Universität Stuttgart

INSTITUT FÜR
KOMMUNIKATIONSNETZE
UND RECHNERSYSTEME
Prof. Dr.-Ing. Andreas Kirstädter

IKR Simulation Library 3.2
User Guide

Reference Guide

October 24, 2012

Editors

Marc Barisch
Marc Necker
Stefan Bodamer
Ingo Kauffmann
Christian Blankenhorn
Christoph Gauger
Klaus Dolzer
Jörg Sommer
Joachim Scharf
Sebastian Scholz

Table of Contents

1 Introduction

In this document, we will introduce the basic concepts of the IKR Simulation Library (IKR SimLib). We demonstrate systematically the application of these concepts with a practical example in a separate part of the user guide [5].

1.1 History

The IKR Simulation Library is a tool, which is mainly used for event-driven simulation of complex systems in the area of communications engineering. Originally, Hartmut Kocher designed an object-oriented version of the IKR SimLib in 1993 during his dissertation [10] and implemented it in C++. Since this original design, we enhanced and improved the IKR SimLib continously.

In 2008, we ported the IKR SimLib to Java while keeping all concepts and mechanisms of the existing C++ class library. Today, two editions of the IKR SimLib are available: The *C++ Edition* and the *Java Edition*. Each edition comes as a separate class library. The IKR SimLib is publicly available under the GNU Lesser General Public License (LGPL) and thus allows changes within the libraries itself as well as proprietary programs to use it.

In the last years, we have successfully used the IKR SimLib for performance evaluation in various projects from different areas in communication network research, e.g. IP, ATM, photonic, mobile, and signaling networks.

1.2 Conceptual Structure

We structure the IKR Simulation Library into three main parts (Fig. 1.1). Basic concepts include simulation support mechanisms like event handling (Section 2), simulation IKR control (Section 3), distribution-oriented random number generation (Section 4), and tools to statistically evaluate measured values (Section 5) as well as reading parameters (Section 10) and printing simulation results (Section 9) are provided.

Beyond that, the IKR Simulation Library contains concepts for constructing hierarchical models from individual components (Section 6) that communicate with each other by exchanging messages (Section 7.1). This message exchange occurs using so-called ports (Section 7), which are used to define an external interface of a model component. Then, we can connect this interface to meters (Section 8), which allow a simple determination of measurement values.

1.3 Naming Conventions

In order to ensure that the source code of the IKR SimLib is easy to read and maintain, we follow Sun's Code Conventions for the Java Programming Language (see http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html). Depending on the type, the first letter of a camel case compound may or may not be capitalized. For example class names use a capitalized letter (e.g., Queue), and variables and attributes a lowercase letter (e.g., name).

The identifier of an element should be self-explanatory (e.g., `maxLength, routingManager)`. If the element's name consists of several individual words, these are preferably written together, where each new part of the word begins with a capitalized letter.

2 Event Handling

The simulation library is based on the principle of event-driven simulation. Stemming from a model in which state transitions take place at discrete points in time (therefore discrete time simulation), the simulated time of the original system is represented by the real-time of the simulation so that the "idle time" between state transitions is bridged and no further processing time is needed (see Fig. 2.1).

The state transitions are described with the help of events. These events have two characteristics. One is a time stamp, which defines the simulated point in time at which the event occurred. The other is, e.g., an event type, which defines the state transition cause, a change of system parameters, generation of subsequent events, etc. An example of an event might be the arrival of a request at the generator or the end of service for a request in a service unit.

2.1 Events

The term "event" as used in the simulation library does not mean a single time-stamped event, rather a summary of single events e.g., the event „end of service in phase 1“. The term "event" is used on a higher level of abstraction.

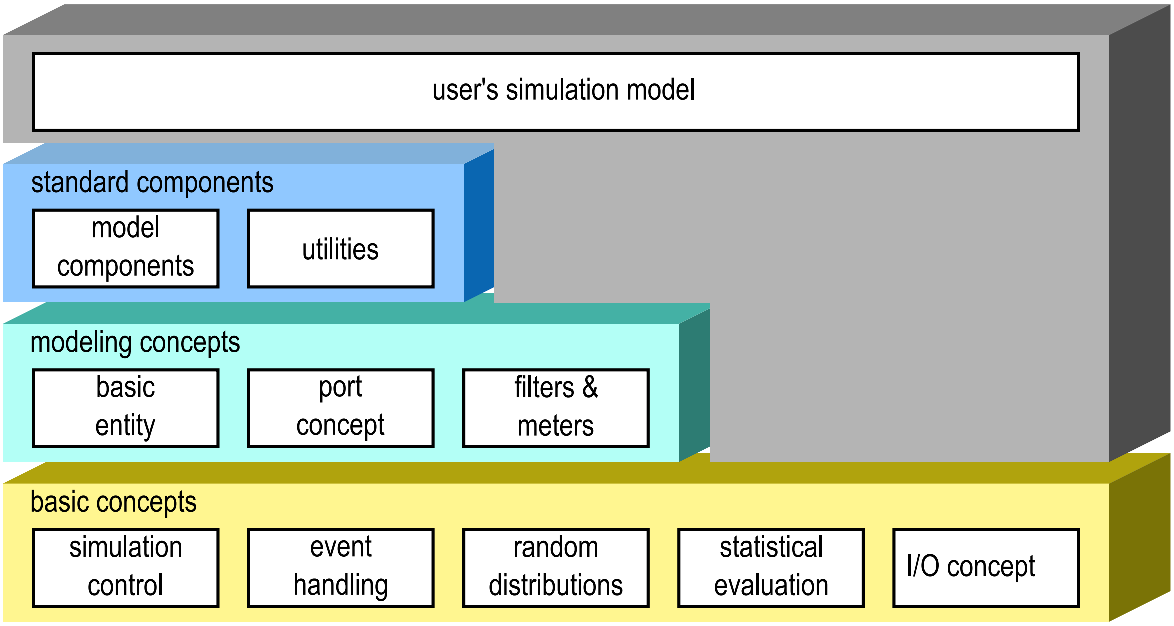


Figure 1.1: Structure of the IKR Simulation Library

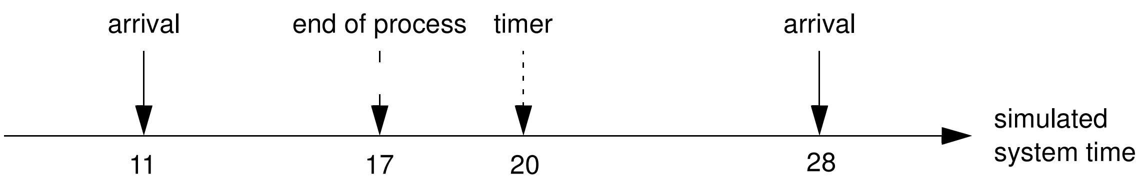


Figure 2.1: Example of event occurences in a system

All events are derived from class `Event`, which has the following characteristics:

- An integer field defines the event type.
- An event can contain a list of embedded events which can be added using `addEmbeddedEvent()`.
- The function `processEvent()` is called by the calendar (Section 2.2) when the event occurs. It will first call the `handleProcessEvent()` functions of all embedded events and then its own `handleProcessEvent()` function.
- The function `cancelEvent()` is called by the calendar (Section 2.2) when the event is cancelled. It will first call the `handleCancelEvent()` functions of all embedded events and then its own `handleCancelEvent()` function.
- A time stamp is <u>not</u> a part of `Event` (see above).

2.2 Calendars

All calendars, e.g. `SimpleCalendar`, are derived from the abstract base class `Calendar`. The class `Calendar` offers an interface for the registration and deregistration of events:

- `postEvent()` publishes an event with the address of the `Event` object and the event time stamp as parameters. Additionally, a priority may be passed on. It is an integer and controls the order in that events are processed, which are posted for the same time. Larger values mean higher priorities. The default priority is 0. The processing order of events posted for the same time and with the same priority is undefined.
- `cancelEvent()` serves to remove the defined event from the calendar which automatically calls the `cancelEvent()` function of the event. In case an event has been posted multiple times, it depends on the calendar implementation which one is removed from the calendar.
- `cancelEvent(double)` can be used to spefically remove an event posted for a particular event time. Only one post is removed per method call, even if there exist multiple posts for the same time.
- `cancelAllEvents()` deletes all calendar contents.
- `cancelAllEvents(Event)` deletes all posts of a particular event.
- `popNextEvent()` returns the event to be executed next and removes it from the calendar.
- `processNextEvent()` removes the next event from the calendar and causes its execution by calling the `processEvent()` method of that event.
- `peekNextEvent()` also returns the event to be executed next without removing it from the calendar.

The form in which the events are stored in the calendar, after being registered in the calendar and waiting for their execution, must be defined in derived classes. The library offers two special calendars:

- `StdCalendar` uses a linear list for storing the events. For a high number of events within the calendar this implementation gets slow. Therefore this class is only recommended for models with few events or for test purposes.
- In `SimpleCalendar` a relatively complex but efficient procedure, especially for large numbers of waiting events, is used to store event information.

Generally, a system has a central calendar.

2.3 Event Handling Procedures

2.3.1 Posting Events

The following steps are necessary if a function of a model component is to register an event:

- The `postEvent()` method must be called with a reference to the event (`Event` or one of its derived classes), which generally is a data element of the model component (Section 6), and the time at which the event is expected to occur as parameters. Optionally, a priority may be passed on. It is an integer and controls the order in which events are processed, which are posted for the same time. Larger values are more important and cause events to be executed primarily.

2.3.2 Cancelling Events

If it should happen that an existing event is no longer valid, because another event has already occurred, then the event will be removed from the event list of the calendar.

- The `cancelEvent()` method is called with a reference to the event as a parameter which will cause the removal of the event.
- Furthermore, the calendar calls the `cancelEvent()` method on the event to be removed.
- The event time is passed along with these methods.

2.3.3 Processing Events

When processing events, the simulation library offers the user many degrees of freedom. In the typical case of a central calendar of the type `Calendar`, or one of its derived classes, the following actions will occur:

- The sequence control (class derived from `Simulation`, Section 3) will call the function `processPendingEvents()`.
- `processPendingEvents()` contains a loop, in which events can be successively processed. This loop can be indirectly stopped by sequence control.
- Within the loop the function `getNextEvent()` from the class `Calendar` is called first, which will remove the currently next event in the event list and return a pointer.
- After that the event is processed by calling the `Event` method `processEvent()` (Section 2.1). Normally the responsibility is passed on to the responsible model component.

2.3.4 Posting Events Multiple Times

Events may be posted multiple times as long as they do not use embedded events. A particular post of an event may can be identified by its time for which it is posted. When specifying a time with `cancelEvent()`, a particular post may be cancelled.

2.3.5 Associating a Context with a Posted Event

Often, it is useful to associate a context with an event, that has been posted to a particular time in the calendar. For example, the `InfiniteServer` associates a message with each event posting in order to forward the respective message on the output port when the event is processed. The easiest way to implement this is by deriving a new class from `Event`. For example see the class `InfiniteServerEvent`.

3 Simulation Control

The sequence control is a summary of tasks, that take place in every simulation, which can be adapted to each problem in a flexible manner. It can also be applied to simple problems without change and without exact knowledge of the internal structures.

In order to control a sequence the abstract base class `Simulation` has been developed that defines functions for the individual steps, which are executed during a simulation. Most of these functions are empty and can be overriden in derived classes.

The class `StdSimulation` derived from `Simulation` has been pre-defined and will suffice for many simulations. One entity of this class is generally declared in the `main()` function of the simulation program and requires the following parameters upon constructor call:

- a reference to the model (class `Model`, Section 6) which is to be used for simulation
- a reference to the simulation environmant (class `SimulationEnvironment`)

The class `Simulation` makes use of `SimulationEnvironment` in order to read command line parameters. The possible parameters are given in the following:

Parameter	*Range*	*Description*
n	> 0	number of batches
s	0 - 127	index of seed for random number generator
p		parameter file
f		filter file
l		log file
b		batch log file
e		export file for batch results
i		import file for batch results
g		parse log file
t		test log file
c		type of calendar

If specified the number of given export and import files needs to match to the number of batches. Simultaneous definition of import and export files is not possible.

3.1 Simulation Phases

The individual phases of a simulation – realized by methods of the class `Simulation` – are processed when the method `run()` from `Simulation` has been called. Normally, this occurs in the `main()` function of the simulation program. Beginning with the derived class `StdSimulation`, in which the methods have been (partly) overridden so that they mirror a normal sequence of a simulation, the following actions take place:

- `initSimulation()`:

- initializes internal data structures
- notifies objects of the class `SimulationControl` (Section 3.3)
- reads the print formats (Section 9)

• `runSimulation()`:

- executes the actual simulation with a subdivision in simulation phases
  - a warm-up (transient) phase at the beginning of simulation
  - several batches during which statistical data is collected

  In each phase event generation and processing is started by calling the method `processPendingEvents()` (Section 2.3.3) of the model object given as parameter on construction of `StdSimulation`
- notifies objects of derived from `SimulationControl` (Section 3.3) at the beginning and end of each simulation phase as well as at the end of the simulation
- prints the partial results after each batch (`printBatchResults()`)

• `printResults()`:

- prints the simulation results by calling the `PrintManager` method `printResults()` with the respective print format names and the output stream as parameters (Section 9)

• `cleanup()`:

- executes clean-up jobs like cancelling all events in the calendar at the end of simulation

Besides this approach, there is another possibility to achieve simulation results, which makes use of batch parallelization. Here the intermediate results of each batch are written to a special file. In a second run, all these batch results are read in and the final results are calculated.

Each parallelized run goes through all of the above mentioned phases. Thus there is a transient phase for each run, which means that in total more has to be simulated in comparison to a non-parallelized simulation. As consequence the usage of batch parallelization is only reasonable when having more CPUs or cores than simulations or when the time to execute the simulation needs to be reduced.

3.2 Triggering State Transitions

The following classes have been introduced in order to create a flexible concept in which state transitions within the simulation (e.g., the end of the simulation or a batch) can be made dependent on the system state (e.g., number of messages that have passed a certain port):

• `Notifier`

- must have an owner of type `MultiNotificationHandler`
- passes calls of its own method `notify()` on to the responsible notification handler

• `NotificationHandler`

- interface class

  - the method `notify()` is called by the notifiers which itself calls the method `handleNotification()`; this method is to be overridden in derived classes

- `MultiNotificationHandler`
  - manages several `NotificationHandler`
  - is connected to sequence control (`Simulation`)
  - the method `notify()` calls a handler method from `Notifier`
  - `Simulation` contains three objects of the type `MultiNotificationHandler`, one each for the end of the transient phase, the end of the batch and the end of the simulation
  - the corresponding handler methods in `Simulation` for the three objects are `handleEndOfTransientPhase()`, `handleEndOfCurrentBatch()` and `handleEndOfSimulation()`

- `SimNotifier`
  - contains three objects of the type `Notifier`, one each for the end of the transient phase, the end of the batch and the end of the simulation
  - the notifiers are registered with the corresponding `MultiNotificationHandler` objects from `Simulation`
  - the function calls `endOfTransientPhase()`, `endOfBatch()` and `endOfSimulation()` causes `notify()` calls of the corresponding `Notifier` objects
  - implements `SimulationControl` (Section 3.3)

- `ControlCounter`
  - controls the duration of a simulation by the number of messages in the batches.
  - derived from `SimNotifier` (Section 8)
  - upon constructor call receives the number of messages per transient phase and per batch as well as the number of batches as parameters
  - is "connected" to a port of the inherited function `attachInput()`"
  - calls, depending on the system state, the function `endOfTransientPhase()` or `endOfBatch()`, if a sufficient number of messages have passed the port

- `ControlTimer`
  - controls the duration of a simulation by the duration (time) of the batches.
  - derived from `Entity`, extends `StdEvent.Handler` and `SimulationControl`
  - upon constructor call the duration of the transient and the batch phase, respectively as well as the number of batches
  - calls, depending on the system state, the function `endOfTransientPhase()` or `endOfBatch()`, if the specified time has elapsed

3.3 Effects of State Transitions

The explicit division of the simulation in single phases by using different methods is done for a purpose. Some classes (e.g. statistics) must perform actions at the beginning/end of phases/partial phases like the initialization or reset of counters. For these purposes, there are the following interfaces:

- `InitSimulationCallback`
- `StartSimulationCallback`
- `StartTransientPhaseCallback`
- `StopTransientPhaseCallback`
- `StartBatchCallback`
- `StopBatchCallback`
- `StopSimulationCallbak`

All objects implementing one or more of these interfaces can be registered with the global instance of the class `SimulationControlManager`. At the beginning/end of each phase/partial phase, the respective functions callback methods will be called.

Some classes of the SimLib already implement some of the interfaces listed above:

- `Statistic` (Section 5)
  - resets the statistics using `handleInitSimulation()` and `handleStopTransientPhase()`
  - resets the batch statistics using `handleStartBatch()`
  - batch evaluation using `handleStopBatch()`
  - publishes the results using `handleStopSimulation()`
- `SimNotifier`
  - internal state is set to `transient` by `handleStartTransientPhase()`
  - internal state is set to `batch` by `handleStartBatch()`
  - checks if end of simulation has been reached `handleStopBatch()`
- `Generator` (Section 6)
  - generates an initial event (activation) upon `handleStartSimulation()`
  - removes incomplete events (deactivation) upon `handleStopSimulation()`
- `SimControlTracer`
- `BatchCounter`
- `ClockedGate`

4 Distributions

In order to reproduce the stochastic processes in the system (e.g., arrival processes, service processes) using events it is necessary to generate random variables for time stamps according to a pre-defined distribution function, possibly taking the system state into consideration. The simulation library offers a number of distributions.

4.1 Generating Random Numbers

The generation of random variables from a distribution function is based on the generation of pseudo-random number sequences with a large period following a uniform distribution between 0 and 1.

Three classes have been derived from `RandomNumberGenerator` that serve as random number generators. In most cases they override the index operator `operator()` and return a random number between 0 and 1:

- `StdRandomNumberGenerator` is a relatively simple but generally sufficient generator [18]. A global instance `SYSTEM_RNG` of this type is generated that is used as the default random number generator for the system (Section 4.2).
- `MixedRandomNumberGenerator` implements a generator with a very large period, which has been generated by mixing two linear congruent generators.

Random variables needed for the simulation are generated based on these random numbers and a specific distribution function (e.g., for random service times or arrival intervals). The distribution function must be inverted (Fig. 4.1), which in some cases can be done exactly, in other cases only approximately [15]. The task of inversion is executed by the distribution classes.

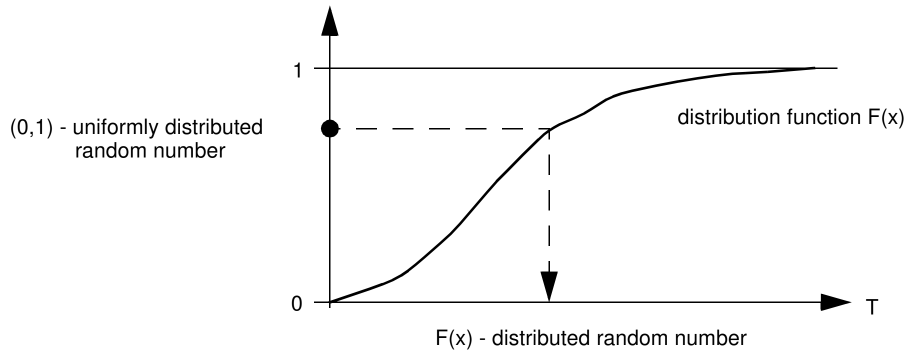


Figure 4.1: Creating a random number according to a certain distribution

4.2 Structure and Interface of Distribution Classes

All distribution classes are derived from the class `Distribution`, which has the following characteristics:

- a final reference `rng` to the its random number generator, which will by default be initialized with a reference to the system default random number generator (Section 4.1)

Basically, discrete and continuous distributions are treated differently. They are represented by the classes `DiscreteDistribution` and `ContinuousDistribution`, which have been derived from `Distribution`.

4.3 Discrete Distributions

Classes, that have been derived from `DiscreteDistribution`, override the abstract method `next()`, which returns an integer value:

4.3.1 Discrete Deterministic Value

Meaning: A constant integer value d is returned

Parameters: constant integer (mean) value d

Distribution:

$$P(X = i) = \begin{cases} 1 & \text{for } i = d \\ 0 & \text{else} \end{cases}$$

Expected value: $\mathrm{E}[X] = d$

Variance: $\mathrm{VAR}[X] = 0$

Coefficient of variation: $c_X = 0$

Generating func.: $G(z) = z^d$

Class: `DiscreteConstantDistribution`

Constructor: `DiscreteConstantDistribution(int mean)`

Parser example:

```
[...].Distribution = DiscreteConstant
[...].Distribution.Mean = 5
```

4.3.2 Discrete Uniform Distribution

Meaning: All integer values i in the interval $b_l \le i < b_u$ (b_l and b_u being also integer values) have the same probability $\frac{1}{b_u - b_l}$

Parameters:

- lower limit b_l
- upper limit $b_u > b_l$

Distribution:

$$P(X = i) = \begin{cases} \frac{1}{b_u - b_l} & \text{for } b_l \le i < b_u \\ 0 & \text{else} \end{cases}$$

Expected value:
$$E[X] = \frac{b_l + b_u - 1}{2}$$

Variance:
$$\mathrm{VAR}[X] = \frac{(b_u - b_l - 1) \cdot (b_u - b_l + 1)}{12}$$

Coefficient of variation:
$$c_X = \frac{\sqrt{(b_u - b_l - 1) \cdot (b_u - b_l + 1)}}{\sqrt{3} \cdot (b_l + b_u - 1)}$$

Generating func.:
$$G(z) = z^{b_l} + \ldots + z^{b_u - 1}$$

Class: `DiscreteUniformDistribution`

Constructor: `DiscreteUniformDistribution(int lowerBound, int upperBound)`

Parser example:
```
[...].Distribution =DiscreteUniform
[...].Distribution.LowerBound = 3
[...].Distribution.UpperBound = 7
```

4.3.3 Bernoulli Distribution

Meaning: Single random experiment with the success probability q $(0 \le q \le 1)$.

Parameters: Success probability = mean value q

Distribution:
$$P(X = i) = p_i = \begin{cases} 1 - q & \text{für } i = 0 \\ q & \text{für } i = 1 \\ 0 & \text{sonst} \end{cases}$$

Expected value:
$$E[X] = q$$

Variance:
$$\mathrm{VAR}[X] = q(1 - q)$$

Coefficient of variation:
$$c_T = \sqrt{\frac{1 - q}{q}}$$

Generating Func.:
$$G(z) = 1 - q + qz$$

Class: `BernoulliDistribution`

Constructor: `BernoulliDistribution(double mean)`

Parser example:
```
[...].Distribution = Bernoulli
[...].Distribution.Mean = 0.6
```

4.3.4 Binomial Distribution

Meaning: Probability for i successes in n Bernoulli trials with the parameter q ($0 \le q \le 1$).

Parameters:

- success probability q ($0 < q \le 1$)
- number of trials $n > 0$

Alternative: mean $E[X] \ge 0$ and variance $\mathrm{VAR}[X]$ ($0 \le \mathrm{VAR}[X] \le \mathrm{E}[X]$) parameters:

- $q = 1 - \dfrac{\mathrm{VAR}[X]}{E[X]}$
- $n = \dfrac{(E[X])^2}{E[X] - \mathrm{VAR}[X]}$

Distribution: $P(X = i) = \binom{n}{i} \cdot q^i \cdot (1-q)^{n-i}$

Expected value: $E[X] = nq$

Variance: $\mathrm{VAR}[X] = nq(1-q)$

Coefficient of variation: $c_T = \sqrt{\dfrac{1-q}{nq}}$

Generating func.: $G(z) = (1 - q + qz)^n$

Class: `BinomialDistribution`

Constructor:

```
BinomialDistribution(double mean, double variance)
BinomialDistribution(double p, int upperBound)
```

Parser example:

```
[...].Distribution = Binomial
[...].Distribution.Mean = 15.0
[...].Distribution.Variance = 10.5
```

4.3.5 Geometric Distribution

Meaning: Probability for i failures prior to the first success in independent Bernoulli experiments with the parameter q ($0 \le q \le 1$)

Parameters: success probability q ($0 < q \le 1$)

With mean parameter m: $q = \dfrac{1}{1+m}$

Distribution:

$$P(X = i) = (1-q)^i \cdot q = \left(\frac{m}{m+1}\right)^i \cdot \frac{1}{m+1} \quad \text{for } i = 0, 1, 2, \ldots$$

Expected value: $E[X] = \frac{1-q}{q} = m$

Variance: $\mathrm{VAR}[X] = \frac{1-q}{q^2} = m \cdot (m+1)$

Coefficient of variation: $c_T = \sqrt{\frac{1}{1-q}} = \sqrt{\frac{m+1}{m}} \geq 1$

Generating func.: $G(z) = \frac{q}{1-z(1-q)} = \frac{1}{m+1-zm}$

Class: `GeometricDistribution`

Constructor: `GeometricDistribution(double mean)`

Parser example:
```
[...].Distribution = Geometric
[...].Distribution.Mean = 2.5
```

4.3.6 Shifted Geometric Distribution

Meaning: Probability for $i-1$ failures prior to the first success in independent Bernoulli experiments with the parameter q $(0 \leq q \leq 1)$

Parameters: Success probability q

With mean value parameter m: $q = \frac{1}{m}$

Distribution: $P(X=i) = (1-q)^{i-1} \cdot q = \left(\frac{m-1}{m}\right)^i \cdot \frac{1}{m-1}$ for $i = 1, 2, \ldots$

Expected value: $E[X] = \frac{1}{q} = m$

Variance: $\mathrm{VAR}[X] = \frac{1-q}{q^2} = m \cdot (m-1)$

Coefficient of variation: $c_T = \sqrt{1-q} = \sqrt{\frac{m-1}{m}} \leq 1$

Generating func.: $G(z) = \frac{qz}{1-z(1-q)} = \frac{z}{m-z(m-1)}$

Class: `ShiftedGeometricDistribution`

Constructor: `ShiftedGeometricDistribution(double mean)`

Parser example:
```
[...].Distribution = ShiftedGeometric
[...].Distribution.Mean = 2.5
```

4.3.7 Poisson Distribution

Meaning: Probability of the number of arrivals in a time interval with the duration t for a Markovian arrival process (limit distribution of a binomial distribution for $n \to \infty$, $q \to 0$, $nq \to \lambda t$)

Parameters: mean value $m = \lambda t > 0$

Distribution:
$$P(X = i) = \frac{(\lambda t)^i}{i!} \cdot \exp(-\lambda t) = \frac{m^i}{i!} \cdot \exp(-m)$$

Expected value: $E[X] = \lambda t = m$

Variance: $\mathrm{VAR}[X] = \lambda t = m$

Coefficient of variation: $c_T = \frac{1}{\sqrt{\lambda t}} = \frac{1}{\sqrt{m}}$

Generating func.: $G(z) = \exp(-\lambda t \cdot (1 - z)) = \exp(-m \cdot (1 - z))$

Class: `PoissonDistribution`

Constructor: `PoissonDistribution(double mean)`

Parser example:
```
[...].Distribution = Poisson
[...].DistributionMean = 2.5
```

4.3.8 Negative Binomial Distribution

Meaning: Probability, that i failed Bernoulli trials with the parameter q will precede k successes

Parameters:

- success probability q $(0 \le q \le 1$
- number of successful trial $k \ge 0$

With mean value and variance parameters:

- $q = \frac{\mathrm{VAR}[X]}{E[X]}$
- $k = \frac{E[X]}{\mathrm{VAR}[X] - E[X]}$

Distribution:
$$P(X = i) = \binom{i + k - 1}{k - 1} \cdot q^k \cdot (1 - q)^i$$

$$= \binom{i + \frac{m^2}{v - m} - 1}{\frac{m^2}{v - m} - 1} \cdot \left(\frac{m}{v}\right)^{\frac{m^2}{v - m}} \cdot \left(1 - \frac{m}{v}\right)^i$$

Expected value: $E[X] = \frac{1-q}{q} \cdot k = m$

Variance: $\mathrm{VAR}[X] = \frac{1-q}{q^2} \cdot k = \frac{m}{q} = v$

Coefficient of variation: $c_T = \frac{1}{\sqrt{(1-q) \cdot k}}$

Generating func..: $G(z) = \frac{q^k}{(1-(1-q) \cdot z)^k}$

Class: `NegBinDistribution`

Constructor: `NegBinDistribution(double mean, double variance)`

Parser example:

```
[...].Distribution = NegBin
[...].Distribution.Mean = 5
[...].Distribution.Variance = 10
```

4.3.9 General Discrete Distribution

Meaning: The first n values p_i ($i = 0, \ldots, n-1$) of the distribution are defined separately

Parameters: single probabilities p_i ($0 \le p_i \le 1$)

Distribution:

$$P(X = i) = \begin{cases} p_i & \text{for } 0 \le i < n \\ 0 & \text{else} \end{cases}$$

Expected value: $E[X] = \sum_{i=0}^{n-1} p_i \cdot i$

Variance: $\mathrm{VAR}[X] = \sum_{i=0}^{n-1} p_i \cdot i^2 - (E[X])^2$

Coefficient of variation: $c_T = \frac{\sqrt{\mathrm{VAR}[X]}}{E[X]}$

Generating func.: $G(z) = \sum_{i=0}^{n-1} p_i \cdot z^i$

Class: `DiscreteGeneralDistribution`

Constructor: `DiscreteGeneralDistribution(double[] probVector)`

Parser example:

```
[...].Distribution = DiscreteGeneral
[...].Distribution.Data = [ 0.1 0.2 0.1 0.4 0.2 ]
```

4.4 Continuous Distributions

The classes derived from `ContinuousDistribution` override the abstract method `next()`, which returns a `double` value.

4.4.1 Deterministic Value

Meaning: A constant value d is returned

Parameters: constant („mean“) value d

PDF:

$$P(T = t) = f(t) = \delta(t - d)$$

DF:

$$P(T \le t) = F(t) = \sigma(t - d) = \begin{cases} 0 & \text{for } t < d \\ 1 & \text{else} \end{cases}$$

Expected value: $E[T] = d$

Variance: $\mathrm{VAR}[T] = 0$

Coefficient of variation: $c_T = 0$

LST: $\Phi(s) = \exp(-sd)$

Class: `ConstantDistribution`

Constructor: `ConstantDistribution(double mean)`

Parser example:

```
[...].Distribution = Constant
[...].Distribution.Mean = 1.7
```

4.4.2 Uniform Distribution

Meaning: All (continuous) values t in the interval $b_l < t < b_u$ appear with the same probability

Parameters:

- lower limit b_l
- upper limit $b_u > b_l$

PDF:

$$P(T = t) = f(t) = \begin{cases} \dfrac{1}{b_u - b_l} & \text{für } b_l \le t \le b_u \\ 0 & \text{sonst} \end{cases}$$

DF:

$$P(T \le t) = F(t) = \begin{cases} 0 & \text{für } t < b_l \\ \dfrac{t - b_l}{b_u - b_l} & \text{für } b_l \le t < b_u \\ 1 & \text{für } t \ge b_l \end{cases}$$

Expected value:

$$E[T] = \frac{b_l + b_u}{2}$$

Variance:

$$\mathrm{VAR}[T] = \frac{(b_u - b_l)^2}{12}$$

Coefficient of variation:

$$c_T = \frac{1}{\sqrt{3}} \cdot \frac{b_u - b_l}{b_u + b_l}$$

LST:

$$\Phi(s) = \frac{1}{b_u - b_l} \cdot \frac{\exp(-b_l s) - \exp(-b_u s)}{s}$$

Class: `UniformDistribution`

Constructor: `UniformDistribution(double lowerBound, double upperBound)`

Parser example:

```
[...].Distribution = Uniform
[...].Distribution.LowerBound = 1.5
[...].Distribution.UpperBound = 13.5
```

4.4.3 Negative Exponential Distribution

Meaning: Time distance between two events (e.g., arrival, process end,...) in a Markovian process with the average rate λ

Parameters: mean value m or rate $\lambda = 1/m$

PDF:

$$P(T = t) = f(t) = \lambda \cdot \exp(-\lambda t) = \frac{1}{m} \cdot \exp\left(-\frac{t}{m}\right)$$

DF:

$$P(T \le t) = F(t) = 1 - \exp(-\lambda t) = 1 - \exp\left(-\frac{t}{m}\right)$$

Expected value:

$$E[T] = \frac{1}{\lambda} = m$$

Variance:

$$\mathrm{VAR}[T] = \frac{1}{\lambda^2} = m^2$$

Coefficient of variation:

$$c_T = 1$$

LST: $$\Phi(s) = \frac{\lambda}{\lambda + s} = \frac{1}{1 + ms}$$

Class: `NegExpDistribution`

Constructor: `NegExpDistribution(double mean)`

Parser example:
```
[...].Distribution = NegExp
[...].Distribution.Mean = 3.6
```

4.4.4 Erlang k Distribution

Meaning: Distribution for the sum of k random variables that are each negative-exponentially distributed with the parameter λ (serial switch in the phase model).

k phases

Parameters:

- order $k > 0$
- rate $\lambda > 0$ of the individual phases or total mean value $m = \frac{k}{\lambda}$

PDF: $$P(T = t) = f(t) = \lambda \cdot \frac{(\lambda t)^{k-1}}{(k-1)!} \cdot \exp(-\lambda t)$$

DF: $$P(T \le t) = F(t) = 1 - \exp(-\lambda t) \cdot \sum_{i=0}^{k-1} \frac{(\lambda t)^i}{i!}$$

Expected value: $$E[T] = \frac{k}{\lambda} = m$$

Variance: $$\mathrm{VAR}[T] = \frac{k}{\lambda^2} = \frac{m^2}{k}$$

Coefficient of variation: $$c_T = \frac{1}{\sqrt{k}} \le 1$$

LST: $$\Phi(s) = \left(\frac{\lambda}{\lambda + s}\right)^k = \left(\frac{1}{1 + ms}\right)^k$$

Class: `ErlangDistribution`

Constructor: `ErlangDistribution(double mean, int order)`

Parser example:
```
[...].Distribution = Erlang
[...].Distribution.Mean = 4.5 # k/lambda
[...].Distribution.Order = 3 # number of phases (k)
```

4.4.5 Hypoexponential Distribution to the Order of k

Meaning: Generalization of the Erlang k distribution (serial switching of k phases with negative-exponentially distributed durations with individual parameters λ_i in the phase model)

Parameters:

- order $k > 0$
- rates λ_i or mean values m_i of the individual phases ($i = 1, \ldots, k$)

PDF: $P(T = t) = f(t) = g_1(t) \otimes \ldots \otimes g_k(t)$ with $g_i(t) = \lambda_i \cdot \exp(-\lambda_i t)$

Expected value:

$$E[T] = \sum_{i=1}^{k} \frac{1}{\lambda_i}$$

Variance:

$$\mathrm{VAR}[T] = \sum_{i=1}^{k} \frac{1}{\lambda_i^2}$$

Coefficient of variation:

$$c_T = \frac{\sqrt{\sum_{i=1}^{k} \frac{1}{\lambda_i^2}}}{\sum_{i=1}^{k} \frac{1}{\lambda_i}} \leq 1$$

LST:

$$\Phi(s) = \prod_{i=1}^{k} \frac{\lambda_i}{\lambda_i + s}$$

Class: `HypoExpDistribution`

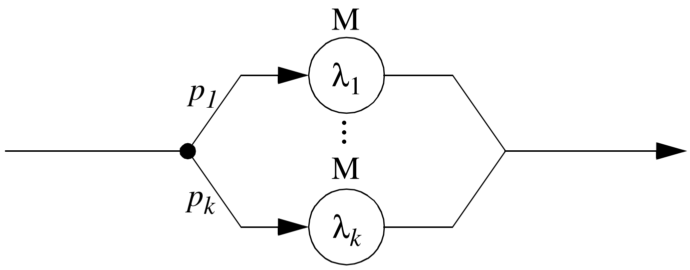
Constructor: `HypoExpDistribution(double[] means)`

Parser example:

```
[...].Distribution = HypoExp
[...].Distribution.Order = 2
[...].Distribution.Means = [ 2.5 , 8]
```

4.4.6 Hyperexponential Distribution to the Order of k

Meaning: Selecting one of k random variables that are negative-exponentially distributed with the individual parameters λ_i and the probabilities p_i (parallel switching in the phase model).



Parameters:

- order $k > 0$
- rates λ_i or mean values m_i of the individual phases ($i = 1, \ldots, k$)
- branching probabilities p_i ($i = 1, \ldots, k$)

PDF:

$$P(T = t) = f(t) = \sum_{i=1}^{k} \lambda_i \cdot p_i \cdot \exp(-\lambda_i t)$$

DF:

$$P(T \le t) = F(t) = 1 - \sum_{i=1}^{k} p_i \cdot \exp(-\lambda_i t)$$

Expected value:

$$E[T] = \sum_{i=1}^{k} \frac{p_i}{\lambda_i}$$

Variance:

$$\mathrm{VAR}[T] = 2\sum_{i=1}^{k} \frac{p_i}{\lambda_i^2} - \left(\sum_{i=1}^{k} \frac{p_i}{\lambda_i}\right)^2$$

Coefficient of variation:

$$c_T = \sqrt{\frac{2\sum_{i=1}^{k} \frac{p_i}{\lambda_i^2}}{\left(\sum_{i=1}^{k} \frac{p_i}{\lambda_i}\right)^2} - 1} \ge 1$$

LST:

$$\Phi(s) = \sum_{i=1}^{k} p_i \cdot \frac{\lambda_i}{\lambda_i + s}$$

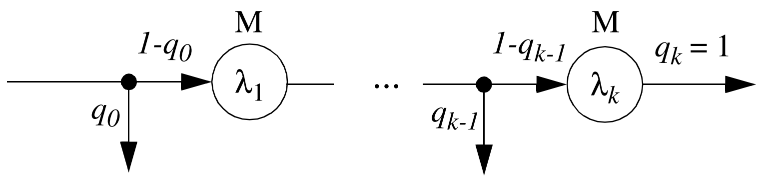
Class: `HyperExpDistribution`

Constructor: `THyperExpDistribution(double[] means, double[] branch-Probs)`

Parser example:
```
[...].Distribution = HyperExp
[...].Distribution.Order = 2
[...].Distribution.Means = [ 2.5 8 ]
[...].Distribution.BranchProbabilities = [ 0.4 0.6 ]
```

4.4.7 Coxian Phase Model

Meaning: Distribution according to the Coxian phase model: Serial switching of a selection of one of k phases each with a negative-exponentially distributed phase duration period (parameter λ_i), whereby after each phase i the system is exited with the probability q_i. Both the hyperexponential as well as the hypoexponential distributions are contained within this model.



Parameters:

- order $k > 0$
- rates λ_i or mean values m_i of the individual phases ($i = 1, \ldots, k$)
- exit probabilities q_i ($i = 0, \ldots, k-1$)

LST:

$$\Phi(s) = q_0 + \sum_{i=1}^{k} \left(\prod_{\nu=0}^{i-1} (1 - q_\nu) \right) \cdot q_i \cdot \prod_{j=1}^{i} \frac{\lambda_j}{\lambda_j + s}$$

Class: `CoxianDistribution`

Constructor: `CoxianDistribution(double[] means, double[] quitProbs)`

Parser example:
```
[...].Distribution = Coxian
[...].Distribution.Order = 2
[...].Distribution.Means = [ 2.5 8 ]
[...].Distribution.QuitProbabilities = [ 0.2 0.5 ]
```

4.4.8 General Distribution

Meaning: A distribution with a certain mean value μ and coefficient of variation c is generated by linking two phases (phase model)

- case I ($c = 0$):
  Only one phase with a constant distribution (mean value μ).
- case II ($0 < c < 1$):
  Serial switching of a constant distribution with the mean value $m_1 = \mu \cdot c$ and a negative-exponential distribution with the mean value $m_2 = \mu \cdot (1 - c)$
- case III ($c = 1$):
  Only one phase with a negative-exponential distribution (mean value μ).
- case IV ($c > 1$):
  Parallel switching of two phases with negative-exponential distributions and the parameters
  $m_{1/2} = \dfrac{\mu}{1 \pm \sqrt{(c^2 - 1)/(c^2 + 1)}}$ (mean value) and
  $p_{1/2} = \frac{1}{2} \cdot (1 \pm \sqrt{(c^2 - 1)/(c^2 + 1)})$ (branching probabilities).

  This corresponds to a hyperexponential distribution of 2nd order, its parameters fulfill the symmetry condition $p_1 \cdot m_1 = p_2 \cdot m_2$.

Parameters:

- mean value $\mu > 0$
- coefficient of variation $c \geq 0$

PDF:

- case I: $P(T = t) = f(t) = \delta(t - \mu)$
- case II: $P(T = t) = f(t) = \dfrac{1}{m_2} \cdot \exp\left(-\dfrac{t - m_1}{m_2}\right)$
- case III: $P(T = t) = f(t) = \dfrac{1}{\mu} \cdot \exp\left(-\dfrac{t}{\mu}\right)$
- case IV: $P(T = t) = f(t) = \dfrac{p_1}{m_1} \cdot \exp\left(-\dfrac{t}{m_1}\right) + \dfrac{p_2}{m_2} \cdot \exp\left(-\dfrac{t}{m_2}\right)$

Expected value: $E[T] = \mu$

Variance: $\mathrm{VAR}[T] = (c \cdot \mu)^2$

LST:

- Case I: $\Phi(s) = \exp(-\mu s)$
- Case II: $\Phi(s) = \dfrac{\exp(-m_1 s)}{1 + m_2 \cdot s}$
- Case III: $\Phi(s) = \dfrac{1}{1 + \mu \cdot s}$
- Case IV: $\Phi(s) = \dfrac{p_1}{1 + m_1 \cdot s} + \dfrac{p_2}{1 + m_2 \cdot s}$

Class: `GeneralDistribution`

Constructor:

```
GeneralDistribution(double mean, double coefficientOfVari-
ation);
```

Parser example:

```
[...].Distribution = General
[...].Distribution.Mean = 2.3
[...].Distribution.CoefficientOfVariation = 1.5
```

4.4.9 Normal Distribution

Meaning: Limit distribution of the sum of many independent random variables with arbitrary statistical characteristics, if the contribution of a single random variable remains neglectably small.

Parameters:

- mean value μ
- standard deviation $\sigma > 0$

PDF:

$$P(T = t) = f(t) = \frac{1}{\sqrt{2\pi}\sigma} \cdot \exp\left(-\frac{(t-\mu)^2}{2\sigma^2}\right)$$

DF:

$$P(T \le t) = F(t) = \frac{1}{2} \cdot erf\left(\frac{t-\mu}{\sqrt{2}\sigma}\right) \text{ with } erf(x) = \frac{2}{\sqrt{\pi}} \cdot \int_0^x \exp(-y^2) \cdot dy$$

Expected value: $E[T] = \mu$

Variance: $\mathrm{VAR}[T] = \sigma^2$

Coefficient of variation: $c_T = \dfrac{\sigma}{\mu}$

LST:

$$\Phi(s) = \exp\left(-\mu s + \frac{(s\sigma)^2}{2}\right)$$

Class: `NormalDistribution`

Constructor: `NormalDistribution(double mean, double standardDeviation)`

Parser example:

```
[...].Distribution = Normal
[...].Distribution.Mean = 6.3
[...].Distribution.StandardDeviation = 1.5
```

4.4.10 Lognormal Distribution

Meaning:

- Distribution of $T = \exp(Z)$, if Z is normally distributed with the parameters μ and σ^2.
- Limit distribution of the product of many independent random variables with arbitrary statistical characteristics if the contribution of a single random variable remains very small.
- In its form similar to a gamma or Weibull distribution

Parameters:

- μ (mean value of Z)
- $\sigma > 0$ (standard deviation of Z)

Alternative parameters mean value and coefficient of variation:

- $\mu = \ln\left(\frac{E[T]}{\sqrt{1 + c_T^2}}\right)$
- $\sigma = \sqrt{\ln(1 + c_T^2)}$

PDF:

$$P(T = t) = f(t) = \frac{1}{\sqrt{2\pi}\sigma \cdot t} \cdot \exp\left(-\frac{(\ln t - \mu)^2}{2\sigma^2}\right) \quad \text{for } t > 0$$

DF: No closed form

Expected value:

$$E[T] = \exp\left(\mu + \frac{\sigma^2}{2}\right)$$

Variance:

$$\mathrm{VAR}[T] = \exp(2\mu + \sigma^2) \cdot (\exp(\sigma^2) - 1)$$

Coefficient of variation:

$$c_T = \sqrt{\exp(\sigma^2) - 1}$$

Class: `LognormalDistribution`

Constructor: `LognormalDistribution(double my, double sigma)`

Parser example:

```
[...].Distribution = Lognormal
[...].Distribution.My = 0.11
[...].Distribution.Sigma = 1.27
```

or with the mean value and variation coefficient:

```
[...].Distribution = Lognormal
[...].Distribution.Mean = 2.5
[...].Distribution.CoefficientOfVariation = 2.0
```

4.4.11 Weibull Distribution

Meaning: The Weibull distribution is often used to model internet traffic because of its heavy tail.

Parameters:

- shape parameter $\alpha > 0$
- scale parameter $\beta > 0$

PDF:

$$P(T = t) = f(t) = \alpha \cdot \beta^{-\alpha} \cdot t^{\alpha - 1} \cdot \exp\left(-\left(\frac{t}{\beta}\right)^{\alpha}\right) \quad \text{for } t > 0$$

DF:

$$P(T \le t) = F(t) = 1 - \exp\left(-\left(\frac{t}{\beta}\right)^{\alpha}\right) \quad \text{for } t > 0$$

Expected value:

$$E[T] = \frac{\beta}{\alpha} \cdot \Gamma\left(\frac{1}{\alpha}\right)$$, whereas $\Gamma(x)$ is the gamma function

Variance:

$$\mathrm{VAR}[T] = \frac{\beta^2}{\alpha} \cdot \left\{2\Gamma\left(\frac{2}{\alpha}\right) - \frac{1}{\alpha} \cdot \Gamma\left(\frac{1}{\alpha}\right)^2\right\}$$

Coefficient of variation:

$$c_T = \sqrt{\frac{2\alpha\Gamma\left(\frac{2}{\alpha}\right)}{\left(\Gamma\left(\frac{1}{\alpha}\right)\right)^2} - 1}$$

Class: `WeibullDistribution`

Constructor: `WeibullDistribution(double alpha, double beta)`

Parser example:

```
[...].Distribution = Weibull
[...].Distribution.Alpha = 0.3
[...].Distribution.Beta = 0.03
```

4.4.12 Gamma Distribution

Meaning: The gamma distribution may, e.g., be applied to characterize video traffic [6]. Special cases contained are the negative-exponential and Erlang k distribution ($\alpha = 1$ or $\alpha = k$).

Parameters:

- shape parameter $\alpha > 0$
- scale parameter $\beta > 0$

The distribution can be alternatively described with the parameters mean value and coefficient of variation:

- $\alpha = \frac{1}{c_T^2}$
- $\beta = \mathrm{E}[T] \cdot c_T^2$

PDF:

$$P(T = t) = f(t) = \frac{\beta^{-\alpha} \cdot t^{\alpha - 1} \cdot \exp(-t/\beta)}{\Gamma(\alpha)} \quad \text{for } t > 0$$

whereby $\Gamma(x)$ is the gamma function

DF: exists only if α is an integer number and positive (see Erlang k distribution)

Expected value: $E[T] = \alpha\beta$

Variance: $\mathrm{VAR}[T] = \alpha\beta^2$

Coefficient of variation: $c_T = \frac{1}{\sqrt{\alpha}}$

Class: `GammaDistribution`

Constructor: `GammaDistribution(double alpha, double beta)`

Parser example:

```
[...].Distribution = Gamma
[...].Distribution.Alpha = 0.44
[...].Distribution.Beta = 10.13
```

or with the mean value and variation coefficient:

```
[...].Distribution = Gamma
[...].Distribution.Mean = 4.5
[...].Distribution.CoefficientOfVariation = 1.5
```

4.4.13 Beta Distribution

Meaning:

- Distribution of a random variable $T = Z_1/(Z_1 + Z_2)$, if Z_1 and Z_2 are gamma distributed with the parameters α_1 and β as well as α_2 and β.
- suitable for the distribution of random shares (values between 0 and 1)
- Special cases contained are the uniform distribution between 0 and 1 as well as several triangular distributions

Parameters:

- shape parameter $\alpha_1 > 0$
- shape parameter $\alpha_2 > 0$

The distribution can also be described with parameters for the mean value (in the range between 0 and 1) and coefficient of variation:

- $\alpha_1 = \dfrac{1 - E[T] \cdot (1 + c_T^2)}{c_T^2}$
- $\alpha_2 = \alpha_1 \cdot \left(\dfrac{1}{E[T]} - 1\right)$

PDF:

$$P(T = t) = f(t) = \frac{t^{\alpha_1 - 1} \cdot (1 - t)^{\alpha_2 - 1}}{B(\alpha_1, \alpha_2)} \quad \text{for } 0 < x < 1,$$

whereby $B(x, y)$ is the Beta function

DF: No closed form of the distribution function (except for special cases)

Expected value:

$$E[T] = \frac{\alpha_1}{\alpha_1 + \alpha_2}$$

Variance:

$$\mathrm{VAR}[T] = \frac{\alpha_1 \cdot \alpha_2}{(\alpha_1 + \alpha_2)^2 \cdot (\alpha_1 + \alpha_2 + 1)}$$

Coefficient of variation:

$$c_T = \sqrt{\frac{\alpha_1}{\alpha_2 \cdot (\alpha_1 + \alpha_2 + 1)}}$$

Class: `BetaDistribution`

Constructor: `BetaDistribution(double alpha1, double alpha2)`

Parser example:

```
[...].Distribution = Beta
[...].Distribution.Alpha1 = 1.5
[...].Distribution.Alpha2 = 3.0
```

or with the mean value and coefficient of variation:

```
[...].Distribution = Beta
[...].Distribution.Mean = 0.33
[...].Distribution.CoefficientOfVariation = 0.60
```

4.4.14 Pareto Distribution

Meaning: Like the Weibull distribution, the Pareto distribution is often used to characterize Internet traffic because of its heavy tail. The superposition of on-off sources (see Sections 4.6.1 and 4.6.2) with heavy-tailed phases is known to produce self-similar traffic.

Parameters:

- minimum value $k > 0$
- shape parameter $\alpha > 0$

In case $\alpha > 2$ this distribution can also be described with parameters mean value μ and coefficient of variation c:

- $\alpha = 1 + \frac{\sqrt{1 + c^2}}{c}$
- $k = \frac{1}{\frac{c}{\sqrt{1 + c^2}} + 1} \cdot \mu$

PDF:

$$P(T = t) = f(t) = \frac{\alpha \cdot k^{\alpha}}{t^{\alpha + 1}} \quad \text{for } t \geq k$$

DF:

$$P(T \leq t) = F(t) = 1 - \left(\frac{k}{t}\right)^{\alpha} \quad \text{for } t \geq k$$

Expected value:

$$E[T] = \begin{cases} \frac{\alpha}{\alpha - 1} \cdot k & \text{for } \alpha > 1 \\ \infty & \text{for } \alpha \leq 1 \end{cases}$$

Variance:

$$\mathrm{VAR}[T] = \begin{cases} \left(\frac{\alpha}{\alpha - 2} - \left(\frac{\alpha}{\alpha - 1}\right)^2\right) \cdot k^2 & \text{for } \alpha > 2 \\ \infty & \text{for } \alpha \leq 2 \end{cases}$$

Coefficient of variation: $c_T = \frac{1}{\sqrt{(\alpha - 2) \cdot \alpha}}$ for $\alpha > 2$

Class: `ParetoDistribution`

Constructor: `ParetoDistribution(double alpha, double minValue)`

Parser example:

```
[...].Distribution = Pareto
[...].Distribution.Alpha = 2.2
[...].Distribution.MinValue = 1.0
```

or with any of the following combinations:
(Mean OR MinValue) AND (CoefficientOfVariation OR Alpha), e.g.,

```
[...].Distribution = Pareto
[...].Distribution.Mean = 1.83
[...].Distribution.CoefficientOfVariation = 1.51
```

4.4.15 Jakes Distribution

Meaning: The class `JakesDistribution` implementes a continuous distribution which delivers random variables in accordance with Jake's Doppler power density spectrum. Jake's power density spectrum is often used to model the distribution of Doppler shifts in a fading channel. The distribution delivers random variables according to Jake's spectrum between -Fdmax and +Fdmax. For an introduction to this topic and the inverse of the cdf see for example

P. Hoeher: Kohaerenter Empfang trelliscodierter PSK-Signale auf frequenzselektiven Mobilfunkkanaelen - Entzerrung, Decodierung und Kanalparameterschaetzung, VDI Fortschrittsberichte, Reihe 10, Nr. 147 (in German).

or

P. Hoeher: A Statistical Discrete-Time Model for the WSSUS Multipath Channel, IEEE Transactions on Vehicular Technology, Vol. 41, No. 4, pp. 461-468, November 1992.

Parameters: maximum Doppler shift $f_{D,\max}$

PDF:

$$P(T = t) = f(t) = \frac{1}{\pi \cdot f_{D,\max} \cdot \sqrt{1 - (t/f_{D,\max})}} \quad \text{for } |t| < f_{D,\max}$$

DF:

$$P(T \le t) = F(t) = \frac{1}{2} + \frac{1}{\pi} \cdot \operatorname{asin}\left(\frac{t}{f_{D,\max}}\right) \quad \text{for } |t| < f_{D,\max}$$

Expected value: $E[T] = 0$

Variance:

$$\mathrm{VAR}[T] = \frac{1}{2}(f_{D,\max})^2$$

Coefficient of variation: not applicable

Class: `JakesDistribution`

Constructor: `JakesDistribution(double fdmax)`

Parser example:
```
[...].Distribution = JakesDistribution
[...].Distribution.Fdmax = 100.0 # Hz
```

4.4.16 Trace Distribution

Meaning: All values are read from an input file passed to the distribution

Parameters: depends on trace file

PDF: depends on trace file

DF: depends on trace file

Expected value: depends on trace file

Variance: depends on trace file

Coefficient of variation: depends on trace file

Class: `TraceDistribution`

Constructor: `TraceDistribution(String fileName)`

Parser example:
```
[...].Distribution = Trace
[...].Distribution.source = <filename>
```

4.4.17 Empirical Distribution

Meaning: distribution function is passed to constructor as vector or parsed from file
c.f. detailed comments in header file description

Parameters: distribution function

PDF:

DF: passed to constructor as vector or parsed from file

Expected value: depends on DF

Variance: depends on DF

Coefficient of variation: depends on DF

Class: `EmpiricalDistribution`

Constructor: `EmpiricalDistribution(double[] vector)`

Parser example:

```
[...].Distribution = EmpiricalDist
# F(x0) must be 0.0, x0 arbitrary
# F(xn) must be 1.000, xn > x0
[...].Distribution.CDF    =    [0.000    0.01000000    1.000
0.05000000 70.000 0.95000000 110.000 1.00000000 ]
```

4.5 Modified Distributions

This section describes distributions that are generated by executing operations on other distributions. They are either derived from `ContinuousDistribution` or `DiscreteDistribution`.

4.5.1 Continuous Distribution with a Discrete Value Range

Meaning:

- Distribution of a real random variable $T = d \cdot N$, whereby N represents an arbitrarily distributed discrete random variable and d the scale factor.
- Main application is the description of a cell distance in a time slot system e.g., on an ATM link. In this case d is the time slot duration.
- A special case of the compound distribution with a constant "inner distribution".

Parameters:

- scale factor/time slot duration d
- discrete distribution p_i of N / the number of time slots

PDF:

$$P(T = t) = f(t) = \sum_{n=0}^{\infty} p_n \cdot \delta(t - nd)$$

Expected value: $E[T] = E[N] \cdot d$

Variance: $\mathrm{VAR}[T] = \mathrm{VAR}[N] \cdot d^2$

Coefficient of variation: $c_T = c_N$

LST: $\Phi(s) = H(\exp(-sd))$, if $H(z)$ represents the generating function of N.

Class: `SlottedDistribution`

Constructor: `SlottedDistribution(DiscreteDistribution noOfSlotsDist, double slotDuration)`

Parser example:

```
[...].Distribution = SlottedDistribution
[...].Distribution.SlotDuration = 1.5
[...].Distribution.NoOfSlotsDist = Geometric
[...].Distribution.NoOfSlotsDist.Mean = 9
```

4.5.2 Piece-wise Linear Distribution

Meaning:

- Distribution of a real random variable $T = d \cdot (Z + Y)$, whereby Z is an arbitrarily distributed integer number random variable, Y is an uniformly distributed continuous random variable between 0 and 1 and d is the scale factor.
- Application example: empirical distribution according to [15], whereby the distribution of Z is a known histogram

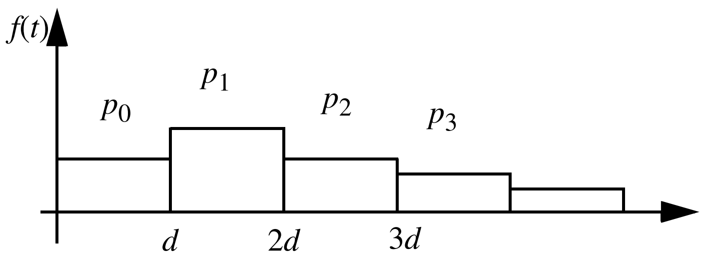
Parameters:

- scale factor/time slot duration d
- discrete distribution p_i of Z / the number of time slots

PDF:

$$P(T = t) = f(t) = \frac{1}{d} \cdot \sum_{n=0}^{\infty} p_n \cdot (\sigma(t - nd) - \sigma(t - (n+1)d))$$

whereby $\sigma(x)$ represents the step function.



Expected value:

$$E[T] = \left(E[N] + \frac{1}{2}\right) \cdot d$$

Class: `PiecewiseLinearDistribution`

Constructor: `PiecewiseLinearDistribution(DiscreteDistribution noOf-SlotsDist, double slotDuration)`

Parser example:

```
[...].Distribution = PiecewiseLinearDistribution
[...].Distribution.SlotDuration = 1.5
[...].Distribution.NoOfSlotsDist = Geometric
[...].Distribution.NoOfSlotsDist.Mean = 9
```

4.5.3 Nested Distribution

Meaning: The nested distribution gives the distribution of a sum of random variables $T_1, T_2, \ldots, T_N$ that each are described by a continuous („inner“) distribution. The number of addends N itself is a random variable with discrete („outer“) distribution. In literature, this distribution is also denoted as compound distribution.

Parameters:

- inner distribution with PDF $g(t)$ and DF $G(t)$
- outer distribution p_i

PDF:

$$P(T = t) = f(t) = \sum_{n=0}^{\infty} p_n \cdot (g_1(t) \otimes \ldots \otimes g_n(t)) \quad \text{with} \quad g_i(t) \equiv g(t)$$

$\forall i$

Expected value: $E[T] = E[N] \cdot E[T_i]$

Variance: $\mathrm{VAR}[T] = \mathrm{VAR}[T_i] \cdot E[N] + \mathrm{VAR}[N] \cdot (E[T_i])^2$

Coefficient of variation:

$$c_T = \sqrt{\frac{c_{T_i}^2}{E[N]} + c_N^2}$$

LST: $\Phi(s) = H(\Psi(s))$, if $H(z)$ represents the generating function of the external and $\Psi(s)$ the LST of the inner distribution.

Class: `NestedDistribution`

Parser example:

```
[...].Distribution = NestedDistribution
[...].Distribution.InnerDist = Constant
[...].Distribution.InnerDist.Mean = 2.3
[...].Distribution.OuterDist = Geometric
[...].Distribution.OuterDist.Mean = 9
```

4.5.4 Continuous Bounded Distribution

Meaning:

Continuous distribution of a random varaible T that is bounded between a lower bound and an upper bound. For the bounding of T, two different modes are available:

1. *with resampling*: If the random number generator returns a value that is smaller than the lower boundary or greater than the upper boundary, a new random number is drawn (this is done as long as a value between lower and upper boundary is obtained).
2. *without resampling*: If the random number generator returns a value that is smaller than the lower boundary or greater than the upper boundary, then the lower respectively upper boundary is return.

Parameters::

- base distribution with PDF $g(t)$ and DF $G(t)$
- lower limit b_l
- upper limit b_u
- boolean value *resampling*.

Class: `BoundedDistribution`

Parser example:

```
[...].Distribution = BoundedDistribution
[...].Distribution.BaseDist = Normal
[...].Distribution.BaseDist.Mean = 5.0
[...].Distribution.BaseDist.StandardDeviation = 4.0
[...].Distribution.LowerBound = 0.0 # optional, default 0
[...].Distribution.UpperBound = 10.0 # optional, def. inf.
[...].Distribution.Resampling = false # optional, def. tr.
```

4.5.5 Discrete Bounded Distribution

Meaning:

Discrete analogon to the continuous bounded distribution, whereby the upper and lower limits each belong to the value range of the random variable X.

Parameters:

- base distribution q_i with the corresponding DF Q_i
- lower limit b_l (integer number)
- upper limit b_u (integer number)
- boolean value *resampling*

Distribution:

$$P(X = i) = p_i = \begin{cases} \dfrac{q_i}{Q_{b_u} - Q_{b_l}} & \text{für } b_l \le i \le b_u \\ 0 & \text{sonst} \end{cases}$$

Class: `DiscreteBoundedDistribution`

Parser example:

```
[...].Distribution = DiscreteBoundedDistribution
[...].Distribution.BaseDist = Binomial
[...].Distribution.BaseDist.Mean = 10.0
[...].Distribution.BaseDist.Variance = 5.0
[...].Distribution.LowerBound = 0 # optional, default 0
[...].Distribution.UpperBound = 20 # optional, def. inf.
[...].Distribution.resampling = false # optional, def. tr.
```

4.5.6 Linear Transformed Continuous Distribution

Meaning: Distribution of a random variable T, that results from a linear transformation $T = aZ + b$ of the random variable Z with a given continuous distribution ("base distribution").

Parameters:

- base distribution with PDF $g(t)$ and DF $G(t)$
- factor $a \neq 0$
- offset b

PDF:

$$P(T = t) = f(t) = g\left(\frac{t-b}{a}\right)$$

DF:

$$P(T \leq t) = F(t) = G\left(\frac{t-b}{a}\right)$$

Expected value: $E[T] = a \cdot E[Z] + b$

Variance: $\mathrm{VAR}[T] = a^2 \cdot \mathrm{VAR}[Z]$

Coefficient of variation:

$$c_T = \frac{1}{\frac{1}{c_z} + \frac{b}{a\sqrt{\mathrm{VAR}[Z]}}}$$

Class: `TransformedDistribution`

Parser example:

```
[...].Distribution = TransformedDistribution
[...].Distribution.BaseDist = NegExp
[...].Distribution.BaseDist.Mean = 2.5
[...].Distribution.Factor = 2.0 # optional, default = 1
[...].Distribution.Offset = 10.0 # optional, default = 0
```

4.5.7 Linear Transformed Discrete Distribution

Meaning: Distribution of a discrete random variable X, that results from a linear transformation $X = aY + b$ of the random variable Y with a given discrete distribution ("base distribution").

Parameters:

- base distribution q_i
- integer number factor $a \neq 0$
- integer number offset b

Distribution:

$$P(X = i) = p_i = \begin{cases} q_j & \text{für } i = aj + b, j = 0, 1, \ldots \\ 0 & \text{else} \end{cases}$$

Expected value: $E[X] = a \cdot E[Y] + b$

Variance: $\mathrm{VAR}[X] = a^2 \cdot \mathrm{VAR}[Y]$

Coefficient of variation:

$$c_T = \frac{1}{\frac{1}{c_z} + \frac{b}{a\sqrt{\mathrm{VAR}[Y]}}}$$

Class: `DiscreteTransformedDistribution`

Parser example:

```
[...].Distribution = DiscreteTransformedDistribution
[...].Distribution.BaseDist = Geometric
[...].Distribution.BaseDist.Mean = 2.5
[...].Distribution.Factor = 2 # optional, default = 1
[...].Distribution.Offset = 1 # optional, default = 0
```

4.5.8 Floored Distribution

Meaning: The „floored distribution“ is a discrete distribution that converts the value obtained from a continuous („base“) distribution to an integer value using the `floor()` function from the Java `Math` package. `floor(x)` implements the floor operator $\lfloor x \rfloor$, i.e. it yields the largest integer value not greater than x.

Parameters: continuous base distribution $f(x)$ (DF $F(x)$)

Distribution: $P(X = i) = F(i + 1) - F(i)$

Class: `FlooredDistribution`

Parser example:

```
[...].Distribution = FlooredDistribution
[...].Distribution.BaseDist = NegExp
[...].Distribution.BaseDist.Mean = 2.5
```

4.5.9 Ceiled Distribution

Meaning: The „ceiled distribution“ is a discrete distribution that converts the value obtained from a continuous („base“) distribution to an integer value using the `ceil()` function from the Java `Math` package. `ceil(x)` implements the ceiling operator $\lceil x \rceil$, i.e. it yields the smallest integer value not less than x.

Parameters: continuous base distribution $f(x)$ (DF $F(x)$)

Distribution: $P(X = i) = F(i) - F(i-1)$

Class: `CeiledDistribution`

Parser example:

```
[...].Distribution = CeiledDistribution
[...].Distribution.BaseDist = NegExp
[...].Distribution.BaseDist = Mean = 2.5
```

4.5.10 Rounded Distribution

Meaning: The „rounded distribution“ is a discrete distribution that converts the value obtained from a continuous („base“) distribution to an integer value using the `rint()` function from the Java `Math` package. `rint(x)` returns the integer value nearest to x (e.g., `rint (1.4)` is 1.0 and `rint (1.6)` is 2.0).

Parameters: continuous base distribution $f(x)$ (DF $F(x)$)

Distribution: $P(X = i) = F(i + 1/2) - F(i - 1/2)$

Class: `RoundedDistribution`

Parser example:

```
[...].Distribution = RoundedDistribution
[...].Distribution.BaseDist = NegExp
[...].Distribution.BaseDist.Mean = 2.5
```

4.6 Source Models

The distributions discussed in the previous two chapters can describe processes of the class GI (General Independent). In addition, a variety of state-dependent distributions exist that are mostly used to model arrival processes. They are called source or generator models. They are derived from `ContinuousDistribution` or `DiscreteDistribution`, so that they externally appear the same as continuous or discrete distributions. Yet, the returned values depend on an internal state and are usually correlated because of this. State changes take place e.g. after a certain number of calls.

4.6.1 Talkspurt Silence Source

Meaning:

- Modeling of on-off sources in packet networks
- Modeling of voice sources

Description: see [13], [19]

- State machine with 2 states (on/off, talkspurt/silence or burst/silence)
- In the talkspurt state X cells/packets arrive at intervals of d. After that a silence phase of the duration S takes place.
- Special case of the GMDP, if S is described by a compound distribution with a constant distribution as inner distribution

Parameters:

- (discrete) distribution of the number of cells X in the talkspurt state
- (continuous) distribution of silence duration S
- arrival interval d („inter-cell time“) in the talkspurt state

Class: `TalkspurtSilenceDistribution`

Constructor:

```
TalkspurtSilenceDistribution (
   DiscreteDistribution talkspurtDist,
   ContinuousDistribution silenceDist,
   double interCellTime)
```

Characteristic values

- Peak rate: $h = 1/d$
- Mean rate: $m = \frac{E[X]}{E[X] \cdot d + E[S]}$
- Burstiness: $b = \frac{h}{m} = 1 + \frac{E[S]}{E[X] \cdot d}$
- For given h and m (or h and b as well as $E[X]$:

  $$h = \frac{1}{h},\ E[S] = E[X] \cdot \left(\frac{1}{m} - \frac{1}{h}\right) = \frac{E[X]}{h} \cdot (b - 1)$$

Parser example:

```
[...].Dist = TalkspurtSilenceDistribution
[...].Dist.TalkspurtDistribution = ShiftedGeometric
[...].Dist.TalkspurtDistribution.Mean = 20
[...].Dist.SilenceDistribution = NegExp
[...].Dist.SilenceDistribution.Mean = 800
[...].Dist.SilenceDistribution.InterCellTime = 10
```

4.6.2 On-Off Infinity Model

Meaning: Superposition of infinitely many on-off sources (see also Section 4.6.1) which allows the modelling of aggregated traffic. Bursts of traffic are generated according to an interarrival time distribution. The burst length distribution represents the length in number of packets which are sent within a burst with the specified intercell time.

Usually a negative-exponential distribution is used as burst interarrival time distribution leading to an M/G/∞ source model.

Description: Traffic is characterized by

- burst length distribution
- burst interarrival time
- inter cell time d

Parameters:

- (discrete) distribution of the burst length X in number of packets
- (continuous) distribution of burst interarrival time T_A
- inter-cell time d

Char. values:

- mean (packet interarrival time): $\mu = \dfrac{E[T_A]}{E[X]}$

Class: `OnOffInfinityDistribution`

Constructor:

```
OnOffInfinityDistribution(
  DiscreteDistribution burstLengthDist,
  ContinuousDistribution burstIATDist,
  double interCellTime)
```

Parser example:

```
[...].Dist = OnOffInfinityDistribution
[...].Dist.BurstLengthDist = Geometric
[...].Dist.BurstLengthDist.Mean = 2.3
[...].Dist.BurstIATDist = NegExp
[...].Dist.BurstIATDist.Mean = 2.88
[...].Dist.BurstIATDist.InterCellTime = 1.0
```

4.6.3 GMDP (Generally Modulated Determistic Process)

Meaning: Modeling of sources with multiple states (usually 2 - 5) in ATM networks

Description: A GMDP comprises a state machine with m states (see [13], [19]):

- In state i arrivals occur in constant intervals d_i. If $d_i < 0$ it is a silence state without arrivals. In that case $-d_i$ describes the virtual slot duration in this state.
- X_i representing the number of arrivals in state i is arbitrarily (discretely) distributed (often shifted geometrically -> MMDP); in the silence state X_i means the number of time slots of the silence phase.
- After X_i arrivals incl. an additional interval of the length d_i (or after X_i slots of the length $-d_i$ in the silence state), a transition occurs to the state j with the probability p_{ij}.
- According to definition: $p_{ii} = 0$ and $\sum_{i=1}^{m} p_{ij} = 1$.

Parameters:

- number of states m
- transition probabilities p_{ij}
- distribution of the number of arrivals X_i in the individual states
- arrival intervals d_i in the individual states

Char. values: Moments and distribution of the arrival intervals, see [20]

Class: `GMDPDistribution`

Constructor:

```
GMDPDistribution(
  double interCellTime,
  DiscreteDistribution phaseLengthDist,
  double[] transitionProbs);
```

Parser example:

```
[...].Dist = GMDPDistribution
[...].Dist.States = 3
[...].Dist.PMAP = [ [0 0.5 0.5] [0.5 0 0.5] [0.5 0.5 0] ]
[...].Dist.PhaselengthDistribution_0 = ShiftedGeometric
[...].Dist.PhaselengthDistribution_0.Mean = 10
[...].Dist.PhaselengthDistribution_1 = ShiftedGeometric
[...].Dist.PhaselengthDistribution_1.Mean = 10
[...].Dist.PhaselengthDistribution_2 = ShiftedGeometric
[...].Dist.PhaselengthDistribution_2.Mean = 5
# negative value denotes silence state
[...].Dist.InterCellTimes = [ -1 2 4 ]
```

4.6.4 MMDP (Markov Modulated Deterministic Process)

Meaning: Most frequent special case of the GMDP (Section 4.6.3) for modeling sources with multiple states (generally 2 - 5) in ATM networks

Description: Like the GMDP, the MMDP can be described by a state machine with m states (see Section 4.6.3). Number of arrivals X_i in the state i is distributed according to a shifted geometric distribution (Section 4.3.6).

Char. values: For moments and distribution of the inter-arrival time see [20].

Class: `MMDPDistribution`

Constructor: `MMDPDistribution(int noOfStates, double[] interCellTimes, double[] meanPhaseLengths, double[][] transitionProbs)`

Parameters:

- Number of states m
- Transition probabilities p_{ii}
- Average number of arrivals a_i in the individual states
- Arrival intervals d_i in the individual states

Parser example:

```
[...].Dist = MMDPDistribution
[...].Dist.States = 3
[...].Dist.PMAP = [[0 0.5 0.5] [0.5 0 0.5] [0.5 0.5 0] ]
[...].Dist.MeanPhaseLengths = [145.5 11.11 55.55]
# negative value denotes silence state
[...].Dist.InterCellTimes = [-1 1274 255]
```

4.6.5 GMPP (Generally Modulated Poisson Process)

Meaning: Modeling of sources with multiple states (usually 2 - 5) in ATM networks

Description: See [13], [19]

State machine with m states.

- Markovian process with the rate λ_i in the state i
- number of arrivals X_i in state i is arbitrarily (discretely) distributed (often shifted geometrically -> MMPP); in the silence state X_i means the number of time slots of the silence phase.
- according to definition: $p_{ii} = 0$ and $\sum_{i=1}^{m} p_{ij} = 1$.

Parameters:
- number of states m
- transition probabilities p_{ij}
- distribution of the number of arrivals X_i in the individual states
- arrival rates λ_i in the states

Char. values: Moments and distribution of the arrival intervals

Class: `GMPPDistribution`

Constructor:
```
GMPPDistribution (double meanIAT, ContinuousDistribution
stateDuration, double[] transitionProbs);
```

Parser example:
```
[...].distribution = GMPP
[...].distribution.States = 2
[...].distribution.MeanIATs = [1 5]
[...].distribution.StateDuration_0 = Constant
[...].distribution.StateDuration_0.Mean = 10000
[...].distribution.StateDuration_1 = Constant
[...].distribution.StateDuration_1.Mean = 10000
[...].distribution.TransitionProbs = [[0 1] [1 0]]
```

4.6.6 Continuous-State GMPP

Meaning: Modeling of sources where the state space is continuous.

Description: State machine with continuous (infinite) state space:
- length of a state according to a continuous distribution
- rate r in a state according to a continuous distribution
- Poisson process with rate r within a state

Parameters:
- continuous distribution of the length of a phase
- continuous distribution of the rate in a phase

Char. values: Moments and distribution of the arrival intervals

Class: `ContStateGMPPDistribution`

Constructor:
```
ContStateGMPPDistribution (
  ContinuousDistribution phaseLengthDist,
  ContinuousDistribution rateDist)
```

Parser example:
```
[...].distribution = ContStateGMPP
[...].distribution.PhaseLengthDist = NegExp
[...].distribution.PhaseLengthDist.Mean = 2.3
[...].distribution.RateDist = Uniform
[...].distribution.RateDist.LowerBound = 1.5
[...].distribution.RateDist.UpperBound = 3.5
```

4.6.7 MMPP (Markov Modulated Poisson Process)

Meaning: Modeling of sources with multiple states e.g., at the call level

Description: State machine with m states (see [8], [13], [19]):

- Markovian process with the rate λ_i in the state i
- Markovian process for modulation with the transition rate q_{ij} from state i to state j
- According to definition: $q_{ii} = -\sum_{i \neq i} q_{ij}$
- Special case of the MAP (Section 4.6.8) with $d_{ii} = \lambda_i$, $d_{ij} = 0$ $\forall(i \neq j)$ and $c_{ij} = q_{ij} - d_{ij}$

Parameters:

- number of states m
- arrival rates λ_i in the states
- transition rates q_{ij}

Char. values: see [8]

- sojourn time S_i in the state i is negative exponentially distributed with the mean value $E[S_i] = 1/\sum_{i \neq i} q_{ij} = -1/q_{ii}$
- sojourn probabilities P_i from linear equation system
  $\sum_i q_{ji} \cdot P_j = 0 \ \ \forall i, \ \sum_i P_i = 1$
- average arrival rate: $\lambda = \sum_i P_i \cdot \lambda_i$

Class: `MMPPDistribution`

Constructor:

```
MMPPDistribution(int noOfStates,
  double[] eventRates,
  double[][] transitionRates)
```

Parser example:

```
[...].distribution = MMPPDistribution
[...].distribution.States = 2
[...].distribution.EventRates = [0.1 0.9]
[...].distribution.RMAP = [[-0.001 0.001] [0.001 -0.001]]
```

4.6.8 MAP (Markovian Arrival Process)

Meaning: Modeling of sources with multiple states e.g., at the call level

Description: See [13], [16], [19]

- State machine with m states
- Continuous Markovian process with the transition rate q_{ij} from state i to state j, which is composed of two components: $q_{ij} = c_{ij} + d_{ij}$
- Transition rate from state i to state j ($j \neq i$) without an arrival event upon transition: c_{ij}
- Transition rate from state i to state j with an arrival event upon transition: d_{ij}
- According to definition: $c_{ii} = -\sum_{i \neq i} c_{ij} - \sum_{i} d_{ij} \Rightarrow q_{ii} = -\sum_{i \neq i} q_{ij}$

Parameters:

- number of states m
- transition rates without arrival c_{ij}
- transition rates with arrival d_{ij}

Char. values: See [16]

Class: `MAPDistribution`

Constructor:

```
MAPDistribution(int noOfStates,
  double[][] c,
  double[][] d)
```

Parser example:

```
[...].distribution = MAPDistribution
[...].distribution.States = 2
[...].distribution.CMAP = [[-0.26 0.05] [0.05 -0.47]]
[...].distribution.DMAP = [[0.2 0.01] [0.02 0.4]]
```

4.6.9 DMAP (Discrete time Markovian Arrival Process)

Meaning: Modeling from sources with multiple states

Description: See [1], [13]

- State machine with m states
- Discrete Markovian process with the transition probability p_{ij} from state i to state j after each time slot Δt, which is a combination of two components: $p_{ij} = c_{ij} + d_{ij}$
- Transition probability from state i to state j without an arrival event upon transition: c_{ij}
- Transition probability from state i to state j with an arrival event upon transition: d_{ij}
- According to definition: $\sum_i p_{ij} = 1$

Parameters:

- number of states m
- transition probabilities without arrival c_{ij}
- transition probabilities with arrival d_{ij}

Char. values: See [1]

Class: `DMAPDistribution`

Constructor:

```
DMAPDistribution(int noOfStates,
  double[][] c,
  double[][] d)
```

Parser example:

```
[...].distribution = DMAPDistribution
[...].distribution.States = 2
[...].distribution.CMAP = [[0.2 0.5] [0.1 0.3]]
[...].distribution.DMAP = [[0.2 0.1] [0.2 0.4]]
```

4.6.10 Video Source

Meaning: Modeling of video sources in ATM/packet networks

Description: See [17]

- Parameters: frame duration d_F, frame size s_F, packet size s_P
- Autoregressive process with the bit rate λ_i in frame i in bits/pixel: $\lambda_i = (a \cdot \lambda_{i-1} + b \cdot \varepsilon) \cdot r(0, \lambda_{max})$ with the constants a and b as well as the function r to limit the rate to the interval $[0, \lambda_{max}]$.
- Usually normal distribution for ε

Parameters:

- frame size s_F
- frame duration d_F
- start rate λ_0
- constant a
- constant b
- maximum rate λ_{max}
- distribution ε
- packet size s_P

Char. values:

- Average packet intervals within the frame i: $E[T] = \frac{s_P \cdot d_F}{s_F \cdot \lambda_i}$

Class: `VideoSourceDistribution`

Constructor:

```
VideoSourceDistribution(int framesize,
  double frameduration, double startrate,
  double constanta, double constantb,
  double upperlimitrate,
  ContinuousDistribution ratedistribution,
  int packetsize)
```

Parser example:

```
[...].dist = VideoSourceDistribution
[...].dist.StartRate = 0.5333333 # bits/pixel
[...].dist.UpperlimitRate = 1.0666667 # max. bits/pixel
[...].dist.RateDistribution = Normal
[...].dist.RateDistribution.Mean = 0.587 # bits/pixel
[...].dist.RateDistribution.StandardDeviation = 1
[...].dist.FrameDuration = 33333 # inter-picture time
[...].dist.FrameSize = 250000 # 250 000 pixel/picture
[...].dist.PacketSize = 384 # ATM cell 48 * 8 bits
[...].dist.ConstantA = 0.8781 # factor for last rate
[...].dist.ConstantB = 0.1108 # factor for new rate
```

4.6.11 WSS (Wide Sense Stationary Process)

Meaning: Noise generator (conversion of white noise to bandwidth-limited, colored noise with the help of a filter)

Description:

- White noise at the input of a filter is uniformly distributed in the interval $(-\sqrt{3}, \sqrt{3})$.
- Behavior is defined by n filter coefficients $h_0, \ldots, h_{n-1}$ as well as the mean value μ.
- Condition for positive event intervals: $\sum_{i=0}^{n-1} |h_i| \cdot \sqrt{3} < \mu$

Parameters:

- mean value μ
- filter coefficients h_i $(i = 0, \ldots, n-1, n > 0)$

Char. values: auto covariance sequence: $R_i = \sum_{k=0}^{n-1-k} h_k \cdot h_{k+i}$ $(i = 0, \ldots, n)$

Class: `WSSDistribution`

Constructor:

```
WSSDistribution(double mean,
  double[] filterCoefficients)
```

Parser example:

```
[...].distribution = WSSDistribution
[...].distribution.Mean = 2
[...].distribution.Coefficients = [0.2 0.3 -0.05]
```

5 Statistics

5.1 Statistics Base Class

All statistics classes are derived from the abstract base class `Statistic` that is characterized by the following:

- reacts to the signals *init simulation*, *start simulation*, *stop simulation*, *start transient phase*, and *stop transient phase*.
- defines an interface of purely abstract functions
  - `resetStatistic()` causes a complete reset of the statistics at the beginning of a simulation as well as after a warm-up phase
  - `resetBatchStatistic()` causes a reset of counters, etc. at the beginning of a batch
  - `computeMeasures()` calculates a partial spot test from the measured values of a batch
- has a `SimNode` to support the exporting, importing, and printing of results (Section 9).

Generally, measured values are acquired with the help of statistics during a batch (e.g. with the function `update()` which often exists in derived classes). These results are used to calculate an intermediate result with `computeMeasures()` at the end of a batch depending on the statistics type. At the end of all batches the individual intermediate results are dealt with as samples. Normally, a mean value is composed from these samples and the confidence interval is determined. The confidence interval is an indication of the statistical significance of the determined value. It is defined as an interval, in which the actual value lies with the probability q (default value $q = 0{,}95$). In order to determine the confidence interval the student-t distribution is applied [7][15].

Depending on the type of measured values, different procedures for registration and evaluation are available. Several classes have been derived from the base class `Statistic`, which define these measurement procedures. There is one class for each of these special statistics, that defines the interface for evaluation (e.g., `SampleStatistic`) and another that implements the evaluation functions (e.g., `StdSampleStatistic`).

The following auxiliary classes are available for the individual statistics:

- `Summation` for the registration and evaluation of measured values or samples from the batches e.g., mean value, variance, standard deviation and confidence interval of the mean values.
- `Range` to determine the smallest and largest mean value during a batch or the complete simulation
- `EstimationManager`, `StatisticEstimation`, `Student`, `StudentSearch`, `StudentCalc`, and `StudentMixed` to determine the confidence interval (calls occur from `Summation` automatically)

Note: In earlier versions, a global statistics manager (class `StatisticManager`), which contained the appropriate `create<*>Statistic()` and `delete<*>Statistic()` methods, was used to create an object of a special statistics class. The interface of the statistics manager was used to assure that the code of the model components remained unchanged, should a statistics type (e.g., `SampleStatistic`, Section 5.2) be realized by a class not belonging to the pre-defined standard statistics class (e.g., `StdSampleStatistic`).

5.2 Statistic Class for Sample Mean and Variance

The classes `SampleStatistic` or `StdSampleStatistic` serve to register single measured values (e.g. run times) of the type `double` with the help of the method `update()`.

The following functions are available for evaluation:

- Mean value of all returned measured values (`getMean()`) including a confidence interval (`getMeanConfidenceInterval()`)
- Standard deviation of all returned measured values (`getDeviation()`) including a confidence interval (`getDeviationConfidenceInterval()`)
- Coefficient of variation of all returned measured values (`getCoefficient()`) including a confidence interval (`getCoefficientConfidenceInterval()`)
- Value range (`getMinimum()` and `getMaximum()`) in reference to all measured values
- Mean value of the returned measured values from the current batch (`getBatchMean()`)
- Coefficient of variation of the returned measured values from the current batch (`getBatchCoefficient()`)
- Variation of the returned measured values from the current batch (`getBatchVariation()`)
- Number of measured values from the batch (`getBatchEvents()`)

The following key words have been defined in `SampleStatistic` to print the results with formats (Section 9):

mean	Mean value of all measured values in reference to all batches
cintmean	Confidence interval of the mean values in +/- notation
deviation	Standard deviation in reference to all batches
cintdeviation	Confidence interval of the standard deviation in +/- notation
cov	Coefficient of variation in reference to all batches
cintcov	Confidence interval of the coefficients of variation in +/- notation
min	Smallest measured value in all batches
max	Greatest measured value in all batches
numberofevents	Number of measured values in the current batch
bmin	Minimum in reference to the current batch
bmax	Maximum in reference to the current batch
bmean	Mean value in reference to the current batch

bvariance	Variance in reference to the current batch
bcov	Coefficient of variation in reference to the current batch

5.3 Weighted Mean Statistic

The class `WeightedMeanStatistic` is simillar to `SampleStatistic` (Section 5.2) with the difference that in the `Update()` method not only a measured value but also a weight is passed. The default value for that weight is 1. Based on that difference, only first order statistic are calculated.

The following functions are available for evaluation:

- mean value of all measures `getMean()`
- mean value of all measures in a batch `getBatchMean()`
- confidence interval of the mean value `getMeanConfidenceInterval()` :

Results can be printed using the following keywords within a print format (see Section 9):

mean	Mean value of all measured values in reference to all batches
cintmean	Confidence interval of the mean values in +/- notation
bmean	Mean value in reference to the current batch

5.4 Counter Statistics

The classes `CounterStatistic` or `StdCounterStatistic` add up measured values (e.g., number of messages that have passed a certain port) of the type `long` during a batch with the help of the method `update()`.

- The following functions are available for evaluation:
- Mean value of batch sums (`getMean()`) including a confidence interval (`getConfidenceInterval()`)
- Value range (`getMinimum()` and `getMaximum()`) of the batch sums
- Sum of all measured values in all batches (`getNumberOfEvents()`)
- Sum of all measured values in the current batch (`getNumberOfBatchEvents()`)
- Number of batches (`getNumberOfBatches()`)

The following key words have been defined in `CounterStatistic` to print the results using print formats (Section 9):

mean	Mean value of the registered values per batch
cintmean	Confidence interval of the mean values with +/- notation
bmin	Minimum of the number of registered values in a batch

bmax Maximum of the number of registered values in a batch

bnumberofevents Sum of the registered values in the last batch

sum Sum of all registered values

numberofbatches Number of all past batches

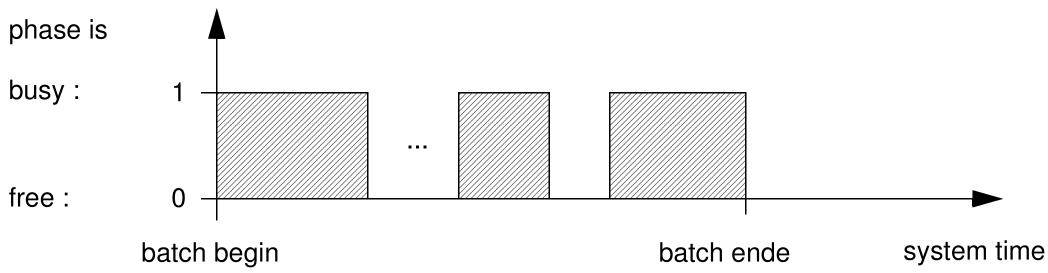


Figure 5.1: Utilization of a service unit

5.5 Integral Statistics

The classes `IntegralStatistic` or `StdIntegralStatistic` serve to record measured values, whose absolute value is not the only important factor, but also the duration of the value's appearance is of importance. Determining the utilization factor of a service unit (Fig. 5.1) or the average length of a queue are just a few examples.

The emphasis of time duration corresponds to integral creation by adding up the integral sections, that are created by each call of `update()` as a product of the returned measured value of the last `update()` call (with declaration of the system time) and the intermediate time interval. At the end of a batch this integral value is divided by the elapsed time since batch begin in order to obtain the mean value.

The following functions are available for evaluation:

- Time-emphasized mean value across all batches (`getMean()`) including a confidence interval (`getConfidenceInterval()`)
- Value range (`getMinimum()` and `getMaximum()`) of returned measured values across all batches
- Time-emphasized mean value in reference to the current batch (`getBatchArea()`)

The following key words have been defined in `IntegralStatistic` to print the results with formats (Section 9):

mean Mean value of all batch integrals each divided by its own batch duration

cintmean Confidence interval of the mean values in +/- notation

min Smallest of all returned measured values during simulation

max	Greatest of all returned measured values during simulation
barea	Integral sum of the current batch divided by the duration of the batch

5.6 Probability Statistics

The class `ProbabilityStatistic` serves to record the results of Bernoulli trials, which are random experiments with only two possible results (e.g., failure probability in a queue with limited capacity). `update()` returns the number of trials that have the same result (`true` or `false`).

The following functions are available for evaluation:

- Batch probability as a quotient of the number of favorable events and the total number of events (`getBatchProbability()`)
- Mean value of the batch probabilities (`getMean()`) including a confidence interval (`getConfidenceInterval()`)

The above mentioned functions, that have only been declared in a virtual manner in `ProbabilityStatistic`, have been implemented in the derived class `StdProbabilityStatistic`. An object of this class can be created with the `createProbabilityStatistic()` method from `StatisticManager` (Section 5.1) and deleted with `deleteProbabilityStatistic()`.

The following key words have been defined in `ProbabilityStatistic` to print the results with formats (Section 9):

mean	Mean value of the probabilities in all batches
cintmean	Confidence interval for the mean value in +/- notation
bprob	Probability in reference to the current batches

5.7 Conditional Mean Statistics

5.7.1 Bucket Utility

`BucketUtility` and its derived classes `LinBucketUtility`, `LogBucketUtility`, and `GeoBucketUtility` are utility classes used by conditional statistics. Upon construction, they require the parameters *minimum value*, *maximum value* and *number of buckets*. From these parameters, they subdivide the range of values in the given number of buckets of equal size and logarithmic/geometric growing size, respectively. Furthermore, they also control the access (e. g., by `update()`) to these buckets

5.7.2 Conditional Mean Statistic

`CondMeanStatistic` is derived from `StdSampleStatistic` and offers the possibility to obtain a mean value statistic conditioned on a value (conditioner). The calculation of the bucket widths and the access to the buckets is controlled by a derived class of `BucketUtility`. `CondMeanStatistic` has three constructors: one using `LinBucketUtility`, one using `LogBucketUtility` and constructor using default and either `LinBucketUtility` or `LogBucketUtility`, depending on the value set as default for *meanBucketSize*. Upon construction the parameters *minimum value*, *maximum value* and *number of buckets* are required for the first two constructors and are passed to a derived class of `BucketUtility`. As `LogBucketUtility` additionally requires a mean value for the bucket size, this parameter is also required upon construction of `CondMeanStatistic` using buckets of logarithmis growing size. The third constructor needs no special parameters besides *name* and *owner*.

The following functions are available for evaluation:

- Value of mean `getMean(int)`
- Value of standard deviation `getDeviation(int)`
- Value of coefficient of variation of all batches `getCoefficient(int)`
- Value of confidence interval to coefficient of variation `getMeanConfidenceInterval(int)`
- Value of confidence interval of standard deviation `getDeviationConfidenceInterval(int)`
- Value of confidence interval of coefficient of variance `getCoefficientConfidenceInterval(int)`
- Value of maximum sample of all batches `getMaximum(int i)`
- Value of minimum sample of all batches `getMinimum(int i)`

The following key words have been defined in `CondMeanStatistic` to print the results with formats (Section 9):

ubound	Upper bound of the respective bucket
lbound	Lower bound of the respective bucket
bucketmean	Mean value of the respective bucket
cintbucket	Confidence interval to mean value in +/- notation

5.7.3 Conditional Probability Statistic

`CondProbabilityStatistic` is derived from `StdProbabilityStatistic` and offers the possibility to obtain a probability statistic conditioned on a value (conditioner). The calculation of the bucket widths and the access to the buckets is controlled by a derived class of `BucketUtility`. `CondProbabilityStatistic` has three constructors: one using `LinBucketUtility`, one using `LogBucketUtility` and constructor using default and either `LinBucketUtility` or `LogBucketUtility`, depending on the value set as default for *meanBucketSize*. Upon construction the parameters *minimum value*, *maximum value* and *number of buckets* are required for the first two constructors and are passed to a derived class of `BucketUtility`. As `LogBucketUtility` additionally requires a mean value for the bucket size, this parameter is also required upon construction of `CondProbabilityStatistic` using buckets of logarithmis growing size. The third constructor needs no special parameters besides *name* and *owner*.

The following functions are available for evaluation:

- Value of mean `getMean(int)`
- Value of coefficient of variation of all batches `getCoefficient(int)`

ubound	Upper bound of the respective bucket
lbound	Lower bound of the respective bucket
bucketmean	Mean value of the respective bucket
cintmean	Confidence interval to mean value in +/- notation

5.8 Distributions Statistics

The classes `DistributionStatistic` or `StdDistributionStatistic` serve to record measured values with the goal of determining a discrete distribution (histogram). The measured values recorded with `update()` are qualified using a raster, that is composed of an upper limit value and a lower limit value as well as the number of partial sections in between. The parameters for the raster must be declared upon constructor call. Additional to a measured value, a weight (with default 1) that is multiplyed with the value can be passed with `Update()`.

The following functions are available for evaluation:

- Number of partial sections (`getArraySize()`, generally fixed)
- Total number of returned measured values (`getNumberOfEvents()`)
- Over-flow probability in reference to all registered measured values with confidence interval (`getOverflowProbability()` or `getOverflowConfInterval()`)
- Under-flow probability in reference to all registered measured values with confidence interval (`getUnderflowProbability()` or `getUnderflowConfInterval()`)
- Value of discretized distribution density functions at a certain point (range index) in the form of probabilities that a measured value falls into a certain partial range (`getProbability()` or `getProbabilityConfInterval()` including a confidence interval)
- Value of discretized distribution functions at a certain point (range index) as a sum of probabilities for the areas below that point (`getDistribution()` or `getDistributionConfInterval()` with a confidence interval)
- Output of the distribution density function as a block graphic (`displayProbability()`)
- Output of the distribution function as a block graphic (`displayDistribution()`)
- Number of the returned measured values in the current batch (`getNumberOfBatchEvents()`)
- Over-flow probability in the current batch (`getBatchOverflowProb()`)
- Under-flow probability in the current batch (`getBatchUnderflowProb()`)

- Value of the discretized distribution density function at a certain point in reference to the current batch (`getBatchProbability()`)
- Value of the discretized distribution function at a certain point in reference to the current batch (`getBatchDistribution()`)

The following keywords have been defined in `DistributionStatistic` to print the results with formats (Section 9):

uprob	under-flow probability in reference to all batches
cintuprob	confidence interval of the under-flow probability
oprob	over-flow probability in reference to all batches
cintoprob	confidence interval of the over-flow probability
list	each bucket is printed using the keywords defined below
ubound	print upper bound of the current bucket (to be used after keyword *list*)
dist	print value of the discretized cumulative distribution function (cdf) referring to current bucket (to be used after keyword *list*)
cdist	print value of the discretized complementary cumulative distribution function (ccdf) referring to current bucket (to be used after keyword *list*)
prob	print value of the discretized probability density function (pdf) referring to current bucket (to be used after keyword *list*)
cintdist	print confidence interval for cdf/ccdf value referring to current bucket (to be used after keyword *list*)
cintprob	print confidence interval for pdf value referring to current bucket (to be used after keyword *list*)
numberofevents	number of measured values during the simulation
buprob	under-flow probability for the current batch
boprob	over-flow probability for the current batch
bdist	print value of the discretized cumulative distribution function (cdf) referring to current bucket for current batch (to be used after keyword *list*)
bprob	print value of the discretized probability density function (pdf) referring to current bucket for current batch (to be used after keyword *list*)

5.9 Rate Statistics

The class `RateStatistic` serves to record rates, i.e. the amount of information per time as well as the number of samples per time. The `update()` function takes a `double` value indicating the amount of information. Besides `name` and `owner`, an entity is required for constructor call to have access to the method `getSystemTime()`.

The following functions are available for evaluation:

- Mean value in [units/time] of all returned measured rates (`getRate()`) including a confidence interval (`getRateConfInterval()`)
- Mean value in [samples/time] of all returned measured rates (`getSampleRate()`) including a confidence interval (`getSampleRateConfInterval()`):

rate	Mean of all rates [units/time] in all batches
cintrate	Confidence interval to mean rates
samplerate	Mean of all rates [samples/time] in all batches
cintsamplerate	Confidence interval to mean rates

5.10 Median Statistics

The classes `MedianStatistic` or `StdMedianStatistic` serve to register single measured values (e.g., run times) of the type `double` with the help of the method `update()`.

The following functions are available for evaluation:

- Median value of all returned measured values (`getMedian()`) including a confidence interval (`getMedianConfidenceInterval()`)
- Value range (`getMinimum()` and `getMaximum()`) in reference to all measured values
- Number of measured values from the batch (`getBatchEvents()`)

If the parameter *maxBufferSize* is set to zero, all samples will be stored internally and the median will be calculated at the end of the simulation. Since this is very memory consuming, *maxBufferSize* can be set to a number > 0, which determines the maximum number of samples the statistic should hold in memory. The median is then calculated at the end of the simulation as an approximation that is based on all samples.

The following key words have been defined in `MedianStatistic` to print the results with formats (Section 9):

median	Median value of all measured values in reference to all batches
cintmean	Confidence interval of the mean values in +/- notation
min	Smallest measured value in all batches
max	Greatest measured value in all batches
bmedian	Number of measured values in the current batch

The calculation of the median usually needs to buffer all measured values. In order to save memory an approximation algorithm is served by the median statistic class:

- all values are saved in a list of value pairs of one double (value) and one integer value (weight)
- the weight is set to 1 for every new value added to the list

- when the list size exceeds the maximum buffer size, all entries are sorted by the value and grouped to pairs by adding their weight and calculating like this:
  value = value1 · weight1 + value2 · weight2
- When the method `getMedian()` is called the list is sorted again by value. Then the list is iterated from beginning until the sum of the weight-values is larger than the half of the total sum of weights. After this the found item and the previous item are selected and an linear approximation between the two value pairs is done.

5.11 Boundary Statistics

`BoundaryStatistic` is an abstract base class derived from `Statistic`. It is the base class of `StdBoundaryStatistic` and `JainBoundaryStatistic`. These two classes are used to find the quantile for a given percentage. The class `StdBoundaryStatistic` provides an implementation to calculate quantiles for a given number of samples and percentage. The class `JainBoundaryStatistic` does the same with an alternative method, the P^2 algorithm described in [9]. Thus, for constructor call, `StdBoundaryStatistic` requires additionaly to the percentage (e.g., 0.95) also the expected number of samples.

The following functions are available for evaluation:

- `getQuantile()` returns the value of the quantile.
- `getPercentage()` returns the value of the input percentage.
- `getNoOfSamples()` returns the number of all collected samples.

The following keywords have been defined in `BoundaryStatistic` to print the results with formats (Section 9):

quantile	value found to be the quantile
prob	input percentage
numberofevents	number of events in all batches

A special case of a quantile is the median, which is the 50%-quantile. The dedicated `MedianStatistic` has been implemented to measure medians (see Section 5.10).

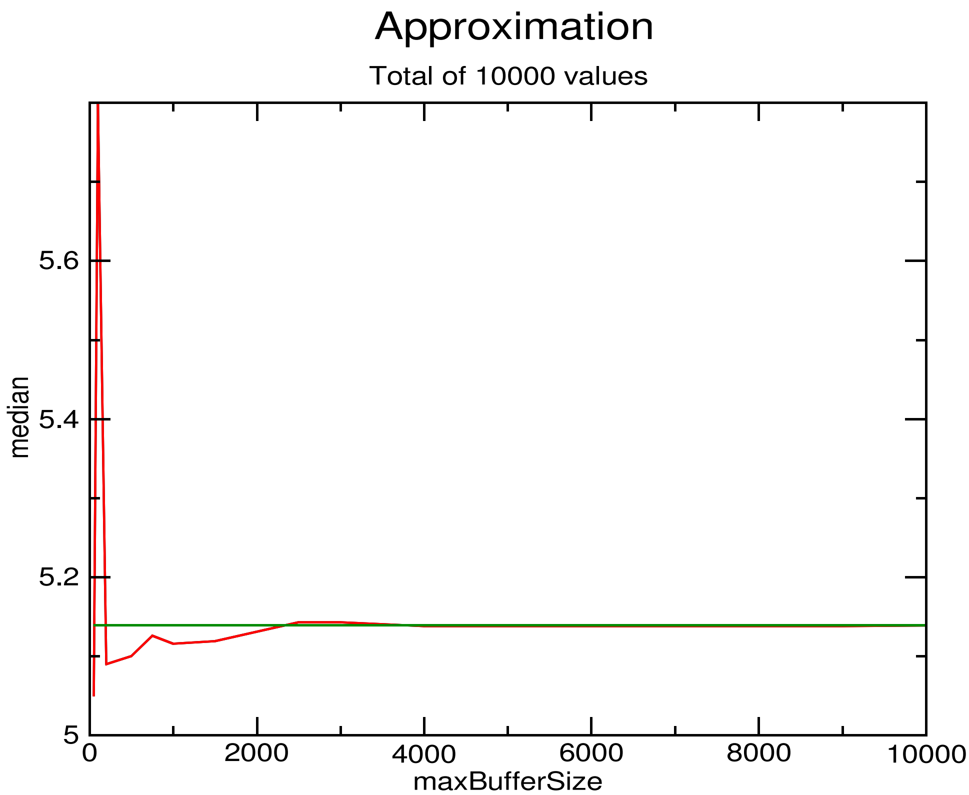


Figure 5.2: Accuracy of the median algorithm

The graph in Fig. 5.2 shows the accuracy of the median approximation in comparison to the accurate value when define the statistic with negative exponentially distributed values. A total of 10,000 samples have been used in this test.

Tests have shown that a MaxBufferSize of 10-20% results a deviation to the accurate value about 0.01%.

6 Model Components

A major advantage of the IKR Simulation Library is the easy construction of a model from model components (e.g. queues, service units) that are then connected together with the help of the port concept.

6.1 General Classes

6.1.1 Hierarchical Models: Class SimNode

The IKR SimLib supports the building of hierarchical models. Therefore, there is an interface to build a model hierarchy, which is called `SimNode`. The top node of the hierarchy belongs to an object of the class `Model`. All other nodes of the hierarchy can belong to objects of arbitrary type.

The interface `SimNode` has the following set of methods:

- building the model hierarchy:
  - `addChild()` adds a child node to an existing node
  - `getChildren()` returns all child nodes of this node
  - `getParent()` returns the parent node
- naming nodes:
  - `getName()` returns the name of this node, only
  - `getFullName()` returns the fully qualified name of this node
- exporting, importing, and printing results:
  - `getPrintHandler()` return a handler for printing results
  - `getExportHandler()` returns a handler for importing and exporting results

There is a default implementation of a `SimNode` called `StdSimNode`, which already implements the methods listed above.

Every simulation program has one single instance of the class `SimNodeManager`. This object knows the root of the model hierarchy and can traverse the hierarchy for purpose of printing.

6.1.2 Model Component Base Class: Class Entity

Model components can be derived from the class `Entity`, that has the following characteristics:

- The following functions support the port concept with its controlled exchange of messages:
  - `addPort()` adds a port to this entity
  - `aliasPort()` adds a alias port, which is basically a pointer to a port object
  - `unaliasPort()` removes a port alias
  - `isPortKnown()` queries, if a port has already been registered

Within the simulation library, several model components already exist which have been derived from `Entity`. These classes will be introduced in the following sub-chapters.

6.1.3 Complete Model: Class Model

At the top of the hierarchically ordered model components, there is a model component derived from `Model`. The class `Model` has the following characteristics:

- The function `processPendingEvents()`, which is generally called by sequence control upon begin of each batch, processes events in a loop by calling the calendar method `processNextEvent()` (Section 2).
- The function `stopProcessingEvents()`, which is generally called by sequence control at the end of each batch, indirectly causes the termination of the loop in `processPendingEvents()` by setting the flag `done`, which is checked after each loop run, to true.

6.2 Traffic End Points

6.2.1 Generator architecture

Generators create messages and thus represent traffic sources. In order to provide for flexibility, a 3-level architecture is implemented or generation of messages.

At the top level, the so called *Generator level*, there are classes derived from the abstract class `Generator`, which are connected to the other components of the simulation model using the port concept. Subclasses of `Factory<Message>` serve the generators at the level in the middle. The bottom level is formed by the different kinds of messages which are generated by the message factories.

6.2.1.1 Generator

The class `Generator` is derived from `Entity` and is the abstract base class for all generators.

`Generator` provides the basic functionality for all derived generators:

- The generation of messages is controlled by events.
- Each time an event has been processed, a new event is registered.
- The creation of a message is performed by calling `createMessage()`, which forces an aggregated object derived from the class `Factory<Message>` (see Section 6.2.1.2) to create a new message. Almost all details of message generation are therefore hidden within the message factory. The term message is used as a synonym for all possible representations of a messages, which are derived from `Message`. For further description of messages see Section 7.1.
- It is possible to set and get message factories during any phase of simulation by calling `setMessageFactory()` and `getMessageFactory()`. Several entities (generators) can share the same message factory.
- For each generator class, there is a corresponding parser class.

StdGenerator

`StdGenerator` is a very flexible Generator with the follwing characteristics:

- Messages are generated according to the interarrival time distribution.
- Message length follows an optional discrete distribution. Each time a new message is generated by a MessageFactory, the length is adapted by calling `setLength()` of the message and passing the new length according to the length distribution.
- If no length distribution is provided, the length of the generated message is not changed. Therefore control of the length remains at the MessageFactory.
- A message factory is required and could either be provided by passing it upon generation of `StdGenerator` or during the parsing procedure.
- All possible combinations of message factories and distributions are described in the source code.

BurstGenerator

The second subclass of `Generator` is `BurstGenerator`. This class generates bursts, which are segmented into messages. Herefore, the distribution of the burst length and the burst interarrival time as well as the message interarrival time and the maximum message length are required.

TraceGenerator

`TraceGenerator` generates messages at timestamps, which are given in a trace file. Further the length of the message is provided by the trace file. The type of the message depends on the default message handled by the according message factory. `TraceGenerator` has two helper classes, namely `TraceFile` and `TraceItem`. The trace file must have have following format:

hh:mm:ss.sss length

The time information can either be absolute or relative to the preeceding event (= interarrival time), this can be set by the boolean attribute `absoluteTime` in the constructor, or in the parameter file when the generator is parsed. Further the behaviour at the end of the trace file can be determined. If `wrapFile` is set to true, the trace file is reread from the beginning, but in the case of absolute time only the IAT between two succeeding events is considered. In the other case the simulation interrupts with an error, if the end of the trace file is reached. Comments are initiated by a leading '#'.

GreedyGenerator

The `GreedyGenerator` generates as many messages as the attached component can handle. That is, as soon as a message has been fetched, the next one is generated and offered. The greedy generator only works with an active receiver attached to it. If a distribution for the length of a message is provided/parsed it sets the message length according to the given distribution. In this case, the length of the default Message generated by the MessageFactory is overwritten. A message factory may be provided to create messages other than `Message`.

6.2.1.2 MessageFactory

The class `MessageFactory` has been introduced in order to make message generation more flexible. This is granted by the standardized methods of the interface `Factory<Message>`. Whenever a new message is needed by a generator the method `createMessage()` is invoked. This provides the possibility to adapt the creation of messages to required demands.

FlexIPMessageFactory

`FlexIPMessageFactory` is more complex and less flexible regarding the messages.

- Only messages of type `IPPacket` are supported.
- Four distributions are required. These distributions are adapting the header entries *SourceId*, *DestinationId*, *GroupId* and *TypeId*.
- Parsing is also supported, the keywords are: *IPPacket*, *SourceIdDist*, *DestinationIdDist*, *GroupIdDist*, *TypeIdDist* and *Tag*

UniqueIDIPMessageFactory

`UniqueIDIPMessageFactory` is different from `IPPacketFactory` as follows:

- it assigns a unique ID to each MessageFactory and sets one field of IPPackets accordingly. A static class variable is used to count the number of message factories and use this as ID at time of construction of the object. This can be used to mark flows.
- it has a variable `field` which determines the field of IPPackets, set with the unique ID. The following values for `field`, which are provided by parsing or in the constructor, are possible: *SourceId, DestinationId, GroupId* or *TypeId.*

Hint

The functionality of the deprecated classes `DistLengthGenerator`, `LabelDis-tLengthGenerator` and `IPDistLengthGenerator` is completly implemented by `StdGenerator`. Further `FlexIPGenerator`, could be replaced by `StdGenerator` together with `FlexIPMessageFactory`.

6.2.2 Sink

The task of a traffic sink is to delete messages on the heap at the end of a chain of passing model components. For this purpose the class `Sink` has been derived from the `Entity` and is implemented with the help of `InputMessageHandler`. It communicates with the preceding model component via the input port (class `InputPort`) (Section 7).

6.3 Service Components

6.3.1 Server

The abstract base class `Phase` derived from `Entity`. The class `StdPhase` are used to model service units. The service time of `UnitPhase` (unit refers here to information unit) is proportional to the length specified in `Message` (Section 7.1) and therefore models a link with a fixed bandwidth. The service time of StdPhase is obtained independent of the length of a packet by a distribution. In the following, the detailed characteristics are described:

- Phase is implemented with the help of Event (Section 2) and InputMessageHandler (Section 7), private derivation.
- The time intervals of the events, that represent the end of service are determined with the help of a continuous distribution (Section 4), that must be delivered as a reference with StdPhase upon constructor call.
- In StdPhase the distribution can be read by the parser (Section 10).
- State of the phase (free or busy) decides if arriving messages at the input port can be accepted or not (class InputPort, Section 7).
- Upon end of service the serviced message is sent to the succeeding service unit via the output port (class SynchronousOutputPort, Section 7), whereby blocking must be avoided.

The class MultiPhasePrioServer derived from Entity realizes a service unit with multiple service phases, that may have different priorities. Interruptions are possible whereby unit behavior is controlled by the global interruption interval as well as the interruption strategy of the individual phases.

- The method createPhase() from MultiPhasePrioServer creates an internal service phase of the type ServerPhase as well as a pair of input and output ports (ServerInputPort and SynchronousOutputPort) that are assigned to this phase.
- The parameters from createPhase() define the characteristics of the phase that will be generated: name, priority, service time distribution function, interruption strategy, queueing strategy.
- The names of the input and output ports are "<Phasename>In" and "<Phasename>Out".
- All phases within a server with the same priority are assigned to a queue of the type ServerPriorityQueue.

6.3.2 Queue

Queues that are based on the base class Queue which is derived from Entity can store any kind of message. Queue has two special derivatives, namely FIFOQueue and LIFOQueue. The first one represents a FIFO queue (First In First Out), wheras the latter one represents a LIFO queue (Last In First Out). Both the FIFO and the LIFO queue have derived classes for an unbounded and a bounded version (BoundedFIFOQueue, UnboundedFIFOQueue, BoundedLIFOQueue and UnboundedLIFOQueue).

The lengths of a queue is measured in units (information units, e. g. bytes). A unit corresponds to the length specified in the member length of Message and is also the basis to determine whether a bounded queue is full. Nevertheless, the length of a queue in number of packets is also remembered by List<> in which the messages are stored. The different kinds of lengths can be retrieved by getCurrentNumberOfUnits() and getCurrentNumberOfMessages(), respectively.

- The communication with the preceding or succeeding model component is processed via an input port (class InputPort) and a message handler of the type

StdInputMessageHandler or via an output port (class OutputPort) and a message handler of the type StdOutputMessageHandler (Section 7).

- In FIFOQueue and LIFOQueue messages that arrive at the input port are buffered including their arrival time with the method put() (if there is enough room in the queue).
- Simultaneously a message for the succeeding model component is offered.
- If there is not enough buffer space in the queue to store the currently arriving message, a BoundedFIFOQueue will drop the message, wheras a BoundedLIFOQueue will drop as many messages from the head of the queue as necessary to store the arriving message. In case it is larger than the maximum allowed queue size, it is dropped immediately without deleting any messages from the queue.
- There are two methods which can be used to react to offers of the preceding or requests of the succeeding model component: isEmpty() and isFull().
- Upon request by the succeeding model component the method get() is used to retrieve the next message from the queue.
- Statistics are kept on
  - the waiting time in reference to waiting messages,
  - the waiting time in reference to all passing messages (transfer time),
  - the queue length in number of units,
  - the queue length in number of messages, and
  - the loss probability.

6.3.3 Single Server Queue

SingleServerQueue implements a queue and a server in one component and thus offers more flexibility than two separated components and the ability for more sophisticated statistics.

SingleServerQueue is an abstact base class for single servers. It provides a non-blocking input port ("input") and a non-blocking (synchronous) output port ("output"). Messages received on the input port are forwarded to the method receiveMessage() which has to be defined in derived classes. Moreover, it contains an end of service event which is handled by HandleEndOfService that has to defined in sublasses.

StdSingleServerQueue is derived from SingleServerQueue. It is still an abstract class as it does not define methods ReceiveMessage and HandleEndOfService. It contains statistic objects measuring server occupancy as well as variables for service time (per length unit) and buffer size which are read as parameters. The corresponding keywords are *ServiceTime*, *ServiceRate*, and *BufferSize*. If *UseStatistics* is set to false the statistic is not used (may be usefull to increase performance).

FIFODropTailQueue is a subclass of StdSingleServerQueue and defines the ReceiveMessage method called on message arrival as well as HandleEndOfService. Incoming messages are immediately processed if the server is free or stored in a queue if the server is busy. When the server has completed service the message at the head of the queue enters service.

keyword	description	type	default
serviceRate	per byte service rate (i.e. service time = msg. length / service rate)	`DOUBLE`	∞
bufferSize	maximum value for sum of lengths of messages in buffer; a value less than 0 corresponds to an infinite buffer size.	`INTEGER`	-1 (∞)
useStatistics	create and update statistics for occupancy (mean), queue length (mean) and loss probability	`BOOLEAN`	true
dequeueFor-Service	if this parameter is set to true a message dequeued when entering service; otherwise the message remains in the buffer until end of service	`BOOLEAN`	true
traceQueueLength	print a trace of the queue length (current time and current queue length)	`BOOLEAN`	false
traceOccupancy	print a trace of the system occupancy (current time and 0 or 1)	`BOOLEAN`	false
busyPeriodStat	If this keyword occurs the parameters of a distribution statistic for busy period evaluation can be defined in a following {} block	`OBJECT`	no busy period statistics

Table 6.1: Keywords for `FIFODropTailQueue`

`CanEnqueue` returns true if the total message length including the message that has just arrived is <= the buffer size which can be specified in the input file.

Like its base classes, `FIFODropTailQueue` can read its parameters from an input stream using the parser described in Section 10. As additional keywords *TraceQLength*, *Trace-Occupancy, DequeueForService*, and *BusyPeriodStat*. The keywords for `FIFODropTailQueue` including those defined in base classes can be are explained in Table 6.1.

If *UseStatistics* is not set to false statstics are kept on

- occupancy
- packet loss probability
- mean queue length
- mean busy period (if *BusyPeriodStat* has been specified)
- busy period distribution (if *BusyPeriodStat* has been specified)

6.3.4 Further Components

Other components that will not be described in further detail are named here:

- `MultiPhasePrioServer`: service unit with priorities

- InifiniteServer and DInfiniteServer: Incoming messages are delayed by a time value which follows an arbitrary or a constant distribution, respectively.
- ClockedGate: synchronized "gate"

6.4 Connector Components

6.4.1 Multiplexer

The base class Multiplexer and its derived classes StdMultiplexer, and PriorityMultiplexer represent components with multiple input ports (StdInputPort) and one output port (OutputPort). Except from PriorityMultiplexer, all inputs are scanned in a round-robin scheduling algorithm. In StdMultiplexer the initial number of input ports is given on construction. Inputs are named: "input 1" ... "input N". Further ports may be added via addPort() and removed via removePort().

- Arriving messageIndication() signals are directly delivered to the output port.
- If the recipient calls isMessageAvailable(), the request will be passed on to the input port, which cyclically follows the input port from which the last message was retrieved ("fairness").
- StdMultiplexer provides the methods addPort() and removePort() to create and delete an input port named "input 1", "input 2", ...
- PriorityMultiplexer is derived from StdMultiplexer and overwrites the method isMessageAvailable() in that way that it will always start its port scan with the first input port. Although this prioritization of the input ports is only valid in "active receiver" mode.

6.4.2 Demultiplexer

The class Demultiplexer is the counter part to Multiplexer and is the base class for StdDemultiplexer, LabelDemultiplexer, and IPDemultiplexer. An arriving message at the input port (InputPort) is passed on to one of the output ports (StdOutputPort) which is determined by calling the method getOutputPort(). Whereas the class StdMultiplexer does not provide an implementation of the method getOutputPort(), the classes LabelDemultiplexer, and IPDemultiplexer are ready-to-use. In StdDemultiplexer the initial number of output ports is given on construction. Outputs are named: "output 1" ... "output N". Further ports may be added via addPort() and removed via removePort().

- The decision which message will reach which port is made with the method getOutputPort(), that must be defined in the derived classes. In order for this decision to be consistent throughout the lifetime of the demultiplexer, LabelDemultiplexer and IPDemultiplexer are not allowed to use removePort().
- LabelDemultiplexer implements the method etOutputPort() by returning the port with the index label of a LabelMessage (contained in the data member label of LabelMessage) minus the start label (contained in startLabel). The start label can be given as parameter on construction and defaults to 0.

- `IPDemultiplexer` is a ready-to use static demultiplexer for packets belonging to the class `IPPacket` and destinguishes between the modes „destination-based“, „group-based“ and „type-based“. The functionallity is the same as described in `LabelDemultiplexer` except that not the label is taken into consideration but destination, group or type, depending on the mode that is set. Therefore, additional to the parameters described for `StaticLabelDemux`, a parameter indicating the mode has to be provided upon construction.
- `Expander` (Section 6.4.3) can also be used to define a demultiplexer. The output ports of this demultiplexer are not allowed to be blocked.

6.4.3 Forking and Branching

The template `Expander` is meant to be a base classe for components with one input port and several output ports. Each of these output ports can be assigned an attribute (its type is defined by the template parameter).

- Upon construction, `Expander` requires the number of ports to be created. The created ports are named "output 1", "output 2", ... However, new ports can be added with the help of the method `addPort()` using the port name and its attribute. With `removePort()` an output port can be removed.
- The attributes can be changed subsequently using `setPortParameter()`, which requires port identification with its index (1, 2,... , n).
- The purely virtual method `handleMessageIndication()` must be defined in derived classes.
- The type used for the output ports in both classes is `SynchronousOutputPort`, i.e. no blocking may occur at an output port.

The classes `Branch` and `BinaryBranch` serve to realize random branching in the flow of messages.

- The class `Branch` derived from `Expander<double>`. It receives the number of ports upon constructor call. The branching probability is assigned to the port (with index = 1, 2,..., n) with the method `setProbability()`. Furthermore, it has a method `addPort()`, which can add an output port with a certain name and assign a branching probability to it. With `removePort()` the port can be removed.
- `BinaryBranch` has only two output ports ("output 1" and "output 2"). The branching probability of the first port is delivered upon constructor call and can be subsequently changed with `setFirstProbability()`.
- The branching probability of all branch classes can be modified with `equalize()` and `normalize()` in such a way, that they are all the same (1/n) or that their sum equals 1.
- During operation start `Branch` will check if the sum of probabilities equals 1 and if necessary an exception is thrown.

The classes `Fork` and `BinaryFork` serve to split the flow of messages. One or more copies of an arriving message at the input port are sent to each output port.

- In `Fork` the number of output ports can de defined upon constructor call. The number of messages sent to a port is initialized with 1.

- `Fork` contains the methods `addPort()` and `removePort()`. In `addPort()` the port name and the number of messages to be sent to that port can be defined (default: 1).
- With `setNoOfMessages()` the number of message copies that are sent to an output port can be modified.
- The class `BinaryFork` represents a light version of a fork component, that only has two output ports ("output 1" and "output 2"). The number of copies made is fixed at 1.
- The virtual method `cloneMessage()` is used in all fork components to create copies of arriving messages. This method calls the method `clone()` from `Message`. Therefore it is important to define `clone()` for each message class when using a fork component.

6.5 Compound Model Components

6.5.1 Sets of Generators

`GeneratorSet` contains a set of several different generators, which are multiplexed onto one output port. This is useful to model a complex traffic source, which generates different kinds of traffic, e.g., different interarrival time distribution and different of messages types. Usually not all Generators are different, therefore it is useful to group equal Generators, at least to simplify the creation of Generators. This is implemented by the class `GeneratorGroup`.

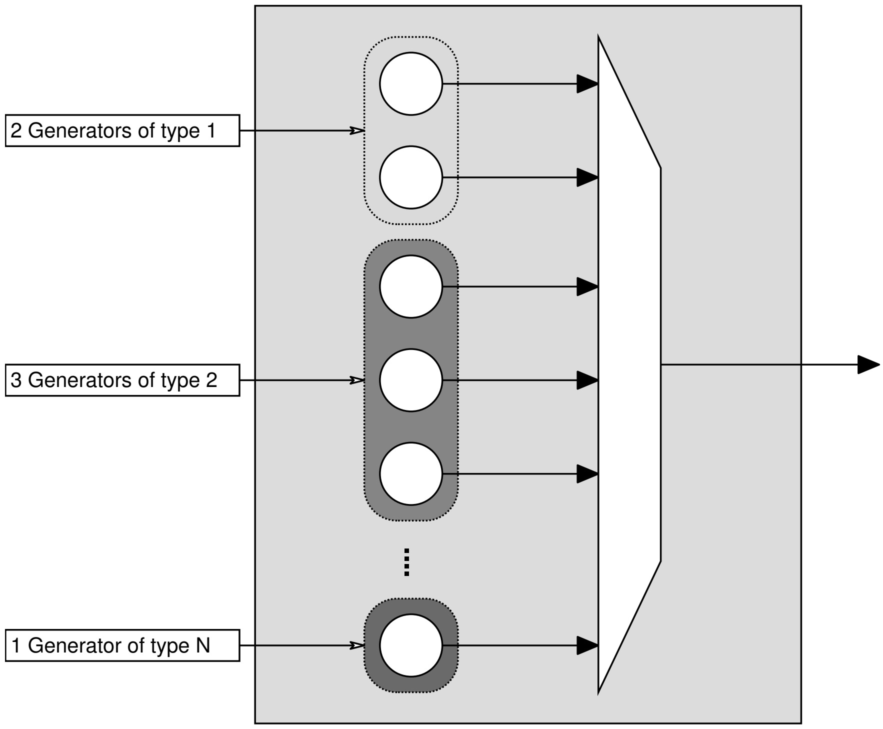


Figure 6.1: Internal Structure of GeneratorSet

GeneratorGroup

The appreciation of `GeneratorGroup` is to simplify the creation procedure of generators within `GeneratorSet`. Each `GeneratorGroup` is responsible for exactly one kind of generator, hence a prototype of the generator is available, which is cloned by calling `create-Generator()`. Further `GeneratorGroup` knows, by the attribute `noOfGens`, how many generators should be produced.

`GeneratorGroup` supports only parsing and the keywords are: *NoOfGens, StdGenerator* and BurstGenerator.

GeneratorSet

As already mentioned, `GeneratorSet` contains severalgGenerators and a multiplexer. The generators are created by the help of `GeneratorGroup` and a parsing procedure. During the parsing procedure several GeneratorGroups with the according sample generator and the number of required copies are generated. It is obviously that just the keyword *GeneratorGroup* is supported. After finishing the parsing procedure, the groups are evaluated by `post-Parse()` for generating single generators. Eventually the GeneratorGroups are deleted and the Generators are connected to the multiplexer.

7 Port Concept

Model components communicate with each other using messages / cells / packets via so-called ports. Therefore, the model components have a defined interface which increases their re-usability.

7.1 Messages

7.1.1 Base Class

The center of the simulation is the exchange of messages (that represent data packets) between individual model components.

All messages are derived from the class `Message` and therefore inherit the following characteristics:

- A so-called message type is defined upon constructor call as a 32-bit word and can be retrieved with `getMessageType()`.
- Messages have a length which is set to one by default. This length can be set by the method `setLength()` and retrieved by `getLength()`.
- Messages can contain several time stamps of the type `TimeStamp` for the purpose of transfer time measurements. They can be managed with the functions `addTimeStamp()`, `getTimeStamp()` and `removeTimeStamps()`. This is especially important when dealing with time measurements (Section 8).
- `Message` contains several output functions (`printMessage()`, `printHeader()`, `printContent()`) that can be overridden if needed.

Creating messages usually takes place in a generator (`Generator`) (Section 6.2.1). Generally a generator contains an element of the class `MessageFactory`, which copies a given default message on to the heap with the help of the copy constructor (can be overriden). By using the port concept, a message is passed from one model component to the next, whereby a reference to each message is additionally passed on during the process of the handshake protocol between two components. Messages are usually deleted in so-called sinks (`Sink`, Section 6.2.2).

7.1.2 Special Message Types

Within an individual model, it might be necessary to have a message with special members. `LabelMessage` and `IPPacket` are two implemented examples. `LabelMessage` has one additional label whereas `IPPacket` includes the most important fields of the IP header like source and destination id, a group and type field as well as a member for a tag usually used for scheduling. Special Methods to access and set these fields start with `get<*>()` and `set<*>()`.

7.2 Ports

7.2.1 Basic Characteristics of Ports

All ports are derived from the same base class `Port`. There are sending ports (base class `OutputPort`) and receiving ports (base class `InputPort`). This is identified by a field within `Port`, which can be accessed with `getPortType()`.

Each port belongs to a certain model component, which must be declared upon initialization as a parameter (`owner`). In addition, each port has a local name (declared upon constructor call), which must be unique within the model component and together with the global name of the superjacent model component composes the global port name.

7.2.2 Connecting Ports

Two model components are connected by creating a directed connection between two corresponding ports which are to be used for their communication. There are several ways of accomplishing this:

- Connection of an output port from model component A with an input port of model component B by calling the `connect()` method from A or B and while declaring the `Entity` entities and their corresponding port names. The following is valid:
  - A may not be owner of B.
  - B may not be owner of A.
- A and B may not be identical (feedback).
- Connection of an output port model component A with an output port of the subjacent model component B by calling `connect()`
- Connection of an input port from model component A with an input port of the superjacent model component B by calling `connect()`.
- As an alternative in both previous cases it is also possible to make the port of the subjacent model component addressable for the superjacent model component by using the `Entity` method `aliasPort()`, so that the port of the superjacent model component becomes superfluous.
- Each port can have a maximum of one incoming and one outgoing connection, so it is not possible to connect a model component via one output port with multiple input ports of other model components.
- If an incoming connection exists, the address of the preceding port (in the private part) is stored as a data element. If an outgoing connection exists the address of the succeeding port is stored. Both occur when connecting with `connect()`.

7.3 Communication between Ports

A handshake protocol has been defined for the exchange of messages between two model components via ports, which has been realized with three functions: .

- `messageIndication()` notifies the receiver, that a message is waiting at the sender.
- `isMessageAvailable()` can be called from the receiver side to inquire if the sender is holding a message.

- `getMessage()` is called from the receiver side to pick up a waiting message from the sender.

Sender offers message ("active sender")

1. Sender entity calls `messageIndication()` of the output port.
2. Output port calls `messageIndication()` of the successor port (= input port of the receiver entity).
3. Input port calls `handleMessageIndication()` of its message handler because there is no successor port.
4. InputMessageHandler calls a handler function of the receiver entity (e.g., `handleMessageIndication()`) or reacts directly.
5. Receiver entity or `InputMessageHandler` decides if the message can be retrieved at the current time and if necessary calls `getMessage()` of the input port.
6. Input port calls `getMessage()` of the predecessor port (= output port of the sender entity).
7. Output port calls `handleGetMessage()` of its message handler, because there is no predecessor port.

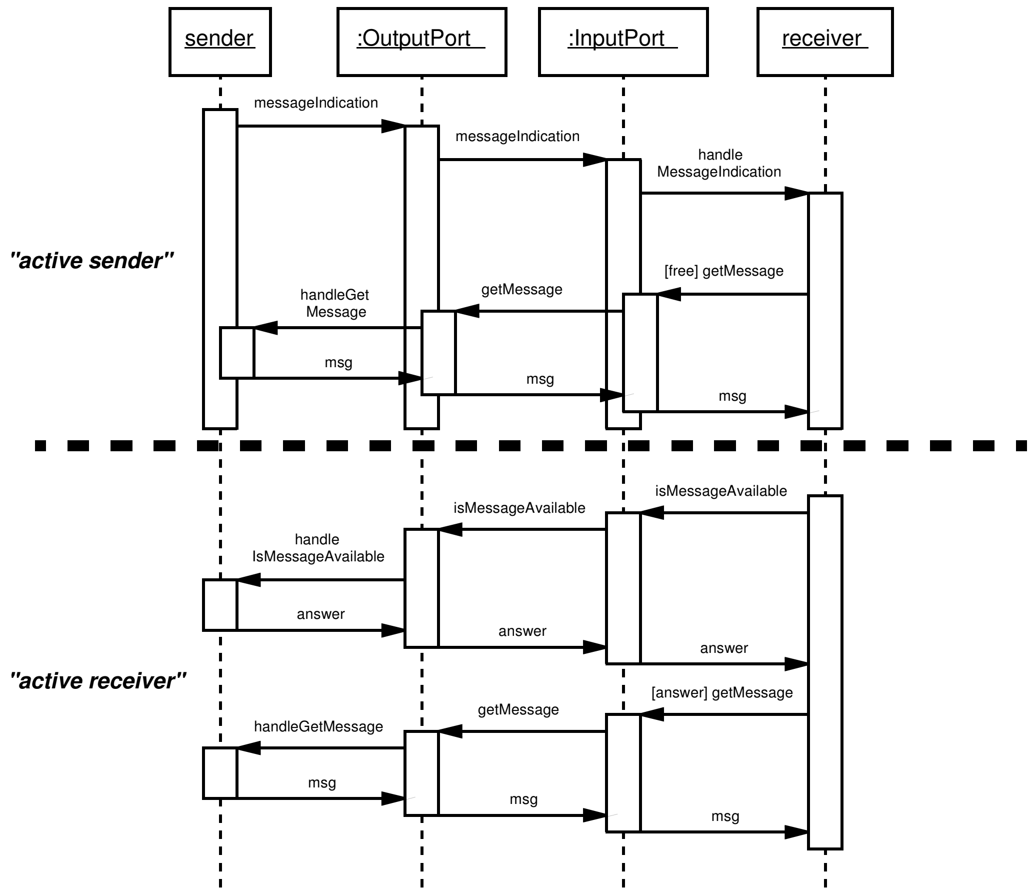


Figure 7.1: Simplified illustration of events during message exchange

8. Output port calls either a handler function of the sender entity (e.g., `handleGetMessage()`) or reacts directly.

A reference to the sent message finally lands as a return value of the original `getMessage()` call of the receiver. To be exact, calls made by the `messageIndication()` and `getMessage()` methods of a port are not directly delivered to the succeeding or preceding port (steps 2 and 6) or the corresponding message handler (steps 3 and 7). In fact the handler methods of any connected message filters (Section 8) are called first and then the TPort method `handleMessageIndication()` or `handleGetMessage()` are called which take care of delivery.

Receiver inquires about the next message ("active receiver")

1. Receiver entity calls `isMessageAvailable()` of the input port.
2. Input port calls `isMessageAvailable()` of the output port of the sender entity.
3. Output port calls `handleIsMessageAvailable()`.
4. Output port calls either a handler function of the sender entity (e.g., `handleIsMessageAvailable()`) or reacts directly. The return value announces if a message is available or not and finally lands at the receiver entity.
5. The receiver entity generally calls `getMessage()` of the input port if `true` is returned.
6. The input port calls `getMessage()` of the output port of the sender entity.
7. The output port calls `handleGetMessage()` of its message handler because it has no preceding port.
8. Output portcalls either a handler function of the sender entity (e.g., `handleGetMessage()`) or reacts directly.

A reference to the sent message finally lands as a return value of the original `getMessage()` call by the receiver. The extension and refining of sequences in reference to message filters (Section 8) and the port's own `handle<*>()` methods as in the "active sender" scenario.

7.4 Special Ports

There are several classes that have been derived from `InputPort` or `OutputPort` in the simulation library, that have ports which represent specially defined functions:

- The class `NonBlockingOutputPort` serves to model output ports which may not be blocked.
  - The class is publicly derived from `OutputPort`.
  - The model component which the port belongs to calls the method `sendMessage()` in order to deliver a message which causes a `messageIndication()` call within the port.
  - The following model component must have retrieved the message at its input port via `getMessage()` before the sending model component can send the next message with `sendMessage()`.

  - Requests by the receiving model component with `isMessageAvailable()` are answered with `true` until the last message delivered to the output port with `sendMessage()` has been picked up.

- The same statements made about the `NonBlockingOutputPort` are valid for the class `SynchronousOutputPort` with the single exception that messages sent by `sendMessage()` and announced to the following model component with `messageIndication()` must be directly picked up i.e., the `handleMessageIndication()` method (or the corresponding method with this functionality) within the message handler or the model component on the receiver side must call `getMessage()`.

- The class `PolledInputPort` is publicly derived from `InputPort`. Its main characteristic is that it ignores message offers by `messageIndication()`, so that a message exchange must always be initiated by the receiver ("active receiver", Section 7.3).

8 Port Monitors

Meters represent tools that can be easily integrated into the model in order to read and evaluate the flow of messages at various points within the model.

8.1 Meters

One way of executing measurements on a model is to insert statistics classes directly into each model component (Section 5). For measurements performed on a given interface between ports of a model component (e.g., transfer time between input and output port of a model component), there are so-called meters that only need to be "plugged in" to one or more ports.

8.1.1 Characteristics of the Base Class

The base class for meters is the class Meter, which has only one particular characteristic. Upon constructor call a system-wide unique ID is allocated to each object. The classes derived from Meter either measure at one point or between two points and thus are called OnePointMeter and TwoPointMeter, respectively.

- The class OnePointMeter provides the method attachInput() which connects a message filter to a port. Parameters include a reference to a model component as well as the (local) port name (Section 7.2.1).
- The class TwoPointMeter provides the method attachFromPort() and attachToPort() which connects a message filter between the ports. Parameters include a reference to a model component as well as the (local) port name (Section 7.2.1).
- The classes OnePointMeter and TwoPointMeter define the method handleGetMessage(). This method gets the message from the message filter and calls the virtual (and therefore overwritable) method evaluateMessage().
- The message filters are created from the function createMessageFilter() which is defined in <*>Port<*>PointMeter and which can be overridden.

8.1.2 One-Point Meters

CDV Meter

Upon construction, CDVMeter and StdCDVMeter require a (fixed) rate which is taken to calculate the cell delay variation (CDV) of the measured values. In addition, DistCDVMeter also requires the array size for the distribution as well as the lower and upper limit of variation from the specified rate that has to be captured in during simulation.

CDVMeter overwrites the method evaluateMessage() which – after calling useMessage() to filter special messages – calculates the CDV which is used to update an internal StdSampleStatistic object.

Interarrival Time Meter

The class IATMeter is the base class of StdIATMeter, and DistIATMeter. IATMeter overwrites the method evaluateMessage() which – after calling useMessage() to filter special messages – calls the method evaluateSample() which is overwritted in its derived classes. `evaluateSample()` updates the appropriate statistic (see also Section 5). Thus, `DistIATMeter` updates a distribution statistic and `StdIATMeter` updates a sample statistic.

According to the respective statistics, different parameters are required upon constructor call.

- `StdIATMeter` does not require special parameters besides name and owner.
- `DistIATMeter` also needs the *array size* of measures as well as the *lower* and *upper limit* of the interarrival time that has to be captured during simulation.

Message Length Meter

The class `MessageLengthMeter` is the base class of `StdMessageLengthMeter` and `MedianMessageLengthMeter`. `MessageLengthMeter` overwrites the method `evaluateMessage()` which - after calling `useMessage()` to filter special messages - calls the method `evaluateMessageLength()`. The `StdMessageLengthMeter` uses a `StdSampleStatistic`, the `MedianMessageLengthMeter` uses a `StdMedianStatistic`.

Count Meter

The class `CountMeter` defines the method `handleGetMessage()`. This method calls the virtual method `countMessage()` which simply increases a counter by one. Upon constructor call, no further parameters are required.

The class `SimulationControlCounter` derived from `CountMeter` which is implemented with the help of the class `SimNotifier` (private derivation) and in the frame of the sequence control is used to indicate the end of a phase/partial phase of the simulation. The `countMessage()` method has been overridden in such a way that one of the methods `endOfTransientPhase()` or `endOfBatch()`, both inherited from `SimNotifier`, are called depending on the partial phase which is currently active in the simulation (warm-up phase or the n-th batch) (Section 3.2), if the counter has reached a certain value. Upon constructor call, the number of messages in the transient phase, the number of messages during a batch as well as the number of batches are requird.

Rate Meter

The class `RateMeter` defines the method `evaluateMessage()`, which update a rate statistic. Before the update, the method `useMessage()` is called to allow the implementation of filters. Upon constructor call, no further parameters are required.

Distribution Rate Meter

The class `DistributionRateMeter` can be used to measure the distribution of short term throughput measurements. For this purpose, the short term average during a given short term period is calculated, which is then used to update an internal `StdDistributionStatistic` at the end of each short term measurement period. The constructor expects the parameters `arraySize` (unsigned integer), `lowerLimit` (double) and `upperLimit` (double), which are similar to the parameters of the `StdDistributionStatistic`. Additionally, it expects the `shortTermPeriod` (Time), a name, and an owner. Finally, the parameter `measurePortsIndividually`

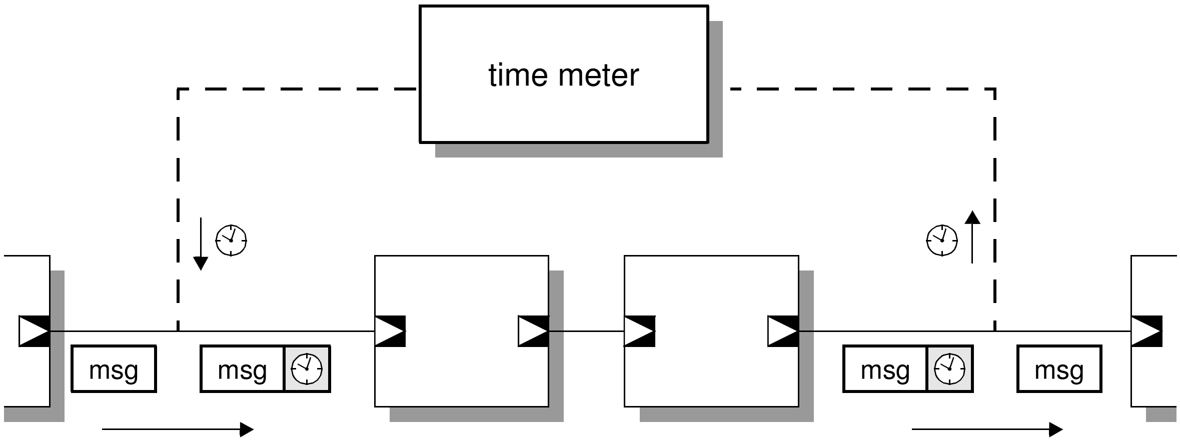


Figure 8.1: Principle of a transfer time meter

(boolean) lets you specify, whether the data rate on each port shall be measured separately, or whether an aggreagate data rate shall be measured. If it is measured separately, the distribution statistic will be updated with all individual short term throughputs at the end of each update period. Otherwise, it will only be updated with the aggregated value.

Boundary Rate Meter

The class `BoundaryRateMeter` can be used to measure the quantile of short term throughput measurements. For this purpose, the short term average during a given short term period is calculated, which is then used to update an internal `JainBoundaryStatistic` at the end of each short term measurement period. The constructor expects the parameters `quantile` (double), the `shortTermPeriod` (Time), a name, and an owner. Additionally, the parameter `measurePortsIndividually` (boolean) lets you specify, whether the data rate on each port shall be measured separately, or whether an aggreagate data rate shall be measured. If it is measured separately, the distribution statistic will be updated with all individual short term throughputs at the end of each update period. Otherwise, it will only be updated with the aggregated value.

8.1.3 Two-Point Meters

Several classes for transfer time measurement are derived from `TwoPointMeter`:

- The class `TimeMeter` is implemented as base class and provides the methods `createTimeStamp()`, `evaluateTimeStamp()` and `deleteTimeStamp()` for an efficient memory management of time stamps (class `TimeStamp`) using a so-called free list. Whereby a time stamp contains the `ID` of the meter as well as a time value. Fig. 8.1 depicts the principle of a transfer time meter.
- The class `StdTimeMeter` derived from `TimeMeter` is a meter to determine the transfer times between two or more ports and provides the following features. Upon constructor call, no further parameters are required.
  - The use of generic message filters and management in a respective data structure

- `attachFromPort()` connects a meter to a port whose time measurement begins by installing a filter of the type `GenMessageFilter`, which uses the `setTimeStamp()` method from `StdTimeMeter` as a handler function.
- `attachToPort()` connects a meter to a port whose time measurement ends by installing a filter of the type `GenMessageFilter` and uses the `readTimeStamp()` method from `StdTimeMeter` as a handler function .
- Calling `setTimeStamp()` causes a time stamp with the current system time to be attached to the message that is just passing the port by calling the `Message` method `addTimeStamp()`.
- `readTimeStamp()` acquires the time stamp that belongs to the meter from the current message using the method `getTimeStamp()` from `Message`, evaluates its transfer time and removes it from the message (`removeTimeStamp()`).
- Evaluation of transfer times with the help of a statistic of the type `SampleStatistic` (Section 5.2)

• The class `DistTimeMeter` is derived from `StdTimeMeter` and additionally provides a distribution statistic. Therefore, the *array size* of measures as well as *lower* and *upper limit* of the interarrival time that has to be captured during simulation are required upon construction.

• The class `BoundaryTimeMeter` is used to find the quantile for a given percentage. It uses the a boundary statistic (see also Section 5.11) and thus provides two constructors, one for `StdBoundaryStatistic` and one for `JainBoundaryStatistic`. Accordingly, additionaly to the percentage (e.g., 0.95) the expected number of samples is required upon construction in case the `StdBoundaryStatictic` is applied.

• The class `MedianTimerMeter` is used to determine the median of transfer time values between two points in a simulation model.

9 Printing Results

The simulation library offers a flexible concept to print out results to an XML file. It utilizes so-called print formats and is supported by a number of special print-out classes. This concept makes a print-out under direct use output streams obsolete, but does not disallow it.

All statistic classes provide default print formats (as described in Section 9.4). Therefore the concepts described in Section 9.1 - Section 9.3 are only important for user which want to modify the default output or create new classes with content to be included in the print-out.

9.1 Registering for Printing

All classes, whose objects are to be addressable for printing results, must implement the interface `SimNode` and registered in the model hierarchy. This is already given for model components and statistics, because `Entity` and `Statistic` own a `SimNode` object.

Each object implementing the interface `SimNode` has a method `getPrintHandler()`, which is called upon printing. In order to print results, a suitable handler implementing the interface `Printable` must be provided, here.

9.2 Sequence of Printing Results

To start printing the results, the function `printResults()` of the global SimNode manager is called with the result type name as a parameter. The call of this function occurs in the methods `printBatchResults()` and `printResults()` of the class `Stdsimulation` (Section 3.1). The printing will be performed by traversing the tree of SimNode objects and calling their corresponding print handlers.

9.3 Filtering Results

The command line allows to specify a filter file (option *-f*). Within such a filter file, the user can specify via several regular expression which output is actually written to the log file. The regular expressions work on the hierarchie of print server names at result level (e.g. *Model:Node1:Meter1:TransferTime:mean*). The applied concept is very closely related to that of iptables.

The first line of the filter file specifies the default behaviour.

- `default include`
  All output not matched by any rule will be written to the log file.
- `default exclude`
  All output not machted by any rule is not written to the log file.

After the specification of the default behaviour several rules follow. After the first fit the filter file is not further evaluated.

A rule consists of a keyword (`include` and `exclude`) indicating what to do with the entires matched by the following regular expression and this regular expression itself. A regular expression may contain wildcards (.*). The meaning of such a wildcard is twofold depending on its position in the regular expression. At the end of the regular expression it matches to any arbitrary further string. At the begining or somewhere in between, the wildcard only matches further strings on this level in the print server hierarchy. Especially it does not match a colon (:) separating the hierarchies. Therefore the regular expression *Model:.*:TransferTime:.** does not match the above example of *Model:Node1:Meter1:TransferTime:mean*, while the regular expression *Model:.*:.*:TransferTime:.** does.

A simple filter file including only the first node of a model looks like the following:

```
default exclude
include Model:Node1:*
```

10 Simulation Parameters

The IKR Simulation Library can read simulation parameters from a file, the so-called parameter file. How to write a parameter file and how to read parameters inside the simulation program will be explained in the following.

10.1 Parameter File Structure

The parameter file will be read on a per-line basis. Each line must have the following structure:

```
<key> = <value>
```

There are two principle for defining keys:

- The key may be any name, which the simulation program can query for. Organizing the name space must be done by the programmer.
- The key or the first part of the key is the fully qualified name of the SimNode that shall read the corresponding parameter(s). Names may contain wildcards to make the key match different queries.

The value will be interpreted as follows:

- Whitespaces will be interpreted as separators of vectors. Scalar values containing a whitespace character must be put in quotes.
- Vectors must begin with a [character and end with a] character. Inside a vector, the values must be separated with whitespaces.
- A Vector can again contain vectors.

You can use the # character to write comments. All characters of a line following the # character will be ignored. Empty lines will be ignored.

Example parameter file:

```
# Calls = 10000
TandemModel.*.NrOrGenerator = 42 # the answer
TandemModel.Node*.Generator = StdGenerator
TandemModel.Node*.Generator*.IATDist = NegExp
TandemModel.Node*.Generator*.IATDist.Mean = 50
```

10.2 Reading Parameter Values

10.2.1 Class Parameters

In every SimLib simulation program, there is one object of class Parameters. This object represents the content of the parameter file. The object provides a set of methods to query the parameters listed in this file. The methods, that can be used for querying, are the following ones:

- `hasParameter(<query>)` returns true if the parameter exists.
- `hasSubTree()` returns true if the subtree exists.

- `get(<query>)` returns the value of the parameter.
- `getOrUseDefault(<query>, <default value>)` returns the value of the parameter if it is listed in the parameter file. If not, the default value will be used. Note that the default value will be passed as a string and will only be parsed if it is needed.

There are different variants for providing the query to the methods listed above:

- `(String name)`: The name must specify the fully qualified name of the queried parameter.
- `(Prefix prefix, String name)`: The name will be extended by the prefix.
- `(SimNode simNode, String name)`: The name will be extended by the name of the simNode.

10.2.2 Class Value

The class `Value` is the abstract base class of the classes `Scalar` and `Vector`. These objects provide methods to return a parameter value in one of the following forms:

- `boolean`
- `enum`
- `String`, `String[]`, or `String[][]`
- `double`, `double[]`, or `double[][]`
- `int`, `int[]`, or `int[][]`
- `long`, `long[]`, or `long[][]`

10.3 Parser Classes

In order to keep the model compoments classes simple, they don't contain the parsing functionality anymore. Instead, there is are separate parser classes, all having the suffix „Parser". All these classes implement an interface derived from the interface `AbstractParser`.

10.3.1 Interface Parser

The most simple sort of parser classes are those implementing the interface `Parser`, which defines the following method:

- `public T parse(Parameter par)`

The class `Parameter` represents a pointer to a location in the parameter tree, which shall be used as the starting point for parsing.

10.3.2 Interface ParserWithRNG

In addition to the parameters from the parameter file, model components may require an object for creating random numbers. Since the references of distribution classes to the randon number generator are final, such instances have to be provided during instantiation. For this purpose, the interface `ParserWithRNG` defines the following method:

- `public T parse(Parameter par, RandomNumberGenerator rng)`

Especially distributions parser classes implement this interface.

10.3.3 Interface ParserWithSimNode

Model compoment may want to insert themselves into the model hierarchy. Therefore, they need a reference to the parent SimNode. For this purpose, the interface `ParserWithSimNode` defines the following method:

- `public T parse(Parameter par, String name, SimNode parentNode)`

Parser classes of model compoments like queue and phases implement this interface.

10.4 Parse Manager Classes

Besides the parser classes that can create objects of one specific type, there are parse manager classes that can select one parser class based on a parameter and create and execute this parser class.

Classes implementing `ParseManager`:

- `MessageFactoryParseManager` parsing and creating message factories

Classes implementing `ParseManagerWithRNG`:

- `ContinuousDistParseManager` parsing and creating continous distributions
- `DiscreteDistParseManager` parsing and creating discrete distributions

Classes implementing `ParseManagerWithSimNode`:

- `GeneratorParseManager` parsing and creating generators
- `PhaseParseManager` parsing and creating phases
- `QueueParseManager` parsing and creating queue

All parse managers can be extended by custom parser objects.

11 Miscellaneous

11.1 Version Identifier

It is possible to get the version identifier of the IKR Simulation Library 3.2. The version identifier will be in most cases the version number.

The version of your source code base is included in the file `Version.java` in the constant string VERSION_STRING.

References

[1] C. BLONDIA, T. THEIMER: *A Discrete-Time Model for ATM Traffic*, RACE 1022, Document PRLB_123_0018_CD_CC/UST_123_0022_CD_CC, 1989.

[2] S. BODAMER: “Object-oriented simulation.” Contribution to lecture *Teletraffic Theory and Engineering*, Institute of Communication Networks and Computer Engineering, University of Stuttgart, 2001.

[3] S. BODAMER, K. DOLZER, C. GAUGER, M. KUTTER, T. STEINERT, M. Barisch: *IKR Utility Library 2.5 User Guide*. IKR, University of Stuttgart, Jun. 2004.

[4] S. BODAMER, K. DOLZER, C. GAUGER, M. KUTTER, T. STEINERT, M. BARISCH: *IKR Component Library 2.5 User Guide*. IKR, University of Stuttgart, Jun. 2004.

[5] S. BODAMER, K. DOLZER, C. GAUGER, M. BARISCH: *IKR Simulation Library 2.5 User Guide*. IKR, University of Stuttgart, Jun. 2004.

[6] J. ENSSLE, *Modellierung und Leistungsuntersuchung eines verteilten Video-On-Demand-Systems für MPEG-codierte Videodatenströme mit variabler Bitrate*, Dissertation, Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1998.

[7] C. GÖRG: *Verkehrstheoretische Modelle und stochastische Simulationstechniken zur Leistungsanalyse von Kommunikationsnetzen*, Habilitation, RWTH Aachen, 1997.

[8] H. HEFFES, D. M. LUCANTONI: “A Markov modulated charcterization of packetized voice and data traffic and related statistical multiplexer performance.” *IEEE Journal on Selected Areas in Communications*, Vol. SAC-4, No. 6, 1986, pp. 856-868.

[9] R. JAIN, I. CHLAMTAC: “The P**2 algorithm for dynamic calculation of quantiles and histograms without storing observations.” *Communications of the ACM*, Vol. 28, Oct. 1985, pp. 1076-1085.

[10] H. KOCHER: *Entwurf und Implementierung einer Simulationsbibliothek unter Anwendung objektorientierter Methoden*, Dissertation, Institute of Communication Networks and Computer Engineering, University of Stuttgart, 1993.

[11] H. KOCHER, M. LANG: „An Object-Oriented Library for Simulation of Complex Hierarchical Systems“, *Proceedings of the Object-Oriented Simulation Conference (OOS ’94)*, Tempe, AZ, 1994, pp. 145-152.

[12] P. J. KÜHN, T. RAITH, P. TRAN-GIA: “Methodik der stationären Systemsimulation.” Contribution to lecture *Teletraffic Theory and Engineering*, Institute of Communication Networks and Computer Engineering, University of Stuttgart.

[13] P. J. KÜHN: “Reminder on queueing theory for ATM networks.” *Telecommunication Systems*, No. 5, 1996, pp. 1-24.

[14] M. LANG, M. STÜMPFLE, H. KOCHER: “Building a Hierarchical CAN Simulator Using an Object-Oriented Environment.” *Proceedings of the 8th GI/ITG Conference on Measurement , Modelling and Evaluation of Computing and Communication Systems (MMB ’95)*, Heidelberg, Sep. 1995, pp. 327-339.

[15] A. M. LAW, W. D. KELTON: *Simulation Modeling & Analysis*, 2nd edition, McGraw-Hill, 1991.

[16] D. M. LUCANTONI, K. S. MEIER-HELLSTERN, M. F. NEUTS: “A Single-Server Queue with Server Vacations and a Class of Non-Renewal Arrival Processes.” *Advances in Applied Probability*, Vol. 22, No. 1, 1989, pp. 676-705.

[17] B. MAGLARIS ET AL.: “Performance Models of Statistical Multiplexing in Packet Video Communications.” *IEEE Transactions on Communications*, Vol. 36, No. 7, 1988.

[18] S. K. PARK, K. W. MILLER: “Random number generators: good ones are hard to find.” *Communications of the ACM*, pp. 1192-1201, Oct. 1988.

[19] G. D. STAMOULIS, M. E. ANAGNOSTOU, A. D. GEORGANTAS: “Traffic source models for ATM networks: a survey.” *Computer Communications*, Vol. 17, No. 6, Juni, 1994.

[20] T. THEIMER: *How to compute the moments of a GMDP*, RACE 1022, Document UST_123_0023_CD_CC, 1989.

[21] *Apache Ant.* http://ant.apache.org/

Appendix A: Files of the IKR SimLib JAVA Edition

The IKR SimLib consists of the following Java archives (jars):

- ikr-simlib-<Version>.jar: This jar file contains the Java class files.
- ikr-simlib-<Version>-src.jar: This jar file contains the Java source code.
- ikr-simlib-<Version>-api.jar: This jar file contains the HTML class description (created by Javadoc)
- ikr-simlib-example-<Version>.jar: This jar file contains the Java class files of the examples.
- ikr-simlib-example-<Version>-src.jar: This jar file contains the Java source code of the examples.
- ikr-simlib-example-<Version>-api.jar: This jar file contains the HTML class description (created by Javadoc) of the examples.

Each jar file contains also the license text (file *COPYING*) for the GNU Lesser General Public License (LGPL) under which this library is published.