCust` is bound to the value 75 and regardless of what any other variables might be bound to.

## 4.2 Markings

**Marking Function**
To inspect the markings of different place instances the following functions are available:

```
fun PerfMark.<PageName>'<PlaceName>: Inst -> (CPN'OGrec → CS ms)
```

where CS is the colour set of the place.

For example in the Ferris wheel model:

```
PerfMark.FerrisWheel'Waiting 1 net_marking
```

returns the multi-set of tokens on the place *Waiting Customers* (with the name *Waiting*) on the *first* instance of the page *FerrisWheel*. The variable `net_marking` refers to the current marking of the entire net.

**Multi-set Representation**
The `PerfMark` functions return the *internal* ML representation of a multi-set. The internal representation of a multi-set differs from the *formal sum* representation[6] which can be seen in connection with tokens during simulations.

The function `ms_to_list` can be used to convert either multi-set representation into a list representation.

```
fun ms_to_list: CS ms -> CS list
```

The function `striptime` can be used to remove the time stamps from a timed multi-set.

```
fun striptime: CS tms -> CS ms
```

The following list shows the different representations of a simple timed multi-set. Item 3 is the result of applying `striptime` to either of the multi-set representations in Items 1 and 2. Item 4 is the result of applying `ms_to_list` to either Item 3 or the formal sum representation of the same multi-set.

1. **Formal sum:** $2`\texttt{true} \,@\, [1,3] + 1`\texttt{false} \,@\, [1]$
2. **Internal:** (2,`true`,[`BI`{`digits=[1],sign=POS`}, `BI`{`digits=[3],sign=POS`}])!!! (1,`false`,[`BI`{`digits=[1],sign=POS`}])!!! `tempty`
3. **Stripped time:** (2, `true`)!! (1, `false`)!! `empty`
4. **List:** [`true`,`true`,`false`]

Chapter 37 of the *Design/CPN Reference Manual* [3] contains further details about:

- `ms_to_list`

[6] See Definition 2.1 in [6]

- the internal ML representation for multi-sets
- untimed and timed multi-sets
- the operators !! and !!!
- the constants `empty` and `tempty`

# 5 Statistical Variables

This section describes the concept of statistical variables. Statistical variables are data structures providing the ability both to accumulate 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 values and statistics that can be accessed in both types of statistical variables are:

- first value
- last value (most recently added value)
- minimum
- maximum
- count (number of updates)
- sum
- sum of squares
- average
- sum of squares of deviation
- standard deviation
- variance

Timed statistical variables include additionally:

- time of first update
- time of last update
- time interval (indicates how much time has elapsed since the statistical variable was first updated)

## 5.1 Untimed Statistical Variables

Figure 18 shows how an untimed statistical variable is updated with different observed values. If an untimed statistical variable is updated with the the same value twice, then the value influences the statistics twice, as expected. Let $x_i,\ i = 1..n$ be the values with which an untimed statistical variable is updated. The sum and average of the values are calculated in the following way:

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

The remaining statistics are calculated in a similar fashion.

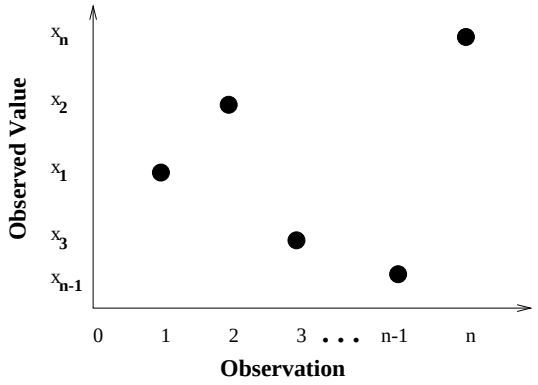


Figure 18: Observed values for untimed statistical variables.

## 5.2 Timed Statistical Variables

Timed statistical variables differ from untimed statistical variables in that an interval of time is used to weight each observed value. Figure 19 illustrates how a timed statistical variable is updated. 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 $x_n$.

At time $t_i$, the timed statistical variable is updated with a new value $x_i$, then at time $t_{i+1}$ it is updated with the value $x_{i+1}$. The interval $[t_i, t_{i+1}]$ is used to weight the value $x_i$, in other words, the weight of the value $x_i$ is $(t_{i+1} - t_i)$. At precisely time $t_i$, $x_i$ has no influence on sum, sum of the squares, average, sum of the squares of deviation, standard deviation or variance because the weight is zero, but for all time $t > t_i$, $x_i$ will influence these values.

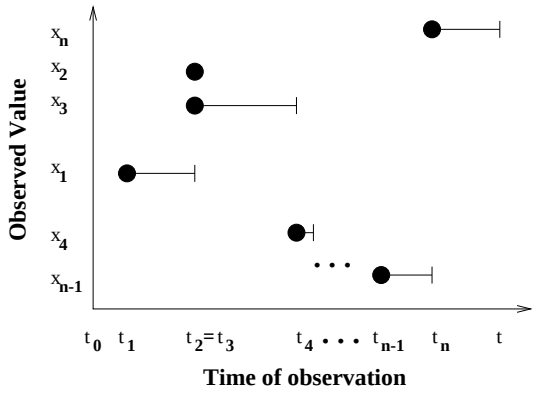


Figure 19: Observed values for timed statistical variables.

The sum and the average of the values at time $t \geq t_n$ are calculated in the following way:

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

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

With timed statistical variables, it is possible for a value to exist for zero time, see for example Fig. 19 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 ($t_2 = t_3$). In this situation, in contrast to the above mentioned statistical measures, the measures maximum, minimum and count 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. This is due to the fact that maximum, minimum, and count are the only statistics that are 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 the statistics are accessed 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 Data Collectors

A data collector is a central concept in the performance tool. Data collectors determine how to extract values from a CPN model, when to extract these values, and what to do with the values. It is possible to maintain several different data collectors (timed or untimed with integer or real values) - each extracting different values from markings and/or binding elements during simulation. Section 6.1 gives a more detailed definition of a data collector. Sections 6.2.1 and 6.2.2 discuss predicate and observation functions, respectively. Section 6.3 shows how to create a data collector. Finally, Sect. 6.4 describes how to initialise data collectors.

## 6.1 What is a Data Collector

A data collector determines when to extract data, how to extract data and and what to do with the data. Each data collector contains two functions: one which determines when to extract data from the CPN model, and one which determines which data to extract. These functions are called the *predicate function* and the *observation function*, respectively.

The values for each data collector must be dealt with somehow. One option is to store the values in an observation log file (see Sect. 7.2). The other option is to maintain a statistical variable. A description of statistical variables was given in Sect. 5. It is also possible to maintain both an observation log file and a statistical variable. Furthermore, one must decide whether the data collector should be timed or untimed. The type of the data collector (untimed or timed) will determine whether or not the associated statistical variable untimed or timed. The difference between untimed and timed statistical variables can be found in Sects. 5.1 and 5.2.

## 6.2 Performance Functions

Three user-defined functions are needed for each data collector; these functions are collectively referred to as performance functions. Two of the functions are associated with a data collector, and the third is used to create a data collector.

The two functions associated with a data collector are the *predicate* and *observation* functions. A predicate function determines *when* a data collector is updated. An observation function is used to calculate the *value* with which the corresponding data collector is updated.

The predicate function is evaluated after each step of the simulator. If it evaluates to `true` then the observation function is invoked, and the corresponding data collector is updated with the value observed by the observation function.

The relation between the predicate function and the observation function is as follows:

---

**if** predicate **then**
    update_datacoll(observation)
**else** ();

---

The *create* function is the third function. It is used to actually create a data collector. The create function is predefined, and it only needs to be called with an appropriate parameter, which will be described later.

All performance functions must be written in the performance node on the performance page. Note that Sects. 6.2.1, 6.2.2, and 6.2.3 contain detailed syntax for the performance functions. These functions can be generated by using the template facility described in Sect. 6.3.

### 6.2.1 Predicate Function

A predicate function must have the following format:

```
fun <PredicateFuncName> (net_marking, binding_element)
```

with type:

```
fun <PredicateFuncName>: CPN'OGrec * Bind.Elem -> bool
```

For the Ferris wheel model a predicate function has been defined; it returns `true` if either of the following transitions occur: *Customer Arrives* (with the name *Customer*) or *LoadNext Customer* (with the name *LoadNext*). This is achieved by matching the binding elements

```
Bind.FerrisWheel'Customer _
```

and `Bind.FerrisWheel'LoadNext _`

The function looks like this:

```
fun CustQueuePred (net_marking, binding_element) =
    let
        fun filterFun (Bind.FerrisWheel'Customer _) = true
          | filterFun (Bind.FerrisWheel'LoadNext _) = true
          | filterFun _ = false
    in
        filterFun binding_element
    end;
```

### 6.2.2 Observation Function

An observation function must have the following format:

```
fun <ObservationFuncName> (net_marking, binding_element)
```

with one of the following types:

```
fun <ObservationFuncName>: CPN'OGrec * Bind.Elem -> int

fun <ObservationFuncName>: CPN'OGrec * Bind.Elem -> real
```

For the Ferris wheel model an observation function has been defined; it depends upon the marking of the place *Waiting Customers* (with the name *Waiting*). The observation function returns the number of tokens on the place in the current marking. This is done by extracting the marking of the place *Waiting Customers* (using the marking function `PerfMark.FerrisWheel'Waiting`) from the current net marking. Finally, the size of the marking is returned.

```
fun CustQueueObs (net_marking, binding_element) =
    let
        val mark_Waiting = PerfMark.FerrisWheel'Waiting 1 net_marking
    in
        CPN'size mark_Waiting
    end;
```

### 6.2.3 Create Function

There are two different create functions:

- `PerfCreate.newIntDataColl`
- `PerfCreate.newRealDataColl`

The return type of the observation function (int or real) will determine which create function to use. In the following `PerfCreate.newIntDataColl` will be used to illustrate the examples. The type and use of the `PerfCreate.newRealDataColl` function are analogous.

Evaluating the `PerfCreate.newIntDataColl` with an appropriate parameter will actually create a data collector. The type of this function is as follows:

```
fun PerfCreate.newIntDataColl :
              {Name:  string,
              PredFun :  CPN'OGrec * Bind.Elem -> bool,
              ObsFun :  CPN'OGrec * Bind.Elem -> int,
              DataCollType :  <untimedDC | timedDC>,
              StatVar :  bool,
              LogFile :  string option} -> unit
```

The `Name` attribute specifies the name of the data collector. The `PredFun` and the `ObsFun` attributes hold the predicate and the observation function, respectively. The `DataCollType` attribute indicates whether the data collector depends on time or not (`timedDC` or `untimedDC`). The attribute `StatVar` specifies whether a statistical variable has to be maintained. Finally, the attribute `LogFile` indicates whether an observation log file has to be maintained. The value `NONE` indicates that an observation log file will not be maintained, and `SOME "filename"` indicates that an observation log file with name `filename` will be maintained.

For the Ferris wheel model, the following is used to create a data collector:

```
PerfCreate.newIntDataColl
        {Name = "CustQueue",
        PredFun = CustQueuePred,
        ObsFun = CustQueueObs,
        DataCollType = timedDC,
        StatVar = true,
        LogFile = SOME ("CustQueue"^
                        Int.toString(floor(!interArrival))^
                        ".log")}
```

The data collector has the name `"CustQueue"`, and it will be updated with values observed by the function `CustQueueObs`. This observation function will be evaluated only when the function `CustQueuePred` returns `true`. These two functions were defined in Sects. 6.2.1 and 6.2.2. The observed values will be stored in a timed statistical variable and an observation log file. The name of the observation log file depends on the value of the variable `interArrival`. For example, when `interArrival=90.0` the name of the observation log file is `CustQueue90.log`.

## 6.3 Creating Data Collectors

A new data collector is created by evaluating performance functions: namely predicate, observation and create functions. These functions must be written in the performance node. It is possible to generate a template for each of these types of functions while using the performance tool.

To create data collectors using ML templates perform the following sequence of operations:

1. Select a group of transitions and places (*Shift* and *left-click*) that you want to refer to in the predicate and/or the observation function.
2. Invoke **Data Collector** in the **Perf** menu. The dialog box shown in Fig. 20 appears.

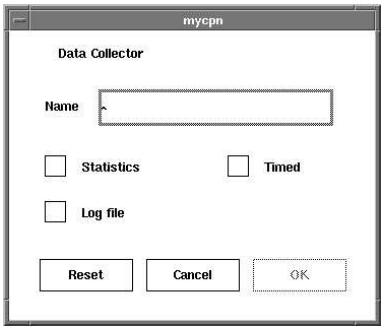


Figure 20: **Data Collector** dialog box.

3. Give the data collector a name.
4. Use the check box **Statistics** if you want a statistical variable to be maintained for the data collector. The statistics can be viewed in a performance report.
5. Use the check box **Log file** if you want an observation log file to be maintained. An observation log file contains each value which has been observed by the observation function.
6. Use the check box **Timed** if you want the data collector to depend on time. If the box is checked, you get a timed statistical variable and/or a timed observation log file, otherwise they are untimed.
7. Click **OK**. This will append some template ML code to the code in the performance node on the performance page. This new code contains templates for the three performance functions: a predicate function, an observation function and a create function.
8. Modify this template ML code to get the intended behaviour.
9. Evaluate these performance functions in one of two ways:
   (a) Select **Initial State** in the **Sim** menu
   (b) Evaluate them manually with the following two steps:
      i. Select the predicate and observation function, evaluate them using **ML Evaluate** in the **Aux** menu, thereby determining the return type of the observation function.

ii. Install the new data collector by evaluating the `PerfCreate.newIntDataColl` or `PerfCreate.newRealDataColl` call.

Examples of template code for the Ferris wheel model can be found in Appendix A. The appendix also contains precise directions for generating the same code.

Templates do not have to be used when writing performance functions. If performance functions are written from scratch, remember that they **must** be written in the performance node. After adding performance functions to the performance node, they must be evaluated as in Step 9 above.

**Important:** No syntax check is made when evaluating ML code in the performance node. The user is responsible for checking if any error has occurred by checking the auxiliary region that appears next to the performance node when **ML Evaluate** is invoked.

## 6.4 Initialising Data Collectors

If the user chooses **Initialize Data Collectors** in the **Perf** menu, then *all* the data collectors are initialised, i.e. all statistics collected in statistical variables are deleted, and all observation log files are emptied.

# 7 Output Facilities

The performance tool can create two types of output files. The first is a performance report which provides statistics about the data that has been collected. The second type is an observation log file which is simply a file that contains each value that has been observed by one specific observation function.

## 7.1 Performance Report

A performance report can be saved anytime a simulation is stopped. Stop criteria can be set by invoking **General Simulation Options** in the **Set** menu. A performance report contains statistics about the data that has been collected during a simulation. These statistics are extracted from all of the statistical variables within the data collectors. A list of the available statistics can be seen in either Fig. 21 below, or in Sect. 5.

### 7.1.1 Report Setup

The layout of the performance report can be specified by using the **Report Setup** entry in the **Perf** menu. The dialog box shown in Fig. 21 appears.

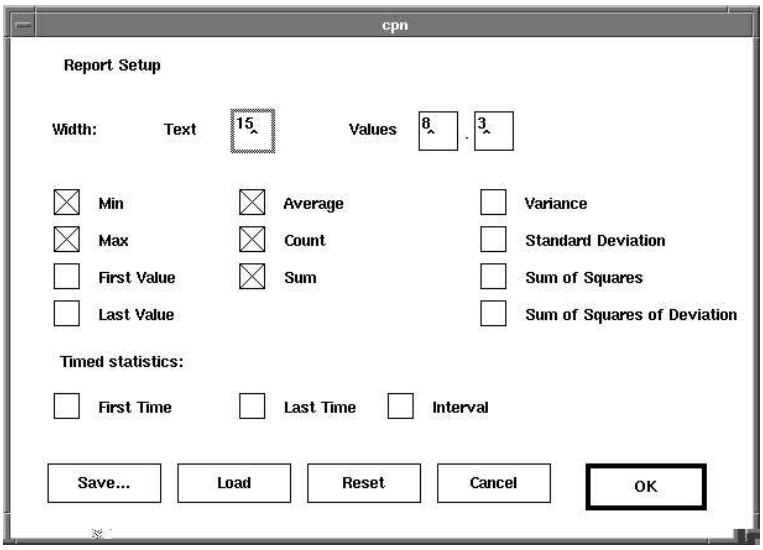


Figure 21: **Report Setup** dialog box.

**Text:** specifies the width of the column containing the names of the data collectors.

**Values:** defines the format of numbers. The first number is the number of digits in the integer part of numbers. The second number indicates the number of decimal positions when reals are included in the performance report. Note that when creating a performance report, if a number is too large to fit in the given column size, then the number is replaced by *'s in the performance report.

**Statistics:** by checking the boxes of some of the statistics, you indicate that you want these statistics to be printed for each statistical variable.

TIMED STATISTICS

| Name | Count | Sum | Average | Minimum | Maximum |
|---|---|---|---|---|---|
| CustQueue | 408 | 192971 | 10.620 | 1 | 21 |

UNTIMED STATISTICS

| Name | Count | Sum | Average | Minimum | Maximum |
|---|---|---|---|---|---|
| CustWait | 200 | 3044 | 15.220 | 0 | 30 |

Current step: 509
Current time: 18171

Figure 22: Example of performance report.

### 7.1.2 Save Report

To save a performance report, select **Save Report** in the **Perf** menu. A dialog box will appear, and the name of the file in which the performance report will be saved can be specified.

An example of a performance report can be seen in Fig. 22. The functions that were used to define the data collectors can be found in Appendix B. The report was created after simulating the Ferris wheel model when *interArrival=90.0* until the time advanced to 18171. At this point the simulation had taken 509 steps.

## 7.2 Observation Log File

Data that has been collected during a simulation can be stored in an *observation log file*. An observation log file will automatically be generated in the file specified in the call for creating the data collection. Note that it may be useful to specify an absolute path to the observation log file. This is due to the fact that otherwise the observation log file will be saved in the current directory which may not be the directory that one would expect.

There are two forms of observation log files: untimed and timed. An untimed and a timed observation log file differ in how they are updated. Each observation log file contains two columns of numbers. The value in the first column depends on the type (timed or untimed) of the observation log file. In an untimed observation log file, the value in the first column is simply a counter indicating the number of the update. Whereas in a timed observation log file, the first column contains the simulation times at which the observation log file was updated. The second column always contains the values that were returned by the observation function. Table 1 shows the format for each type of observation log file.

**Important:** When you select **Initial State** (in the **Sim** menu) the performance node is reevaluated, and data collectors are created. If an observation log file has the same name each time the performance node is evaluated, then the contents of the log file will be lost when a new simulation is started. You must either create unique file names or move the observation log files before starting a new simulation. An example of how to create varying file names can be seen in Sect. 6.2.3.

| Untimed | | Timed | |
|---|---|---|---|
| 1 | $x_1$ | $t_1$ | $x_1$ |
| 2 | $x_2$ | $t_2$ | $x_2$ |
| 3 | $x_3$ | $t_3$ | $x_3$ |
| | $\vdots$ | | $\vdots$ |

Table 1: Observation log file format.

The data in the observation log files can be used to plot graphs. In particular, the data can be plotted by the **gnuplot** plotting program [5]. Figure 23 shows a graph that was created using **gnuplot** and the data in the `CustQueue90.log` observation log file. The gnuplot script that was used to create the graph in Fig. 23 can be found in Appendix C.

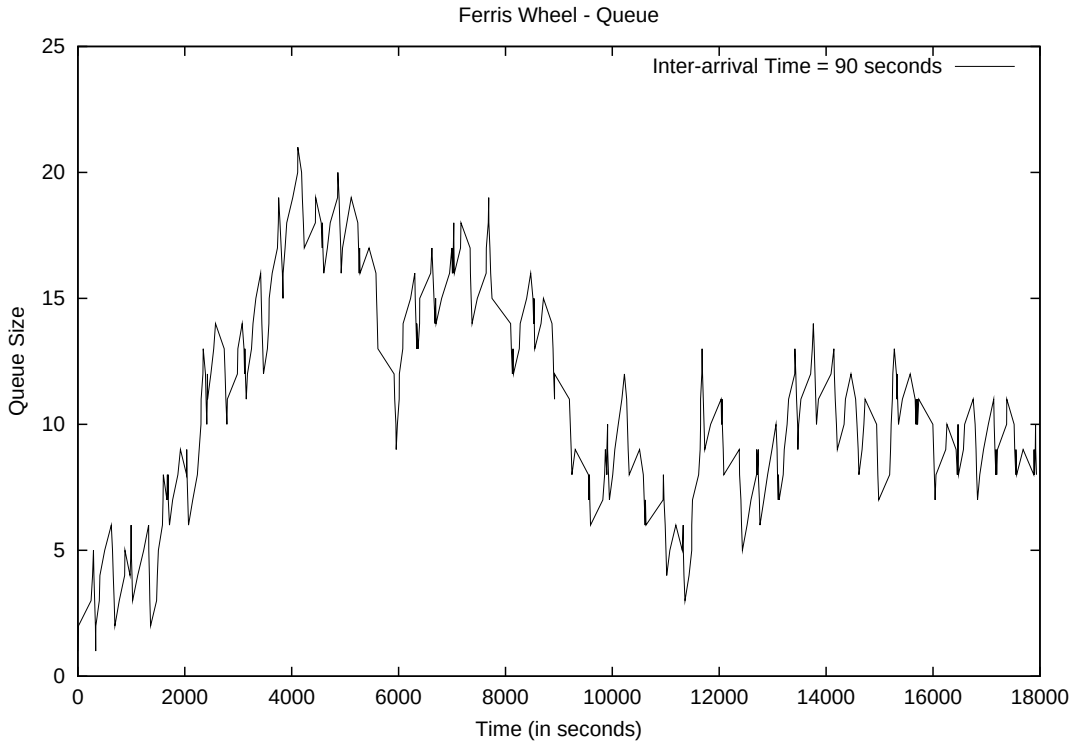


Figure 23: Plotted data from observation log file.

# 8 Accessing Statistical Variables within Data Collectors

This section describes functions that can be used to access statistical variables within data collectors (see Sect. 5 for details about statistical variables). When defining predicate and observation functions, it is sometimes advantageous to be able to access statistical measures of the statistical variable inside a data collector, e.g. to be able to refer to the current sum or average. One can also be interested in creating an observation log file which can be used for plotting the variance of a certain measure.

This is possible by using the functions described in this section. Note that the functions can only be used after switching to the performance tool and cannot be used in arc expressions.

A structure named `PerfAccessDC` contains the functions. The general format of the functions is:

```
PerfAccessDC.<Measure>[I|R] "Name of DC"
```

where

- `<Measure>`: Identifies the measure to be accessed, i.e. `Sum`, `Avrg`, etc.
- `[I|R]`: Some measures have different types – depending on the type of the values stored in the data collector, i.e. the return type of the observation function. "`I`"=`int`, "`R`"=`real`, while "" indicates that only one type is possible for this measure.
- `"Name of DC"`: identifies the data collector to be accessed, i.e. it has to be replaced by the full name of the data collector.

Note that the statistical measures that depend on time are evaluated at *current time*.

## 8.1 Functions for Accessing Data Collectors

**Average**

```
fun PerfAccessDC.Avrg:string -> real
```

**Count**

```
fun PerfAccessDC.Count:string -> int
```

**First Observed Value**

Type of values: *int*

```
fun PerfAccessDC.FirstI:string -> IntInf.int
```

Type of values: *real*

```
fun PerfAccessDC.FirstR:string -> real
```

**Maximum**

Type of values: *int*

```
fun PerfAccessDC.MaxI:string -> IntInf.int
```

Type of values: *real*

```
fun PerfAccessDC.MaxR:string -> real
```

**Minimum**

Type of values: *int*

```
fun PerfAccessDC.MinI:string -> IntInf.int
```

Type of values: *real*

```
fun PerfAccessDC.MinR:string -> real
```

**Sum**

Type of values: *int*

```
fun PerfAccessDC.SumI:string -> IntInf.int
```

Type of values: *real*

```
fun PerfAccessDC.SumR:string -> real
```

**Sum of Squares**

Type of values: *int*

```
fun PerfAccessDC.SSI:string -> IntInf.int
```

Type of values: *real*

```
fun PerfAccessDC.SSR:string -> real
```

**Sum of Squares of Deviation**

```
fun PerfAccessDC.SSD:string -> real
```

**Standard Deviation**

```
fun PerfAccessDC.StD:string -> real
```

**Last Observed Value**

Type of values: *int*

```
fun PerfAccessDC.ValueI:string -> IntInf.int
```

Type of values: *real*

```
fun PerfAccessDC.ValueR:string -> real
```

**Variance**

```
fun PerfAccessDC.Vari:string -> real
```

## 8.2 Additional Functions for Timed Data Collectors

The functions below access statistical measures that are only defined for timed data collectors.

**Time Interval**

```
fun PerfAccessTimedDC.Interval: string -> IntInf.int
```

**Time of Last Observation**

```
fun PerfAccessTimedDC.LastTime: string -> IntInf.int
```

**Time of First Observation**

```
fun PerfAccessTimedDC.StartTime: string -> IntInf.int
```

## 8.3 Exceptions

of
If

# 9 Exception Reporting

This section discusses reporting of exceptions in the performance tool. A number of different exceptions may be raised while manipulating data collections. The performance tool provides some basic feedback concerning exceptions that have been raised, provided that the exceptions are raised in connection with data collection or file manipulation in the performance tool.

An exception report is created for the following types of exceptions:

- I/O exceptions raised when manipulating observation log files.
- I/O exceptions raised when saving performance reports.
- Exceptions raised when updating and accessing statistical variables.
- Exceptions raised when evaluating predicate and observation functions.

If one of the above types of exceptions is raised, an *exception report* will be created. This is done by first creating a page called **Perf_Exceptions**. On this page, an auxiliary node will be created, and the description of the exception that has been raised is written in this node. After the report has been written on the **Perf_Exceptions** page, a dialog box opens indicating that an exception has been raised, and the simulation stops.

Each exception report contains the name of the CPN diagram and some indication of the type of exception that was raised. If an exception is raised when evaluating a predicate or observation function, then the current step number and binding element are also included in the exception report. If an exception is raised while updating a data collection, then the name of the data collection will be included in the report. Finally, if an exception is raised in connection with file manipulation, then the name of the file will be included in the exception report. An example of an exception report is shown in Fig. 24.

```
Design/CPN Performance Tool Exception Report

Diagram:  /users/cpnuser/model/FerrisWheel

An Io exception was raised while manipulating
an observation log file.

Error:  open_out "/users/cpnuser/CustQueue.log":
openf failed, Permission denied.
```

Figure 24: Example of an exception report.

## References


[1] S. Christensen and L.M. Kristensen. State space analysis of hierarchical coloured petri nets. In B. Farwer, D. Moldt, and M.O. Stehr, editors, *Proceedings of Workshop on Petri Nets in System Engineering (PNSE'97) Modelling, Verification, and Validation, Hamburg, Germany*, volume 205, pages 32–43. University Hamburg, Fachberich Informatik, 1997.

[2] Design/CPN Online
Online: http://www.daimi.au.dk/designCPN/.

[3] *Design/CPN Reference Manual*.
Online: http://www.daimi.au.dk/designCPN/man/.

[4] Theo Drimmelen. Implementation of Statistical Functions in Design/CPN.
Online: http://www.daimi.au.dk/designCPN/libs/pdf/.

[5] Gnuplot
Online: http://www.cs.dartmouth.edu/gnuplot_info.html.

[6] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.

[7] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.

[8] Kurt Jensen, Søren Christensen, and Lars M. Kristensen. *Design/CPN Occurrence Graph Manual*. Department of Computer Science, University of Aarhus, Denmark.
Online: http://www.daimi.au.dk/designCPN/man/.

[9] Standard ML of New Jersey
Online: http://cm.bell-labs.com/cm/cs/what/smlnj/.

# A Template Functions

This appendix contains template code that was generated by the performance tool. A description of the exact steps that were taken in order to generate the given code can be found below. Appendix A.1 shows template functions for collecting data concerning the length of the customer queue. Appendix A.2 contains template functions for customer wait time. The modified version of the code which can be used to create data collectors can be found in Appendix B.

## A.1 Customer Queue

This appendix contains template code which can be modified in order to calculate the length of the customer queue. When the transition *Customer Arrives* (with the name *Customer*) occurs, a new customer arrives and the length of the queue increases. When the transition *LoadNext Customer* (with the name *LoadNext*) occurs, the length of the queue decreases because a customer leaves the queue. The customers waiting in the queue are modelled by the tokens on the place *Waiting Customers* (with the name *Waiting*).

The following seven steps were taken in order to generate the following template code.

1. Select (in group mode) the transitions *Customer Arrives* and *LoadNext Customer* and the place *Waiting Customers.*
2. Invoke the **Data Collector** item in the **Perf** menu.
3. Give the data collector the name “CustQueue”.
4. Check the box **Statistics**.
5. Check the box **Log file**.
6. Check the box **Timed**.
7. Click OK, and the following code is added to the performance node.

---

```
(* -=> CustQueue <=- *)

fun CustQueuePred (net_marking, binding_element) =
    let
      val mark_Waiting = PerfMark.FerrisWheel'Waiting 1 net_marking                    5
      fun filterFun (Bind.FerrisWheel'LoadNext (1, {noOfCust=noOfCust,n=n,loadTime=loadTime,
                                                    arrivalTime=arrivalTime})) =
        | filterFun (Bind.FerrisWheel'Customer (1, {nextCust=nextCust,n=n})) =
        | filterFun _ =
    in                                                                               10
       filterFun binding_element
    end;
```

```
fun CustQueueObs (net_marking, binding_element) =
    let
      val mark_Waiting = PerfMark.FerrisWheel'Waiting 1 net_marking
      fun filterFun (Bind.FerrisWheel'LoadNext (1, {noOfCust=noOfCust,n=n,loadTime=loadTime,
                                                    arrivalTime=arrivalTime})) =
        | filterFun (Bind.FerrisWheel'Customer (1, {nextCust=nextCust,n=n})) =
        | filterFun _ =
    in
        filterFun binding_element
    end;

PerfCreate.new<Int|Real>DataColl
  {Name = "CustQueue",
   PredFun = CustQueuePred,
   ObsFun = CustQueueObs,
   DataCollType = timedDC,
   StatVar = true,
   LogFile = SOME "CustQueue.log"};
```

## A.2 Customer Waiting Time

This appendix contains template code which can be modified in order to calculate the average waiting time for the customers riding the Ferris wheel. When the transition *LoadNext Customer* (with the name *LoadNext*) occurs, the variable `arrivalTime` on the input arc to *LoadNext Customer* is bound to the time (in seconds) when the customer joined the queue. This value can be used to calculate how long the customer waited in the queue.

The following six steps were taken in order to generate template code.

1. Select transition *LoadNext Customer.*
2. Invoke the **Data Collector** item in the **Perf** menu.
3. Give the data collector the name “CustWait”.
4. Check the box **Statistics**.
5. Check the box **Log file**.
6. Click OK, and the following code is added to the performance node.

```
(* -=> CustWait <=- *)

fun CustWaitPred (net_marking, binding_element) =
  let
    fun filterFun (Bind.FerrisWheel'LoadNext (1, {noOfCust=noOfCust,n=n,loadTime=loadTime,
                                                  arrivalTime=arrivalTime})) =
      | filterFun _ =
  in
    filterFun binding_element
  end;

fun CustWaitObs (net_marking, binding_element) =
  let
    fun filterFun (Bind.FerrisWheel'LoadNext (1, {noOfCust=noOfCust,n=n,loadTime=loadTime,
                                                  arrivalTime=arrivalTime})) =
      | filterFun _ =
  in
    filterFun binding_element
  end;

PerfCreate.new<Int|Real>DataColl
  {Name = "CustWait",
   PredFun = CustWaitPred,
   ObsFun = CustWaitObs,
   DataCollType = untimedDC,
   StatVar = true,
   LogFile = SOME "CustWait.log"};
```

# B Performance Functions

This appendix contains the modified code of the template code in Appendix A. These functions were used when simulating the Ferris wheel model in order to generate the observation log files and performance reports that are discussed in Sects. 1.3, 7.1 and 7.2.

## B.1 Customer Queue

`CustQueuePred` returns `true` when either transition *Customer Arrives* (with name *Customer*) or transition *LoadNext Customer* (with name *LoadNext*) occurs, i.e. it returns `true` only when the length of the queue changes.

`CustQueueObs` returns the number of tokens on place *Waiting Customers* (with name *Waiting*), i.e. it returns the number of customers in the queue. This function is evaluated *only* when `CustQueuePred` returns `true`, i.e. it is evaluated only when the size of the queue changes.

```
(* -=> CustQueue <=- *)

fun CustQueuePred (net_marking, binding_element) =
    let
        fun filterFun (Bind.FerrisWheel'Customer (1, {nextCust=nextCust,n=n})) = true
          | filterFun (Bind.FerrisWheel'LoadNext (1, {noOfCust=noOfCust,n=n,loadTime=loadTime,
                                                      arrivalTime=arrivalTime})) = true
          | filterFun _ = false
    in
        filterFun binding_element
    end;

fun CustQueueObs (net_marking, binding_element) =
    let
        val mark_Waiting = PerfMark.FerrisWheel'Waiting 1 net_marking
    in
        CPN'size mark_Waiting
    end;

PerfCreate.newIntDataColl
  {Name = "CustQueue",
   PredFun = CustQueuePred,
   ObsFun = CustQueueObs,
   DataCollType = timedDC,
   StatVar = true,
   LogFile = SOME ("CustQueue"^Int.toString(floor(!interArrival))^".log")};
```

## B.2 Customer Waiting Time

`CustWaitPred` returns `true` when transition *LoadNext Customer* (with name *LoadNext*) occurs, i.e. it returns `true` each time a customer is removed from the queue.

`CustWaitObs` is evaluated when *LoadNext Customer* occurs. When a customer is removed from the queue, the variable `arrivalTime` is bound to the time at which the customer joined the queue. Therefore, `(current time - arrival time) div 60` is the customer's waiting time in whole minutes (remember: in this model a unit of time is a second). The definition of function `intTime` can be found in Fig. 3.

---

```
(* -=> CustWait <=- *)

fun CustWaitPred (net_marking, binding_element) =
    let
        fun filterFun (Bind.FerrisWheel'LoadNext (1, {noOfCust=noOfCust,n=n,loadTime=loadTime,
                                                      arrivalTime=arrivalTime})) = true
          | filterFun _ = false
    in
        filterFun binding_element
    end;

fun CustWaitObs (net_marking, binding_element) =
    let
        fun filterFun (Bind.FerrisWheel'LoadNext
                       (1, {noOfCust=noOfCust,n=n,loadTime=loadTime,
                            arrivalTime=arrivalTime})) = (intTime() - arrivalTime) div 60
    in
        filterFun binding_element
    end;

PerfCreate.newIntDataColl
  {Name = "CustWait",
   PredFun = CustWaitPred,
   ObsFun = CustWaitObs,
   DataCollType = untimedDC,
   StatVar = true,
   LogFile = SOME ("CustWait"^Int.toString(floor(!interArrival))^".log")};
```

---

# C Gnuplot Scripts

The following scripts were used with gnuplot unix version 3.7.

## C.1 Combined Customer Queues

This script was used to produce the graph shown in Fig. 5. The data that was plotted was saved in three files: `CustQueue60.log`, `CustQueue90.log`, and `CustQueue120.log`. If the script is saved in a file named `combinedqueue.gnu`, then the commands in the script can be executed by typing the following on the command line in gnuplot:

`gnuplot>` load 'combinedqueue.gnu'

---

```
# gnuplot script for Customer Queue
# This is a comment

set title "Ferris Wheel - Queue"
show title
set xlabel "Time (seconds)"
show xlabel
set ylabel "Queue Size"
show ylabel

#set terminal postscript landscape monochrome
#set output "combCustQueue.ps"

plot [] [0:120] \
'CustQueue60.log' title 'Inter-arrival Time = 60 seconds' with lines , \
'CustQueue90.log' title 'Inter-arrival Time = 90 seconds' with lines , \
'CustQueue120.log' title 'Inter-arrival Time = 120 seconds' with lines


#set terminal x11
#replot
```

---

## C.2 Customer Queue

This script was used to produce the graph shown in Fig. 23. The data that was plotted was saved in the file `CustQueue90.log`. If the script is saved in a file named `CustQueue90.gnu`, then the commands in the script can be executed by typing the following on the command line in gnuplot:

```
gnuplot> load 'CustQueue90.gnu'
```

---

```
# gnuplot script for Customer Queue
# This is a comment

set title "Ferris Wheel - Queue"
show title
set xlabel "Time (in seconds)"
show xlabel
set ylabel "Queue Size"
show ylabel

#set terminal postscript landscape monochrome
#set output "CustQueue90.ps"

plot \
'CustQueue90.log' title "Inter-arrival Time = 90 seconds" with lines


#set terminal x11
#replot
```

---

# Index