Phase-type Distributions for Realistic Modelling in Discrete-Event Simulation

Philipp Reinecke
philipp.reinecke@fu-berlin.de
Freie Universität Berlin
Institute for Computer Science
Takustraße 9
14195 Berlin, Germany

Gábor Horváth
ghorvath@hit.bme.hu
Budapest University of Technology and Economics
Dept. of Telecommunications
Magyar tudósok krt. 2
1117 Budapest

ABSTRACT

Phase-type (PH) distributions are a valuable tool for representing real-world phenomena such as failure-times or response-times in an analytically tractable way. Recently, the application of phase-type distributions in simulation has received increasing attention. In simulation, phase-type distributions enable good representation of empirical distributions, even if the data does not follow one of the well-known statistical distributions. Furthermore, since phase-type distributions have Markovian representations, analytical approaches can be used to support simulation results. So far, however, well-known discrete-event simulators do not support PH distributions. In this paper we introduce the libphprng library for generating random-variates from PH distributions. Libphprng combines efficient methods with easy usage. Furthermore, libphrng integrates seamlessly with discrete-event simulators such as OMNeT++ without any changes to the library core.

General Terms

Algorithms, Experimentation, Measurement, Performance, Reliability

Categories and Subject Descriptors

G.3 [**Probability and Statistics**]: Distribution functions, Random-Number-Generation; C.4 [**Performance of Systems**]: Modelling Techniques; I.6.m [**Simulation and Modelling**]: Miscellaneous

Keywords

Phase-type distributions, Random-variate generation, Discrete-event simulation

OMNeT++ 2012, March 19-23, Desenzano del Garda, Italy

DOI 10.4108/icst.simutools.2012.247727

1. INTRODUCTION

Many aspects of system resilience can, and ideally should, be studied at different abstraction levels: Experimentation on test-beds is necessary to gain an understanding of the system under realistic conditions. Discrete-event simulation enables fast evaluation of many different configurations. Abstract mathematical models provide general insights and elegant solution methods. Real-world phenomena are often represented in evaluation by stochastic models fitted to measurement traces. These models replicate the phenomenon in a test-bed or simulation, or include its characteristics in analytical solutions. Probability distributions are a common type of stochastic model used in system evaluation.

Phase-type (PH) distributions are a class of distributions that are known to be very powerful for modelling typical phenomena such as response-times, failure times, or interarrival times. PH distributions are defined as the distribution of the time to absorption in a Markov chain with one absorbing state. This characteristic enables elegant analytical approaches, e.g. [7]. While PH distributions have long been used in analytical approaches, they are seldom applied in simulation or measurement studies. The main reason for this gap is certainly the separation between scientific communities. To practitioners, many concepts related to PH distributions might appear rather abstract, if not esoteric, while the focus in the modelling community is typically not on discrete-event simulation or on providing practical implementations.

In this paper we propose the libphprng library for generating PH-distributed random variates. The library is part of the Butools package [1] and can be linked to existing discrete-event simulators to enable the use of PH distributed random variates in simulations. The only adjustment necessary is the implementation of a small wrapper module. We provide wrapper modules for OMNeT++ and NS-2.

The remainder of this paper is structured as follows. First, we provide a very brief and gentle introduction to phase-type distributions, paying particular attention to their advantages over other distributions. We then describe the architecture and features of the library and compare it to other approaches. Finally, in Section 5 we illustrate application of PH distributions in a case-study using the OMNeT++ discrete-event simulator [17], before concluding the paper in Section 7.

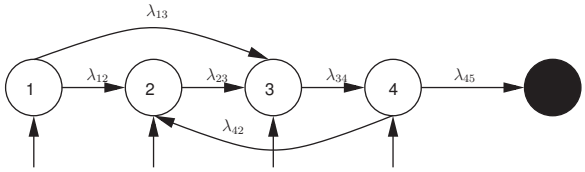

Figure 1: A continuous-time Markov chain with one absorbing state.

2. A GENTLE INTRODUCTION TO PHASE-TYPE DISTRIBUTIONS

Continuous phase-type distributions (PH distributions) are defined as the distribution of time to absorption in a Continuous-Time Markov Chain (CTMC) with one absorbing state [7].[1] The simplest case of a PH distribution is the exponential distribution with rate $\lambda > 0$. If we consider n exponentially distributed random variables $X_1, \ldots, X_n$ where the rates of the exponential distributions are $\lambda_1, \ldots, \lambda_n > 0$, different distributions can be constructed by combining the individual distributions $F_1(t), \ldots, F_n(t)$. For instance, a mixture of these distributions with probabilities $\alpha_1, \ldots, \alpha_n$ whose sum is 1 forms the hyper-exponential distribution with density

$$f_{HEx}(t) = \sum_{i=1}^{n} \alpha_i f_i(t). \tag{1}$$

This distribution arises if we assume that with probability α_i a sample is drawn from the ith random variable, X_i. As an other example, we may consider the sum of exponentially distributed random variables, which has a hypo-exponential distribution. If all the rates are equal to λ, we get an Erlang distribution, with density

$$f_{Erl}(t) = \frac{\lambda t^{n-1} \mathrm{e}^{-\lambda t}}{(n-1)!}. \tag{2}$$

Phase-type distributions generalise the idea of combining exponential distributions. Consider the continuous-time Markov chain shown in Figure 1 and assume we enter this chain at some state i. Now, as time progresses, state-changes will occur and different states will be visited, before the chain finally enters the absorbing state. For each state that is visited, the time before going to the next state follows an exponential distribution. Thus, the time that it takes to reach the absorbing state from an initial state i is a sum of samples from exponential distributions. By entering at different states i or by taking different paths through the CTMC, different mixtures of distributions occur.

To put this into more formal terms, consider a tuple $(\boldsymbol{\alpha}, \mathbf{Q})$ of a vector $\boldsymbol{\alpha} \in \mathbb{R}^n$ and a matrix $\mathbf{Q} \in \mathbb{R}^{n \times n}$. Let $\mathbb{1}$ be the column vector of ones of appropriate length, and let $\boldsymbol{\alpha}\mathbb{1} = 1$ and $\boldsymbol{\alpha} \geq \mathbf{0}$. That is, $\boldsymbol{\alpha}$ is a non-negative vector whose elements sum to one. Furthermore, let

$$\mathbf{Q} = \begin{pmatrix} -\lambda_{11} & \cdots & \lambda_{1n} \\ \vdots & \ddots & \vdots \\ \lambda_{n1} & \cdots & -\lambda_{nn} \end{pmatrix} \tag{3}$$

be such that $\lambda_{ij} \geq 0$ for $i, j = 1, \ldots, n; i \neq j$ and $\lambda_{ii} > 0$ and $\mathbf{Q}\mathbb{1} \leq \mathbf{0}$ (i.e., all non-diagonal elements of $\mathbf{Q}$ are non-negative, all diagonal elements are negative, and all row-sums of $\mathbf{Q}$ are either zero or negative). If $\mathbf{Q}\mathbb{1} \neq \mathbf{0}$, then $\mathbf{Q}$ gives the transition rates for the transient states of the continuous-time Markov chain with generator matrix

$$\hat{\mathbf{Q}} = \begin{pmatrix} \mathbf{Q} & \mathbf{Q}\mathbb{1} \\ \mathbf{0} & 0 \end{pmatrix}, \tag{4}$$

while $\boldsymbol{\alpha}$ gives the initial probabilities of the transient states. The absorbing state of this CTMC is $n+1$, and the tuple $(\boldsymbol{\alpha}, \mathbf{Q})$ describes the behaviour of the CTMC in the transient part. Given this notation, we can define PH distributions:

Definition 2.1. *Let $(\boldsymbol{\alpha}, \mathbf{Q})$ be as described before. The* probability density function (PDF)*, cumulative distribution function (CDF), and kth moment, respectively, are defined as follows [7, 2, 14]:*

$$f(x) = \boldsymbol{\alpha}\mathrm{e}^{\mathbf{Q}x}(-\mathbf{Q}\mathbb{1}), \tag{5}$$

$$F(x) = 1 - \boldsymbol{\alpha}\mathrm{e}^{\mathbf{Q}x}\mathbb{1}, \tag{6}$$

$$E\left[X^k\right] = k!\boldsymbol{\alpha}(-\mathbf{Q})^{-k}\mathbb{1}. \tag{7}$$

where $\mathbb{1}$ is the column vector of ones of the appropriate size. The size *of the $(\boldsymbol{\alpha}, \mathbf{Q})$ representation is the size of the vector $\boldsymbol{\alpha}$, which is equal to the size of the square matrix $\mathbf{Q}$.*

Note that $\boldsymbol{\alpha}$ is a row vector and both $\mathbb{1}$ and $-\mathbf{Q}\mathbb{1}$ are column vectors. Consequently, the above definitions indeed yield scalar values.

To illustrate these concepts, consider again the exponential distribution with rate λ. In this case, $n = 1$, $\boldsymbol{\alpha} = (1)$ and

$$\mathbf{Q} = \begin{pmatrix} -\lambda \end{pmatrix}. \tag{8}$$

For the hyper-exponential distribution of size n we have

$$\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \text{ and} \tag{9}$$

$$\mathbf{Q} = \begin{pmatrix} -\lambda_1 & 0 & & & \\ 0 & -\lambda_2 & 0 & & \\ & \ddots & \ddots & \ddots & \\ & & & -\lambda_{n-1} & 0 \\ & & & 0 & -\lambda_n \end{pmatrix}, \tag{10}$$

while the hypo-exponential distribution is defined by

$$\boldsymbol{\alpha} = (1, 0, \ldots, 0) \text{ and} \tag{11}$$

$$\mathbf{Q} = \begin{pmatrix} -\lambda_1 & \lambda_1 & & & \\ 0 & -\lambda_2 & \lambda_2 & & \\ & \ddots & \ddots & \ddots & \\ & & & -\lambda_{n-1} & \lambda_{n-1} \\ & & & 0 & -\lambda_n \end{pmatrix}. \tag{12}$$

Note that the $(\boldsymbol{\alpha}, \mathbf{Q})$ representation for PH distributions is by no means unique. That is, there usually exist several different tuples that define the same distribution [7]. Furthermore, there exist several sub-classes of PH distributions. While these sub-classes are less flexible than the general PH class, there are special structures available for them that allow more efficient algorithms for random-variate generation. Of particular interest is the class of acyclic phase-type

[1] Note that the same concept exists for DTMCs, however, in the present work we focus on the continuous-time case.

(APH) distributions, which contains all PH distributions that have a representation without cycles in the Markov chain. For this class, efficient algorithms and algorithms for optimisation of the representation have been proposed in [10].

3. PH DISTRIBUTIONS IN SYSTEM EVALUATION

As discussed in the introduction, system evaluation should be done at different abstraction levels. In order to evaluate a system, one typically performs experimentation on test-beds, simulation, or analysis. In all three approaches, a model of the real system is used. This system model must contain important phenomena characterising the real system, such as service-times or failure-times. In order to get representative results, these phenomena, as present in the model, should be based on observations of the real system. Given such observations, the simplest approach is to replay measurement traces directly. However, measurement traces do not generalise. That is, results obtained for one trace can only be trusted to hold for that trace, and do not give general results about the system. Furthermore, traces can only be used in simulation and experimentation, but are not applicable to analytical approaches.

These issues are typically addressed by using a stochastic model instead of the trace itself to replicate the phenomenon in the evaluation. Probability distributions are the most common way of modelling real-world phenomena for use in system evaluation (cf. e.g. [16]). In simulation and test-beds, they are used to generate random variates that replicate the phenomenon, e.g. service-time, while in analytical approaches the properties of the distribution are used.

Given a measurement trace, random variates may be generated by directly drawing from the empirical distribution of the data. This is a straightforward approach that can be implemented using e.g. OMNeT++'s `cPSquare` module. On the other hand, this method of drawing from an empirical distribution based on raw data cannot be used in analytical methods; consequently, one cannot employ the same stochastic model in the simulation and in the analysis. A more general approach is then to parameterise one of the various theoretical distributions that are available with OMNeT++ such that it represents important properties of the data. For example, the mean may be represented by choosing an exponential distribution with appropriate parameterisation, and both mean and variance may be reflected by e.g. using a lognormal distribution. Although more general, this approach still has its shortcomings. First, many data sets cannot be represented well by these theoretical distributions. For instance, measurement data from TCP connections over lossy links often exhibits distinct peaks at the values of the TCP RTO timeout, i.e. at 3 s, 6 s, 9 s, and so on [11]. While we may parameterise one of the standard statistical distributions such that it has the same mean and variance as this data, the characteristic shape of the density cannot be reproduced. Second, theoretical distributions differ in their parameterisation and in the methods of drawing random variates. Therefore, there is no generic way of applying these distributions. Third, although possible, analysis using general distributions is often quite tedious.

Phase-type distributions address all three of these problems: As we will illustrate in Section 5, they can be parameterised such that they reflect the empirical moments, density, or distribution function closely. Second, all PH distribution share the representation as a tuple $(\boldsymbol{\alpha}, \mathbf{Q})$ of a stochastic vector $\boldsymbol{\alpha}$ and a sub-generator matrix $\mathbf{Q}$. There exist several generic tools that automate the task of fitting PH distributions to given measurement data, e.g. PhFit [2] or G-FIT [15]. The tuple representation also enables the development of generic algorithms for random-variate generation from PH distributions [9]. These algorithms can generate random variates from a PH distribution irrespective of the distribution it approximates. Third, as phase-type distributions are still Markovian, they can be used efficiently in analytical approaches (cf. e.g. [7]).

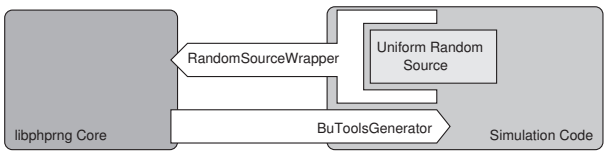

Figure 2: Architecture of libphprng.

4. THE LIBPHPRNG LIBRARY

We have developed the libphprng library to enable the efficient use of phase-type distributions in system evaluation using discrete-event simulation. Libphprng employs the methods for efficient random variate generation described in [13, 9], [3], and the optimisation method described in [10]. Libphprng is part of the Butools package [1], which provides easy-to-use implementations of various methods for Markovian and non-Markovian distributions and processes. Libphprng is a shared library that implements a C++ class for random-variate generation from phase-type distributions and provides a generic interface for connecting the library to arbitrary simulation tools. Figure 2 shows the basic architecture of libphprng: The library core provides the functionality for generating PH-distributed random variates. This code is accessible through the `BuToolsGenerator` class. The user creates an object of this class for each required PH-distributed random-variate source, specifying a file that contains the parameters of the distribution. Upon creation, a `BuToolsGenerator` object reads the PH distribution from the file and parameterises itself such that it generates random variates from this distribution. Then, each call to the method `BuToolsGenerator::getVariate()` yields one random variate.

The code for generating random variates requires uniform random numbers. OMNeT++ and other discrete-event simulators provide elaborate uniform random number generators. In particular, OMNeT++ offers several user-configurable independent random-number streams. These facilities are supported by libphprng through the use of a thin, simulator-specific wrapper module for the simulator's native random number generator. The user needs to instantiate an object of a suitable `UniformRandomWrapper` class and register this object with the `BuToolsGenerator` object. The `BuToolsGenerator` object will then use the specified uniform random number generator, supporting simulator-specific features such as independent streams. We provide wrapper code for both OMNeT++ and ns-2; other wrappers can easily be built following these examples.

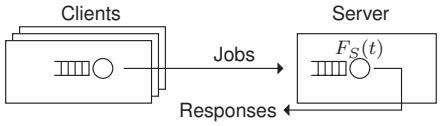

Figure 3: The Scenario considered in the Case-Study.

4.1 Related Work

We are aware of only one other tool that supports phase-type distributions in discrete-event simulators: The `ArrivalProcess` module [4] provides stochastic processes for OMNeT++. In particular, `ArrivalProcess` supports Markovian Arrival Processes (MAPs), of which phase-type distributions are a sub-class. With proper input, `ArrivalProcess` can be used to generate phase-type distributed random variates: Let $\mathbf{D}_0, \mathbf{D}_1$ be the MAP matrices. By setting $\mathbf{D}_0 = \mathbf{Q}$ and $\mathbf{D}_1 = -\mathbf{Q}\mathbb{1} \cdot \boldsymbol{\alpha}$, we can use `ArrivalProcess` to generate random variates from a phase-type distribution with parameters $(\boldsymbol{\alpha}, \mathbf{Q})$.

`ArrivalProcess` is capable of generating random variates from more elaborate stochastic processes than libphprng. For instance, `ArrivalProcess` can be used to generate correlated random variates, which is not possible with libphprng. On the other hand, the limitation of libphprng to phase-type distributions enables much more efficient algorithms, as structural properties, sub-classes, and optimisation can be used. Such methods are not known for more general stochastic processes. We will illustrate the performance advantage of libphprng in Section 6. Furthermore, from an application point of view, libphprng is more general, in that it is completely independent of the simulator. Since it is implemented as a library, libphprng can be linked dynamically and seamlessly to any simulator (or other application requiring PH random variates), thus enabling efficient random-variate generation without changes to the library and with minimal changes to simulation code.

5. AN ILLUSTRATIVE EXAMPLE

We will now discuss the application of the libphprng library in system evaluation using the OMNeT++ discrete-event simulator. The case-study we present is similar to that in [12], however, since our goal here is to illustrate the use of libphprng, we simplify the study considerably and focus on methodological and implementation details.

Consider the problem of developing an algorithm for computing the request timeout in a service-oriented system. When this timeout elapses, the client assumes that the request failed or will take very long to complete. In this case, the client aborts and restarts its request. If failures and long completion-times are caused by transient faults, the restarted request may yield a response that arrives sooner than if the client had waited for the response to the original request. In service-oriented systems, such transient faults can be due to common influences such as IP packet loss and temporary server overload. In such situations, response-times may often be reduced by restart. Restarting too soon, or too often, however, increases load and may thus exacerbate the problem, increasing response-times even further. Therefore, the optimal restart interval must be computed based on operating conditions.

```
class WorkServer: public cSimpleModule {
        /* ... */
        OMNetRandomSourceWrapper * orsw;
        PhGen * phgen;

        private void initServiceTime
                        (char* filename);
        private double getServiceTime();
}

void WorkServer::initServiceTime
                (char* filename) {
        phgen = new BuToolsGenerator(filename)
        orsw = new
                OMNetRandomSourceWrapper(2);
        phgen->setUniformRandomSource(orsw);
}

double WorkServer::getServiceTime() {
        return phgen->getVariate();
}

/* ... */
```

Figure 4: Source code for the inclusion of libphprng in a simple OMNeT++ server class.

While there exist analytical approaches for computing the optimal timeout in simple scenarios [16], more complex service strategies or job semantics as well as scenarios with multiple clients competing for the same resources require simulation.

In our example we consider the following scenario (Figure 3): Clients generate jobs according to a Poisson process. Each client has an internal FIFO queue from which it removes jobs one-by-one and submits them to the server for processing. If the job is not finished before the restart timeout, the client aborts the job and immediately re-submits it to the server. The next job is taken from the queue only after the current job has been completed. In the server, incoming jobs are stored in one single FIFO queue and processed on a first-come, first-served basis. After the server completes a job, it sends back a response to the client. The time for job processing is governed by the service-time distribution F_S. We are interested in whether restart may help in this scenario as well as in the optimal restart timeout. For simplicity, in the following we assume that there is only a single client. In this case, no queueing occurs in the server.

We implemented this scenario as a simple simulation in OMNeT++. In order to use the libphprng library within a simulation, two steps are required:

1. The simulation source code needs to be modified such that the `BuToolsGenerator` object from the libphprng library is used.
2. The simulation must be linked with the `libphprng.so.1` and `libOMNetRandomSourceWrapper.so.1` libraries.

Figure 4 illustrates the modifications required for using the libphprng library in our example. The figure shows the relevant parts of the `WorkServer` class. In our simulation,

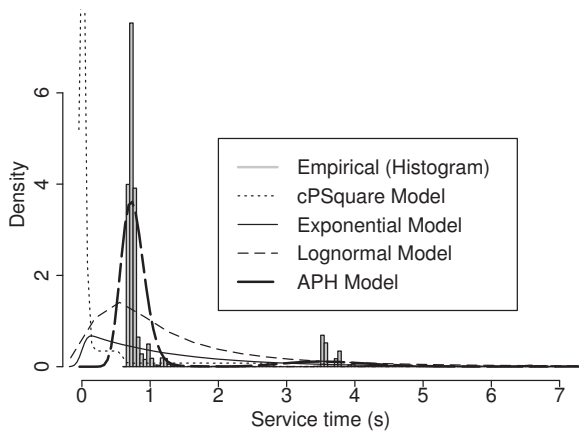

Figure 5: Histogram of empirical service-times and approximated service-time densities (truncated at 7 s).

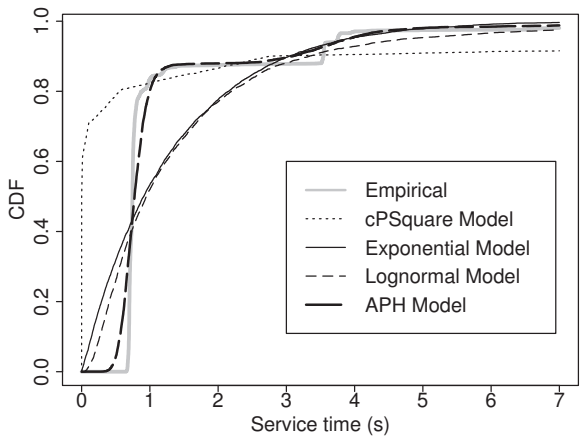

Figure 6: Empirical distribution functions of the empirical and approximated service-time distributions (truncated at 7 s).

the `WorkServer` class implements the simple server. The method `initServiceTime()` is called during initialisation of a `WorkServer` instance to initialise the service time distribution. During operation, every time a job from the queue enters service or is restarted, a service-time sample is required to parameterise the delay. For each service-time sample, `WorkServer` calls its `getServiceTime()` method.

The first modification affects `initServiceTime()`: The server class uses a single `BuToolsGenerator` object to store the service-time distribution. This object is initialised in the method `initServiceTime()`, as follows: First, the new object is instantiated. Upon instantiation, it reads the distribution from the given file. Second, a wrapper object for the uniform random number stream with index 2 is created, and, third, the wrapper object is registered with the `BuToolsGenerator` object.

During operation, calls to `getServiceTime()` must return service times from the service-time distribution. With each call, `getServiceTime()` draws a PH-distributed random variate from the PH distribution stored in `phgen`, by calling the `getVariate()` method.

5.1 Evaluation

We will now illustrate the advantage of using phase-type distributions in our case-study. We consider one client and vary the restart interval from 0.025 s to 60 s. For each restart interval, we observe response-times and compute the mean. We are interested in whether there is an optimal restart timeout, and in its value.

We use a job interarrival time of 10 s and base the service-time distribution F_S on response-time measurements that we have obtained in a test-bed with injected packet loss. We compare the results obtained using four different models for the service-times: First, we use the empirical distribution of the samples to generate random variates. In the following, this model is called *cPSquare Model*. Second, for the *Exponential Model* we use an exponential distribution, parameterised such that the mean is equal to the mean of the data set. Third, we use a *Lognormal* distribution that we fitted to the data set using the R statistical tool [8] such that it reflects the mean and the variance well. Finally, the *APH Model* is an acyclic phase-type distribution of size 50 fitted using the PhFit tool [2].

The cPSquare model is implemented using the `random()` method of OMNeT++'s `cPSquare` class. In the initialisation routine of the server module we create an object of this class and import our data set into the object. The Exponential and Lognormal models use the respective functions of OMNeT++'s library of continuous distribution. The APH model employs the libphprng library, as described above.

First, we consider how well the models fit the measurement data. Using our simulation, we computed one million samples from each model. Figure 5 shows the densities of the empirical data and of the models. Note that the empirical density has two clusters of data, one close to 1 s, and the other close to 4 s. This peculiar shape is typical for HTTP measurements with IP packet loss and can be explained from the interaction between HTTP and TCP, as follows [5, 11]: The TCP retransmits packets after the RTO timeout elapses. While in normal operation the TCP adjusts the timeout based on observed round-trip times, during the connection setup phase the RTO starts at 3 s and is doubled every time it elapses. Therefore, a retransmission of the lost packet takes place only after 3 s, and consequently the corresponding HTTP connection is delayed by at least 3 s. The first cluster thus corresponds to requests whose TCP connections were not affected by packet loss, while the requests in the second cluster have been delayed by one packet loss. For an accurate simulation study, we aim to reproduce these features of the empirical distribution.

Observe that only the APH model shows both spikes of the density, while the other models do not fit the empirical density well. In particular, the cPSquare model underestimates service-times, while the Exponential and Lognormal models do not represent the clusters at all. However, the Lognormal model captures the variance of the distribution. The differences become more evident in Figure 6, where we show the cumulative distribution functions (CDFs) of the data sets: The only model that reflects the steps in the empirical distribution is the acyclic phase-type distribution.

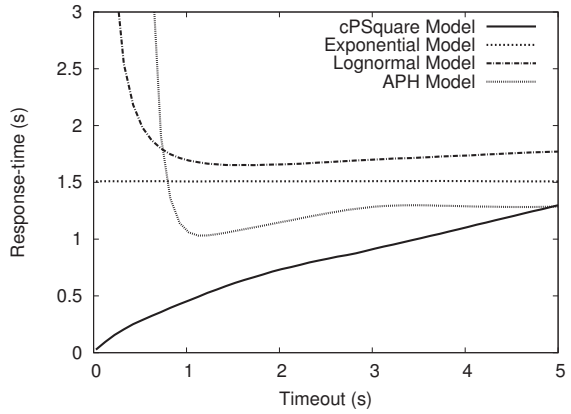

Figure 7: Mean response-times for different timeouts, using different service-time distributions.

Figure 7 shows mean response-times observed for different timeout intervals using these service-time models. Note that there are large differences between the response-time curves: For the cPSquare model, mean response-times grow monotonously with increasing timeout. This implies that the optimal timeout is at zero. Immediate restart, however, is certainly not feasible, and thus results with the cPSquare model imply that restart should not be applied. A similar conclusion can be drawn from the Exponential model, where restart does not change the mean response-time at all. With the Lognormal model, we observe that very low restart intervals increase the mean response time drastically, and that the mean response time first drops and then grows slowly as the timeout value increases. While results with this model do not clearly indicate the existence and location of an optimum, they do show that the timeout should not be below 1 s. Finally, with the APH model we clearly observe that there is an optimum around 1.25 s. This observation is corroborated by the result of computing the optimal timeout using the algorithm from [16].

Our observations highlight the need for accurate models. In our case study we wanted to obtain insights into the applicability of restart in a scenario where service-times are distributed as in our measurements. The cPSquare model implies that restart should not be applied. Taking into account the bad fit between the empirical distribution and the cPSquare model, we should not trust this result. The Exponential model implies that restart neither helps nor hurts. However, this result has no bearing on the scenario at hand at all: As shown in [18], the memoryless property of the exponential distribution means that restart has no effect on an exponential distribution. Consequently, irrespective of the parameterisation, any exponential distribution would have yielded the same result. In contrast, the simulations with the Lognormal model point towards the existence of an optimal timeout. Since the applicability of restart strongly depends on the variance of the distribution [18], and the Lognormal model captures the variance well, we may trust this result more than those from the first two models. Finally, the experiments with the APH model, which captures the shapes of the density and distribution well, show both the existence of an optimal timeout and its location.

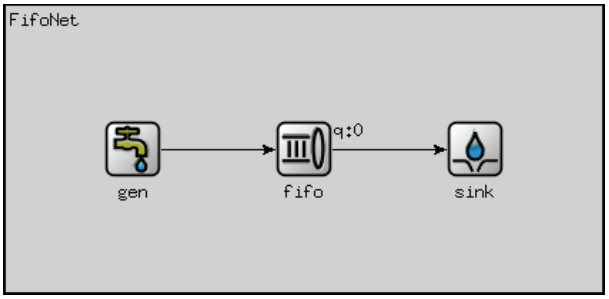

Figure 8: Queueing-Model used in the Performance Evaluation

Method	Simulation speed	Portion of sim. time
libphprng	359,364 events/sec	93%
Arrival-Process [4]	247,159 events/sec	95%

Table 1: Simulation performance with random PH-distribution.

6. PERFORMANCE OF THE LIBPHPRNG LIBRARY

In this section we demonstrate that the generation of random variates can take a significant portion of the total simulation time. Since drawing PH distributed random variates requires more effort than drawing basic random variates (like exponential, normal, etc.), it is necessary to put emphasis on the efficiency of these methods to reduce the simulation time. As mentioned earlier, libphprng is able to exploit the special structural properties of the input in order to achieve better performance, but it will also be shown that our implementation is efficient with arbitrary input (exhibiting no supported structure) as well.

Our results will be compared to the `ArrivalProcess` object proposed [4]. As MAPs are a superclass of PH arrival processes, this comparison is not entirely fair, since `ArrivalProcess` supports a much larger set of stochastic processes. It will be obvious, however, that, if PH distributions are sufficient, a much more efficient implementation can be produced.

For comparison purposes we use the simple model depicted in Figure 8. It consists of a source that generates messages with a deterministic inter-arrival time. The messages enter an infinite queue that serves them according to a FIFO discipline. The service times are PH-distributed random variables. After service, messages are transferred to the sink, where they are destroyed. To make the queue busy the deterministic inter-arrival time of the queue has been set to obtain a utilization of 96%.

In the first experiment the input PH distribution has no supported special structure (its matrix $\mathbf{Q}$ is dense). The distribution in this experiment consists of 20 states. We picked the entries of both the generator matrix and the initial vector randomly (ensuring that $(\boldsymbol{\alpha}, \mathbf{Q})$ is a phase-type distribution). The simulation ends after 30 sec of CPU time. We consider both the simulation speed, as reported by OMNeT++, and the portion of CPU time that was spent in the random-variate generator, according to `callgrind` from the Valgrind package [6].

Method	Simulation speed	Portion of sim. time
Exponential	5,473,758 events/sec	7.5%
Lognormal	4,907,375 events/sec	12.5%
libphprng	1,598,788 events/sec	70%
Arrival-Process [4]	397,440 events/sec	93%

Table 2: Simulation performance for models from Section 5.

We summarise our results in Table 1. Note that both simulations spend a significant amount of time in the random-variate generation routines. This result makes it obvious that it is definitely worth to develop and use efficient random-variate generation methods, since these methods can dominate the execution time when there are lots of random events in the model. Second, Table 1 shows that our library is almost 50% faster than the general `ArrivalProcess` module.

In our second experiment we compare the speed of random-variate generation from the different models considered in Section 5. We employ the Exponential, Lognormal, and APH models. The APH model is simulated using both the `ArrivalProcess` module and libphprng. In contrast to `ArrivalProcess`, libphprng can make use of the fact that the model is in APH form and thus has a special structure that allows both optimisation and efficient random-variate generation [10].

The results of the second experiment are shown in Table 2. First, observe that the two phase-type generators are much slower than random-variate generation from an exponential or lognormal distribution. Thus, better representation of the phenomenon under study (cf. Section 5) is bought at the cost of increased simulation time. This highlights the importance of efficient methods for PH-random-variate generation. We note that in this experiment our libphprng library is roughly four times as fast as `ArrivalProcess`. This is due to the fact that libphprng can make use of the special structures available for APH distributions.

7. CONCLUSION

In this paper we have introduced the libphprng library for random-variate generation from phase-type distributions. We have illustrated the usefulness of accurate models in system evaluation. Our case-study was complemented by a performance evaluation of the costs of random-variate generation in simulation models. We found that drawing random variates from PH distributions requires significantly more CPU time than random-variate generation from more basic distributions, like the exponential and lognormal distribution. However, methods that exploit special structures reduce the computational overhead to a reasonable level.

8. ACKNOWLEDGMENTS

This work was supported by DFG grant Wo 898/5-1. The authors would like to thank Alexandra Danilkina and Bastian Blywis for their thoughtful comments on earlier versions of this paper.

9. REFERENCES

[1] L. Bodrog, P. Buchholz, A. Heindl, A. Horváth, G. Horváth, I. Kolossváry, Z. Németh, P. Reinecke, M. Telek, and M. Vécsei. Butools: Program packages for computations with PH, ME distributions and MAP, RAP processes. http://webspn.hit.bme.hu/~butools, October 2011.

[2] A. Horváth and M. Telek. PhFit: A General Phase-Type Fitting Tool. In *TOOLS '02: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, pages 82–91, London, UK, 2002. Springer-Verlag.

[3] G. Horváth and M. Telek. Acceptance-rejection methods for generating random variates from matrix exponential distributions and rational arrival processes. In *Int. Conf. on Martix Analytic Methods (MAM)*, New York, New York, USA, june 2011.

[4] J. Kriege and P. Buchholz. Simulating Stochastic Processes with OMNeT++. In *Proceedings of the 4th International OMNeT++ Workshop (OMNeT++ 2011)*. ICST, 2011.

[5] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice*. Addison Wesley, 2001.

[6] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, June 2007.

[7] M. F. Neuts. *Matrix-Geometric Solutions in Stochastic Models. An Algorithmic Approach*. Dover Publications, Inc., New York, 1981.

[8] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[9] P. Reinecke, L. Bodrog, and A. Danilkina. Phase-type Distributions. In K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, editors, *Resilience Assessment and Evaluation: Past, Current and Future Trends*. Springer, 2012. (to appear).

[10] P. Reinecke, M. Telek, and K. Wolter. Reducing the costs of generating aph-distributed random numbers. In B. Müller-Clostermann, K. Echtle, and E. Rathgeb, editors, *MMB & DFT 2010*, number 5987 in LNCS, pages 274–286. Springer-Verlag Berlin Heidelberg, 2010.

[11] P. Reinecke, A. P. A. van Moorsel, and K. Wolter. A Measurement Study of the Interplay Between Application Level Restart and Transport Protocol. In M. Malek, M. Reitenspieß, and J. Kaiser, editors, *Service Availability. Proceedings of the First International Service Availability Symposium*, volume 3335 of *LNCS*, pages 86–100. Springer, May 2004.

[12] P. Reinecke and K. Wolter. A Simulation Study on the Effectiveness of Restart and Rejuvenation to Mitigate the Effects of Software Ageing. In *WoSAR 2010*, 2010.

[13] P. Reinecke, K. Wolter, L. Bodrog, and M. Telek. On the Cost of Generating PH-distributed Random Numbers. In G. Horváth, K. Joshi, and A. Heindl, editors, *Proceedings of the Ninth International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-9)*, pages 16–20, Eger, Hungary, September 17–18, 2009 2009.

[14] M. Telek and A. Heindl. Matching Moments for

Acyclic Discrete and Continous Phase-Type Distributions of Second Order. *International Journal of Simulation Systems, Science & Technology*, 3(3–4):47–57, Dec. 2002.

[15] A. Thümmler, P. Buchholz, and M. Telek. A Novel Approach for Phase-Type Fitting with the EM Algorithm. *IEEE Trans. Dependable Secur. Comput.*, 3(3):245–258, 2006.

[16] A. P. A. van Moorsel and K. Wolter. Analysis of Restart Mechanisms in Software Systems. *IEEE Transactions on Software Engineering*, 32(8), August 2006.

[17] A. Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, June 2001.

[18] K. Wolter. *Stochastic Models for Fault Tolerance - Restart, Rejuvenation and Checkpointing*. Springer Verlag, 2010.