

alphaFactory: a tool for generating the alpha factors of general distributions

(Tool Paper)

Elvio Gilberto Amparore, Susanna Donatelli

Università di Torino, Dipartimento di Informatica, Italy
{amparore,susi}@di.unito.it

Abstract. The Uniformization method computes the probability distribution of a CTMC of maximum rate μ at the time a general event with PDF $f(x)$ fires. Usually, $f(x)$ is taken as the deterministic distribution, leading to the computation of the CTMC probability at time t , but Uniformization may be extended to use other distributions. The extended Uniformization does not manipulate directly the distribution, as the whole computation is based on the *alpha-factors* of $f(x)$, and the maximum CTMC rate μ . This tool paper describes **alphaFactory**, a tool that computes the series of alpha-factors of a general distribution function starting from $f(x)$. The main goal of **alphaFactory** is to provide a freely available implementation for the computation of alpha-factors, to be used inside any extended Uniformization method implementation. Truncation of the infinite series of alpha-factors is determined by a novel error bound, which provides a reliable truncation point also in case of defective PDFs. **alphaFactory** can be easily integrated into other existing tools, and we show its integration inside the GreatSPN framework, to solve Markov Regenerative Stochastic Petri Nets.

Keywords: alpha-factors, general distributions, Markov Regenerative Processes, Markov Regenerative Stochastic Petri Nets, GreatSPN, extended uniformization.

1 Introduction

The Uniformization method in its basic form [16, 21] computes the probability distribution of a CTMC of maximum rate μ at a fixed time t . This method has been widely applied to the computation of the transient solutions of systems in many domains, systems expressed using a variety of formalisms (queuing networks, stochastic Petri nets, stochastic process algebra, . . .). The Uniformization method in its extended form [15] computes the probability distribution at time t , with t distributed according to a random variable. A typical application of this method is found in the steady-state solution of *Markov Regenerative Stochastic Petri Nets* (MRSPN) [11], i.e. stochastic Petri nets where transitions have exponential and generally-distributed delays, subject to the constraint that at most

one general event is enabled in any state (so called *enabling restriction*). MRSPN are a generalization of Deterministic and Stochastic Petri Nets (DSPN) [1]. Furthermore, the computation of CTMC probabilities at time t , where t is generally distributed, is a central step in the computation of the *subordinated Markov chains* of Markov Regenerative Processes (MRgP) [18].

The general event firing distribution can be described by its probability distribution function (PDF) $f(x)$. Extended Uniformization does not manipulate directly $f(x)$, but instead the whole computation is based on the *alpha-factors* of $f(x)$ and the maximum CTMC rate μ . Intuitively, the alpha-factor $\alpha(m, \mu)$ of $f(x)$ for a CTMC \mathcal{M} of maximum rate μ is the probability of taking m steps in the *uniformized DTMC* of \mathcal{M} before the generally distributed event fires. Alpha-factors form an infinite sequence for $m \in \mathbb{N}_{\geq 0}$, usually truncated according to a certain error bound ϵ . For the deterministic case, the alpha-factors reduce to a sequence of Poisson probabilities, which are usually computed with the Fox–Glynn method [13][17] due to its numerical stability. For arbitrary general distributions $f(x)$, the problem of computing alpha-factors reduces to the computation of an integral of the product of $f(x)$ with a Poisson probability. To allow a variety of general functions, the computation should do symbolic integration of $f(x)$.

Some applications also need a reliable support for *defective PDFs*, i.e. distribution functions $f(x)$ with $0 < (\int_0^\infty f(x) dx) < 1$. This makes the tool more robust when the PDF definition has small numerical error which may happen, for instance, when using fitted expolynomial distributions.

Contribution: This tool paper describes **alphaFactory**, a program that computes the alpha-factors of general distribution functions from their probability distribution function $f(x)$ and the maximum CTMC rate μ . **alphaFactory** is written in ISO C++ and has only the Boost-C++ library¹ as a dependency.

alphaFactory provides a freely available implementation for the computation of alpha-factors. The tool is designed to be used both as a standalone command line program, or linked as a component into another program, using a simple API. At its core the tool implements the definition and derivation of alpha-factors provided by German in [15], with a new truncation point of the sequence that is correct for both defective and non-defective PDFs. A proof of the correctness of the new error bound is given in Section 5.

The paper also describes how the use of **alphaFactory** has allowed the extension of the GreatSPN framework [2] to include the solution of MRSPN. Indeed, thanks to the use of **alphaFactory** the DSPN solver of GreatSPN [3] was easily transformed into an MRSPN solver.

Existing tools. There is a single tool that we know of which offers an implementation of Extended Uniformization and of the alpha-factors: SPNica [14]. SPNica is a Mathematica package written by R. German for solving MRSPN and, according to our experience, it is a very reliable solver, which unfortunately does not scale up to even moderately sized models (few hundreds of states): indeed, by definition of the author himself [14], SPNica is a prototype, a proof of concept.

¹ <http://www.boost.org/>

SPNica includes an implementation of the functions for alpha-factors computation, but their use requires the availability of the proprietary Mathematica framework. The software structure of `alphaFactory` is heavily influenced by the design choices of SPNica, but `alphaFactory` does not have the dependency on Mathematica as it is all implemented in ISO C++.

A second software that includes Extended Uniformization, again based on the technique proposed by German in [15], is TimeNET [23][7], which supports eDSPN Petri net models with general transitions (basically MRSPN nets). However, the alpha-factors module is not a separate independent component, and no clear analysis of its characteristics were possible. The expression language for general distributions supported by TimeNET is similar to that of SPNica, and also to that of `alphaFactory`, because of the common SPNica source.

Extended Uniformization is an efficient technique for MRSPN, but there are other tools that can solve MRSPN as a particular case of Non Markovian Stochastic Petri Nets, where typically the enabling restriction of only one general transition enabled in any state, is lifted. The tool WebSPN [6] represents non-Markovian transitions using state-space expansion [19], either discrete (which catches well the behaviour of low-variance distributions like the deterministic or the uniform) or continuous (which catches well high-variance distributions, like hyper-exponentials). In that case, the general distribution behaviour is approximated using a larger state space, represented as a Kronecker product. Another tool that supports general distributions is Oris [8], which again follows a different approach than the one considered in this paper, based on representation and manipulation of mathematical expressions and functions supported over polyhedral and Difference Bound Matrix (DBM) domains [10]. The Oris approach is particularly well suited for expolynomials distributions. Both WebSPN and Oris have a more expensive solution than the one based on Extended Uniformization: WebSPN in terms of larger state spaces, and Oris in computation time due to the need of performing symbolic manipulation of functions. This is certainly not surprising considering that these tools offer a solution for Petri nets with more than one general transitions enabled in a state. Another approach that targets MRSPN analysis is based on Laplace transform inversion, as described in [12].

Paper outline. Section 2 recalls the Uniformization method and its extended form, along with the alpha-factors definition, whose properties are recalled in Section 3, extended to the defective distribution case. Section 4 describes the architecture of the tool, in particular the structure of the alpha-factors evaluation. Section 5 and 6 describe the computation of the error bound and the alpha-factor algorithm. Section 7 describes how `alphaFactory` can be integrated into existing tools, a possibility that is illustrated by the integration in GreatSPN; An example of application of the tool to a real model is also shown. Finally, section 8 concludes the paper by identifying new possible research development based on the availability of `alphaFactory`.

2 Problem definition

The Uniformization method [16][21] is used to compute the *instantaneous* and *accumulated* transient probabilities of CTMCs. In its extended form [15] it is defined as follows. Let \mathbf{Q} be the infinitesimal generator of the CTMC, and let $\gamma = \max_i (-\mathbf{Q}_{i,i})$ be the maximum rate in any CTMC state. The *uniformized DTMC* \mathbf{U} of \mathbf{Q} is then defined as $\frac{1}{\mu}\mathbf{Q} + \mathbf{I}$, for an arbitrary $\mu \geq \gamma$.

Let g be the event that ends the transient evaluation of the CTMC. g can be seen as an event that is concurrently enabled with the other events represented by the CTMC \mathbf{Q} . Let $f(x)$ be the PDF of g . The PDF $f(x)$ is required to be integrable. Let $F(x)$ be the CDF of g . Given an initial probability distribution π_0 over the CTMC states, the *instantaneous* and *accumulated* transient probability distribution at the time g fires are given by:

$$\pi_g^{\text{inst}} = \pi_0 \cdot \sum_{m=0}^{\infty} \mathbf{U}^m \cdot \alpha_f(m, \mu), \quad \pi_g^{\text{acc}} = \pi_0 \cdot \sum_{m=0}^{\infty} \mathbf{U}^m \cdot \alpha_{\bar{F}}(m, \mu) \quad (1)$$

The scalar term $\alpha_f(m, \mu)$ is the *alpha-factor* of the PDF $f(x)$ for rate μ . The scalar term $\alpha_{\bar{F}}(m, \mu)$ is the alpha-factor of the *complementary CDF* (CCDF) for rate μ , where the complement CDF $\bar{F}(x)$ is defined as $(1 - F(x))$. The term π_g^{acc} is also commonly referred to as the *cumulative sojourn time* distribution.

The alpha-factors are defined as:

$$\begin{aligned} \alpha_f(m, \mu) &= \int_0^{\infty} e^{-\mu x} \frac{(\mu x)^m}{m!} \cdot f(x) dx = \int_0^{\infty} \beta(m, \mu x) \cdot f(x) dx \\ \alpha_{\bar{F}}(m, \mu) &= \int_0^{\infty} e^{-\mu x} \frac{(\mu x)^m}{m!} \cdot \bar{F}(x) dx = \int_0^{\infty} \beta(m, \mu x) \cdot \bar{F}(x) dx \end{aligned} \quad (2)$$

for $m \in \mathbb{N}_{\geq 0}$, $\mu > 0$, and with $\beta(m, \lambda) = \frac{\lambda^m e^{-\lambda}}{m!}$ the m -th Poisson probability.

In many applications, $f(x)$ is chosen to be distributed as a deterministic event that happens at time t . Hence, its PDF is a Dirac impulse $f_{\text{det}}(x) = \delta(x - t)$, and its CDF is the discontinuous function $F_{\text{det}}(x) = 1$ if $x \leq t$ and 0 otherwise. In this case, the integral can be simplified [22], leading to:

$$\alpha_{f_{\text{det}}}(m, \mu) = e^{-\mu t} \frac{(\mu t)^m}{m!}, \quad \alpha_{\bar{F}_{\text{det}}}(m, \mu) = \frac{1}{\mu} \left(1 - \sum_{k=0}^m e^{-\mu t} \frac{(\mu t)^k}{k!} \right) \quad (3)$$

However, we are interested in the computation of the alpha-factors as in Eq. (2), which is more general. From an implementation point-of-view, the two most relevant problems of computing Eq. (2) are the necessity of a symbolic integrator, and the numerical stability of the formulas. Both will be treated in Section 4.

3 Properties of alpha factors:

The work in [15, ch. 8] has derived some of the following properties of alpha-factors, that we report. Let $c = \int_0^{\infty} f(x) dx$. Then:

Property 1. The sum $\sum_{m=0}^{\infty} \alpha_f(m, \mu) = c$, for any $\mu > 0$.

Property 2. The sequence limit of $\alpha_f(m, \mu)$ is 0: $\lim_{m \rightarrow \infty} \alpha_f(m, \mu) = 0$.

Property 3. If c is finite and $f(x) > 0, \forall x \geq 0$,
then it holds that: $0 < \alpha_f(m, \mu) < c$ for all $m \geq 0$.

Property 1 is important because it gives the expected values of the entire sequence of $\alpha_f(m, \mu)$. A non-defective PDF will generate a sequence of alpha-factors $\alpha_f(m, \mu)$ that sums to 1. Alpha factors are upper bounded (Property 3) and converge to 0 (property 2). This allows to establish a truncation point M to approximate the infinite sequence.

Accumulated alpha-factors are subject to these properties:

Property 4. The accumulated alpha factor $\alpha_{\bar{F}}(m, \mu)$ is given by:

$$\alpha_{\bar{F}}(m, \mu) = \frac{1}{\mu} \left(1 - \sum_{n=0}^m \alpha_f(n, \mu) \right) = \frac{1}{\mu} \sum_{n=m+1}^{\infty} \alpha_f(n, \mu)$$

It follows that the sequence of $\alpha_{\bar{F}}(m, \mu)$ converges to 0 when $c = 1$.

Property 4 is useful since the computation of the accumulated alpha-factors can be derived from the sole sequence of $\alpha_f(m, \mu)$. Since we want to consider also defective PDFs, we extend the previous statements (established in [15]) with the following properties:

Property 5. The limit of $\alpha_{\bar{F}}(m, \mu)$ is $\frac{1-c}{\mu}$. Therefore, when $c = 1$
the sequence of $\alpha_{\bar{F}}(m, \mu)$ converges to 0.

Proof. A proof of the limit is:

$$\lim_{m \rightarrow \infty} \alpha_{\bar{F}}(m, \mu) = \frac{1}{\mu} \left(1 - \lim_{m \rightarrow \infty} \sum_{k=0}^m \alpha_f(k, \mu) \right) = \frac{1-c}{\mu}$$

Derivation uses property 4 and 2. □

Property 6. The sum of the sequence of $\alpha_{\bar{F}}(m, \mu)$ is:

$$\sum_{m=0}^{\infty} \left(\alpha_{\bar{F}}(m, \mu) - \frac{1-c}{\mu} \right) = \int_0^{\infty} x \cdot f(x) dx = \mathbb{E}[X]$$

Therefore, the sum does not depend on the value of μ .

Proof. The equivalence can be derived in this way:

$$\begin{aligned}
\sum_{m=0}^{\infty} \left(\alpha_{\bar{F}}(m, \mu) - \frac{1-c}{\mu} \right) &= \frac{1}{\mu} \sum_{m=0}^{\infty} \left(\left(1 - \sum_{k=0}^m \alpha_f(k, \mu) \right) - \left(1 - \sum_{k=0}^{\infty} \alpha_f(k, \mu) \right) \right) = \\
&= \frac{1}{\mu} \sum_{m=0}^{\infty} \sum_{k=m+1}^{\infty} \alpha_f(k, \mu) = \frac{1}{\mu} \sum_{m=1}^{\infty} m \cdot \alpha_f(m, \mu) = \\
&= \sum_{m=1}^{\infty} \int_0^{\infty} \frac{m}{\mu} \cdot e^{-\mu x} \frac{(\mu x)^m}{m!} \cdot f(x) dx = \\
&= \sum_{m=0}^{\infty} \int_0^{\infty} \beta(m, \mu x) \cdot x \cdot f(x) dx = \\
&= \int_0^{\infty} x \cdot f(x) dx = \mathbb{E}[X]
\end{aligned}$$

Derivation uses properties 4 and 1, and the trivial relation $\sum_{m=0}^{\infty} \beta(m, \mu x) = 1$. \square

Property 7. If c is finite and $f(x) > 0, \forall x \geq 0$, then it holds that:

$$\alpha_{\bar{F}}(m, \mu) \geq \frac{1-c}{\mu}, \quad \forall m \geq 0$$

Proof. Assuming $f(x) \geq 0$ for $x \geq 0$, it holds that: $\alpha_f(m, \mu) \geq 0$, for any $m \geq 0$. Therefore, property 7 is a direct consequence of property 4, which ensures that $\alpha_{\bar{F}}(m, \mu)$ values are monotonically non-increasing, and property 5, which gives the limiting behaviour of the series. \square

Property 5 shows that the sequence $\alpha_{\bar{F}}(m, \mu)$ may converge to a value that is different from 0 for defective PDFs. Property 7 establishes a lower bound for the sequence. A single truncation point for both the $\alpha_f(m, \mu)$ and the $\alpha_{\bar{F}}(m, \mu)$ sequences can then be established based on the convergent behaviours of both.

4 Architecture of alphaFactory

alphaFactory is a small tool written in ISO C++ whose sole purpose is the computation of the alpha-factors $\alpha_f(m, \mu)$ and $\alpha_{\bar{F}}(m, \mu)$, given the textual representation of function $f(x)$ and the rate μ . The tool is made of a single C++ compilation unit, plus a header file. A compile-time macro `ALPHAFACTORSLIB` controls whether the tool is compiled as a standalone command-line program, or linked inside another program. The main goal of `alphaFactory` is that of being used inside numerical solvers that use Uniformization for the computation of instantaneous/accumulated transient probabilities. The tool follows the formula derivations found in [15, pp.394-398]. For the sake of completeness, we report the formulas of the transformation rules derived in that book.

The language of the functions ϕ accepted by `alphaFactory` is the following:

$$\begin{aligned} \phi & ::= \textit{number} \mid \phi \circ \phi \mid \text{Pow}(\phi, \phi) \mid \text{Exp}(\phi) \mid \text{Log}(\phi) \mid x \mid \\ & \quad \text{I}(\psi) \mid \text{R}(\psi, \psi) \mid \text{Uniform}(\psi, \psi) \mid \text{Triangular}(\psi, \psi) \mid \\ & \quad \text{Erlang}(\psi, \psi) \mid \text{TruncatedExp}(\psi, \psi) \mid \text{Pareto}(\psi, \psi) \\ \psi & ::= \textit{number} \mid \psi \circ \psi \mid \text{Pow}(\psi, \psi) \mid \text{Exp}(\psi) \mid \text{Log}(\psi) \end{aligned}$$

where $\circ \in \{+, -, *, /\}$. Number literals are floating point real numbers. The term x is the integral variable. The functions `Pow`, `Exp` and `Log` are the power, the exponential and the natural logarithm, respectively. The function `I`(ϕ) is a Dirac delta unit impulse $\delta(x - \phi)$. It represents the concentration of the probability mass at single point ϕ , and it is interpreted as if the probability of a firing at time ϕ is 1. The function `R`(a, b) is a rectangular signal that assumes value 1 over the range $[a, b]$, and 0 outside that range. The language ψ is just a simplified language for algebraic expressions over constant terms.

The remaining elements of ϕ are non-primitive functions:

- `Uniform`(a, b) is the uniform distribution, defined as $1/(b - a) * \text{R}(a, b)$.
- `Triangular`(a, b) is the triangular distribution, defined as:

$$\frac{4 * (x - a)}{(a - b)^2} * \text{R}\left(a, \frac{a + b}{2}\right) - \frac{4 * (x - b)}{(a - b)^2} * \text{R}\left(\frac{a + b}{2}, b\right)$$

`Erlang`(λ, r) is the Erlang function with rate λ and r phases, defined as:

-

$$\frac{\lambda^r}{(r - 1)!} * x^{r-1} * e^{-\lambda * x}$$

- `TruncatedExp`(λ, t) is the exponential distribution of rate λ truncated at time t , defined as: $\lambda * e^{-\lambda * x} * \text{R}(0, t) + e^{-\lambda * t} * \text{I}(t)$. It is obtained by multiplying the exponential distribution with a rectangular signal `R`(0, t), so that after t the distribution is truncated. To compensate the truncation, an impulse of probability $e^{-\lambda * t}$ happens at time t , so that the overall truncated exponential is not a defective PDF.

`Pareto`(k, s) is the Pareto distribution of real scale parameter k and shape parameter s , $s \in \mathbb{N}_{>0}$, defined

-

$$\text{as: } \begin{cases} \frac{s * k^s}{x^{s+1}} & \text{if } x > k \\ 0 & \text{if } x \leq k \end{cases}$$

These simple building blocks allow to define common distribution functions, like expopolynomials distributions.

The evaluation of a function $f(x)$ starts by building the Abstract Syntax Tree (AST) of the formula. The tool uses a recursive descent parser for this task. ASTs are made by just three node types:

1. *Term* leaf nodes that contain real values.

2. *Symbol* leaf nodes that contain the integration variable x .
3. *Function* nodes, that are n -ary operators for a single arithmetic operand. The function operand is one among $\{+, -, *, /, \text{Pow}, \text{Exp}, \text{Log}, \text{I}, \text{R}\}$, or a non-primitive operand among $\{\text{Uniform}, \text{Triangular}, \text{Erlang}, \text{TruncatedExp}, \text{Pareto}\}$.

Once the parser has finished, it is possible to manipulate the expression of $f(x)$ at the AST level. The tool has four main AST-manipulation functions: `evaluate(e)`, `simplify(e)`, `integrate(e)` and `moment(e, k)`.

- `evaluate(e)` does the numerical evaluation of e . The expression e must have only constant terms or function, i.e. it cannot have the integration variable x .
- `simplify(e)` implements polynomial simplification and rearrangement of the expression argument e into a canonical form. It works by applying a fixed set of transformation rules. Rules use pattern matching and node substitution, and are encoded inside the function. For instance, the function $x * x$ is canonicalized as x^2 , or the function x^0 is simplified as 1. The transformation rules are:

```

simplify( $\psi$ ) → evaluate( $\psi$ )
simplify( $\phi * 1$  or  $\phi + 0$ ) →  $\phi$ 
simplify( $\phi * \phi * \dots * \phi$ ) →  $\phi^n$ 
simplify( $\phi_1 * (\phi_2 + \phi_3)$ ) →  $\phi_1 * \phi_2 + \phi_1 * \phi_3$ 
simplify( $\phi^0$ ) → 1
simplify( $\phi^1$ ) →  $\phi$ 
simplify(any function  $\phi$ ) → recursively apply simplify on  $\phi$  operands

```

The function is also responsible for the expansion of the non-primitive functions `Uniform(a, b)`, `Triangular(a, b)`, `Erlang(λ, r)`, `TruncatedExp(λ, t)` `Pareto(k, s)`, and operand reordering (terms in a product are always arranged following a specified canonical order).

- `integrate(e)` computes the symbolic integral $\int_0^\infty e(x) dx$ using AST manipulation, assuming that the expression is in canonical form (obtained by `simplify`). As before, it implements a set of transformation rules to compute the symbolic result. The implemented rules are:

$$\int_0^\infty f(x) dx$$

```

integrate( $x$ ) →  $\frac{1}{2} * x^2$ 
integrate( $t$ ) →  $t * x$ 
integrate( $e^x$ ) →  $e^x$ 
integrate( $e^{k*x}$ ) →  $\frac{1}{k} * e^{k*x}$ 
integrate( $x^m$ ) →  $\frac{1}{m+1} * x^{m+1}$ 
integrate( $c * I(t)$ ) →  $c$ 
integrate( $c * x * I(t)$ ) →  $c * t$ 
integrate( $c * e^{l*x} * x^h * R(0, b)$ ) →  $c * (-l)^{-h-1} * (\Gamma(h+1) - \Gamma(h+1, -b * l))$ 

```

$\text{integrate}(c * \mathbf{R}(a, b)) \rightarrow \text{integrate}(c * \mathbf{R}(0, b)) - \text{integrate}(c * \mathbf{R}(0, a))$
 $\text{integrate}(c * \phi) \rightarrow c * \text{integrate}(\phi)$
 $\text{integrate}(\phi_1 - \phi_2) \rightarrow \text{integrate}(\phi_1) - \text{integrate}(\phi_2)$
 $\text{integrate}(\text{sum of } \phi_i) \rightarrow \text{sum of } \text{integrate}(\phi_i)$

where c, t are constant terms, $\Gamma(z)$ and $\Gamma(s, z)$ are the complete and the upper incomplete gamma functions, respectively. The rules are actually standard integration rules. A product with a rectangular signal $\mathbf{R}(a, b)$ is equivalent to computing the integral over the $[a, b]$ range instead of the $[0, \infty)$ range.

- **moment**(e, k) is the moment generating function. It computes the k -th moment of the random variable with PDF e as: `evaluate(integrate($e * x^k$))`.

Once `alphaFactory` has built and simplified the AST e of the expression of $f(x)$, it starts by computing the 0 and 1 moments of $f(x)$. The 0 moment, being the area of $f(x)$, is checked to be 1. If it is not, $f(x)$ is a defective PDF, and a warning message is printed. In some applications, defective PDFs are allowed, so the tool does not stop for this condition. Property 1 also tells that the 0 moment gives the sum of the sequence of $\alpha_f(m, \mu)$, which is used for error bound. The first moment, being $\mathbb{E}[X]$, can be used to bound the sum of $\alpha_{\bar{F}}(m, \mu)$ for non-defective PDFs (property 6).

At this point, alpha-factors can be computed for AST e using the recursive function `alpha`(m, μ, e). This function computes the m -th factor for a CTMC of rate μ . The function `alpha`(m, μ, ϕ) applies these transformation rules to ϕ :

$\text{alpha}(\phi_1 + \phi_2) \rightarrow \text{alpha}(m, \mu, \phi_1) + \text{alpha}(m, \mu, \phi_2)$
 $\text{alpha}(c * \mathbf{I}(a)) \rightarrow c * \beta(m, \mu * a)$
 $\text{alpha}(\mathbf{I}(a)) \rightarrow \beta(m, \mu * a)$
 $\text{alpha}(c * \mathbf{R}(a, b)) \rightarrow \text{alpha}(m, \mu, c * \mathbf{R}(0, b)) - \text{alpha}(m, \mu, c * \mathbf{R}(0, a))$
 $\text{alpha}(\mathbf{R}(a, b)) \rightarrow \text{alpha}(m, \mu, \mathbf{R}(0, b)) - \text{alpha}(m, \mu, \mathbf{R}(0, a))$
 $\text{alpha}(c * e^{l*x} * x^h) \rightarrow c * \gamma_a(m, \mu, h, -l, a)$
 $\text{alpha}(c * e^{l*x} * x^h * \mathbf{R}(0, a)) \rightarrow c * \gamma_\infty(m, \mu, h, -l)$
 $\text{alpha}(\text{Pareto}(k, s)) \rightarrow - \frac{e^{-\mu * k^m * s * \mu^m}}{(m-s) * m!}$

These formulas are directly derived by solving the integral of Eq. 2 over the function argument, and are taken from [15, p. 396]. The rest of the evaluation of the alpha-factors relies on four recursive functions β , γ_a , γ_∞ and η , that are needed to evaluate the integral terms symbolically. Memoization of the partially evaluated results is employed to speed up the computation. The m -th Poisson probability $\beta(m, \lambda)$ function is implemented using the usual recursive relation:

$$\beta(m, \lambda) = \begin{cases} e^{-\lambda} & \text{if } m = 0 \\ \frac{\lambda * \beta(m-1, \lambda)}{m} & \text{otherwise} \end{cases}$$

The two γ factors are derived by integrating the expolynomial equations (6th and 7th rules of `alpha`) with the Poisson function, expanding Eq. (2). The full

derivation can be found in [15, pp. 154–155]. This results in a recursive relation for γ_∞ and γ_a , defined as:

$$\gamma_\infty(m, \mu, h, l) = \begin{cases} \frac{h!}{(\mu + l)^{h+1}} & \text{if } m = 0 \\ \frac{m + h}{m} * \frac{\mu}{\mu + l} * \gamma_\infty(m - 1, \mu, h, l) & \text{otherwise} \end{cases}$$

and:

$$\gamma_a(m, \mu, h, l, a) = \begin{cases} \frac{h!}{(\mu + l)^{h+1}} \left(1 - \sum_{i=0}^m \beta(i, (\mu + l)a) \right) & \text{if } m = 0 \\ \frac{m + h}{m} * \frac{\mu}{\mu + l} * \gamma_a(m - 1, \mu, h, l, a) - \eta(m, \mu, h, l, a) * \beta(m, (\mu + l) * a) & \text{otherwise} \end{cases}$$

where $\gamma_\infty(m, \mu, h, l)$ is equivalent to $\gamma_a(m, \mu, h, l, \infty)$. The η factor is defined as:

$$\eta(m, \mu, h, l, a) = \begin{cases} \frac{a^h}{\mu + l} & \text{if } m = 0 \\ \eta(m - 1, \mu, h, l, a) * \frac{\mu}{\mu + l} & \text{otherwise} \end{cases}$$

5 Bounds of α -tails:

Since the series of alpha-factors $\alpha_f(m, \mu)$ and $\alpha_{\bar{F}}(m, \mu)$ are infinite, defined for all $m \in \mathbb{N}_{\geq 0}$, it is necessary to approximate the sequence up to a right truncation point M . Using an accuracy parameter ϵ , the sequence of instantaneous alpha-factors $\alpha_f(m, \mu)$ can be truncated at M :

$$\sum_{m=M+1}^{\infty} \alpha_f(m, \mu) < \epsilon \Rightarrow \sum_{m=0}^M \alpha_f(m, \mu) = c - \epsilon$$

The relation, derived in [15, p. 152], is directly derived from Property 1, which ensures that the sum of the entire sequence is c , and Property 2 which ensures that the sequence converges to 0.

The right truncation point of the sequence of accumulated alpha-factors $\alpha_{\bar{F}}(m, \mu)$ is slightly different, since they converge to 0 only for non-defective PDFs (i.e. $c = 1$). In the general case, the sequence converges to $\frac{1-c}{\mu}$, as stated in Property 5. Therefore, a truncation point M' can be set such that:

$$\sum_{m=M'+1}^{\infty} \left(\alpha_{\bar{F}}(m, \mu) - \frac{1-c}{\mu} \right) < \epsilon \Rightarrow \sum_{m=0}^{M'} \left(\alpha_{\bar{F}}(m, \mu) - \frac{1-c}{\mu} \right) = \mathbb{E}[X] - \epsilon$$

using ϵ as an absolute error for the summation. Therefore, a method can be devised that ensures that both sequences are truncated below the requested accuracy ϵ using $R = \max(M, M')$ as the truncation point. This requires to know both the 0-moment c and the first moment $\mathbb{E}[X]$ of the random var. X .

6 Alpha-factors computation algorithm

After having introduced all the required elements, it is now possible to show the core alpha-factors computation function. The pseudo-code of the method is shown in Algorithm 1.

Algorithm 1 Pseudocode of the alpha-factors generation function.

Function `compute_alpha_factors_dbl`(f, μ, ϵ):

```

 $e \leftarrow \text{simplify}(\text{parse}(f))$ 
 $c \leftarrow \text{moment}(e, 0)$ 
 $\mathbb{E}[X] \leftarrow \text{moment}(e, 1)$ 
 $err_f \leftarrow c$ 
 $err_{\bar{F}} \leftarrow \mathbb{E}[X]$  if  $\mathbb{E}[X] \neq \infty$  else 0
 $m \leftarrow 0$ 
 $value_{\bar{F}} \leftarrow \frac{1}{\mu}$ 
while ( $err_f > \epsilon \vee err_{\bar{F}} > \epsilon$ ):
   $\alpha_f(m, \mu) \leftarrow \text{alpha}(m, \mu, e)$ 
   $value_{\bar{F}} \leftarrow value_{\bar{F}} - \frac{\alpha_f(m, \mu)}{\mu}$ 
   $\alpha_{\bar{F}}(m, \mu) \leftarrow value_{\bar{F}}$ 
   $err_f \leftarrow err_f - \alpha_f(m, \mu)$ 
   $err_{\bar{F}} \leftarrow err_{\bar{F}} - (\alpha_{\bar{F}}(m, \mu) - \frac{1-\epsilon}{\mu})$ 
   $m \leftarrow m + 1$ 
return  $\langle \alpha_f, \alpha_{\bar{F}} \rangle$ 

```

The algorithm is an implementation of the method defined in [15, p. 394], with the new bound R described in Section 5. It starts by parsing the function and building the AST. It then computes the first two moments, initializing the error control variables err_f and $err_{\bar{F}}$ with the moment values. The function then iterates until both error thresholds are below ϵ . At each iteration, the alpha-factor $\alpha_f(m, \mu)$ is computed. The accumulated alpha-factor $\alpha_{\bar{F}}(m, \mu)$ is derived implicitly by subtracting incrementally all the instantaneous alpha-factors, starting from the initial value $\frac{1}{\mu}$. This follows Property 4. The algorithm then subtracts the alpha-factors from the err_f and $err_{\bar{F}}$ control variables, and repeats. Even if it is true that the sequence of $\alpha_{\bar{F}}(m, \mu)$ can be completely derived from the sequence of $\alpha_f(m, \mu)$, it is important to compute both together, in order to establish the single truncation point R that guarantees that both sequence have an absolute error below ϵ .

Numerical precision. The `alphaFactory` implementation uses multi-precision floating point. Floating point precision is controlled using the `MPFLOAT_PRECISION` constant, which is defaulted to 1024 bits. This allows to treat factors with large difference in magnitude, without a dangerous loss of precision. Of course, a different strategy (like the one used by the previously mentioned Fox–Glynn method) could be devised to improve the accuracy without resorting to multi-precision arithmetic. In particular, the error control variables are subject to multiple subtractions, which could result in numerical instability without enough precision.

The strategy used by the Fox–Glynn method and many other Uniformization methods is that of starting from the central value of the $\beta(m, \mu)$ series, and the computing the left and right tails independently. Unfortunately, it is hard to derive a single computational strategy that is at the same time general w.r.t $f(x)$ and that computes the values in a non-sequential order. However, if we restrict to some specific classes of general functions (like expolynomials), a better strategy could be devised.

Tool Validation. The tool has been validated against the results computed by SPNica and by a direct evaluation of the alpha-factor formulas in Mathematica. Results confirm the correctness of the tool on a benchmark of distributions. The tool is also equipped with a small unit test, that runs using the `test` command line argument. The unit test verifies that the tool re-computes correctly the alpha-factors from a small set of PDFs, and compares the obtained results with the values computed by evaluating the corresponding formula integrals in Mathematica.

6.1 Example of running alphaFactory

Figure 1 shows four by three plots obtained running alphaFactory on four PDFs.

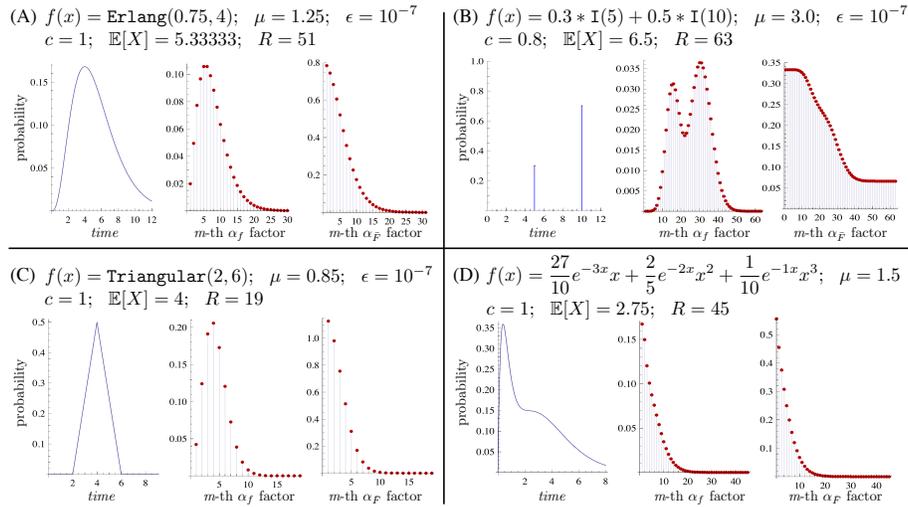


Fig. 1. Alpha-factor distributions generated by alphaFactory from four PDFs.

For each PDF three graphs are shown: the PDF itself and instantaneous and accumulated alpha-factors (left to right). The plot header reports the computed values for c , $\mathbb{E}[X]$, and the truncation point R . PDF (A) is an Erlang distribution of rate 0.75 and 4 phases. Alpha factors are computed for a CTMC of rate $\mu = 1.25$, with accuracy $\epsilon = 10^{-7}$. Values for (A) are obtained running the tool

from the command line:

```
./alphaFactory 'Erlang(0.75, 4)' 1.25 0.0000001
```

The second line in the header of plot (A) reports the computed values for c and $\mathbb{E}[X]$, and the truncation point R . PDF (B) is a linear combination of two Dirac impulses, which is intuitively a random choice between two deterministic events. The combination is defective, as can be seen by the computed integral value $c = 0.8$. PDF (C) is a triangular distribution, which is actually decomposed into a polynomial combination of two rectangular signals. Finally, PDF (D) is an exponential distribution. When run from the command line as a standalone tool, `alphaFactory` first writes the two moments of the function, followed by the number of factors and by a list of one factor per line. The generated alpha-factors can be used directly inside a Extended Uniformization method, following Eq. (1).

7 Integration of `alphaFactory` into other softwares

As mentioned before, `alphaFactory` can be included in another software project as a static library or as a C++ compilation unit, to be used non-interactively. The Application Programming Interface of `alphaFactory` is minimal and it is made by just two exported functions:

- `verify_alpha_factors_expr` takes in input a character strings and verifies if it is a valid input expression for the tool, if it is defective and if the tool is capable of integrating that function. This function is useful for expression validation, for instance during model loading, or within a graphical editor.
- `compute_alpha_factors_dbl(const char* fg, double mu, double eps)` computes the alpha factor distributions as in Eq. (2). The function returns a pair of vectors, containing the values of $\alpha_f(m, \mu)$ and $\alpha_{\bar{f}}(m, \mu)$. Argument `mu` is the uniformized CTMC rate. Argument `eps` specifies the computation accuracy ϵ .

The minimalist API allows to integrate the tool into other softwares that use the Uniformization method with a minimal effort. We have integrated `alphaFactory` in our DSPN solver [3], making it capable of solving MRSPN models.

7.1 Integration of `alphaFactory` into GreatSPN

We now show a small example of an application of generalized functions in MRSPN, solved with the help of `alphaFactory`. The tool has been integrated inside the DSPN solver of GreatSPN [3], which is now capable of solving MRSPN Petri net models with general transitions with the usual *enabling restriction*. We consider the case [5] of a multi-utility company, who works in a specified geographical area of about 2200km², with 531K inhabitants. The problem the company is interested in is the optimal allocation of human resources, in order to comply with the national regulation authority rules, which require that in case

of call from a client of a detected leak of gas, a technician must be on-site in less than 1 hour. When on site, the technician first secures the problem. Then, he may decide to actually fix the problem, if there are no other open requests. Otherwise, he leaves the site, sending an external plumber to do the fix.

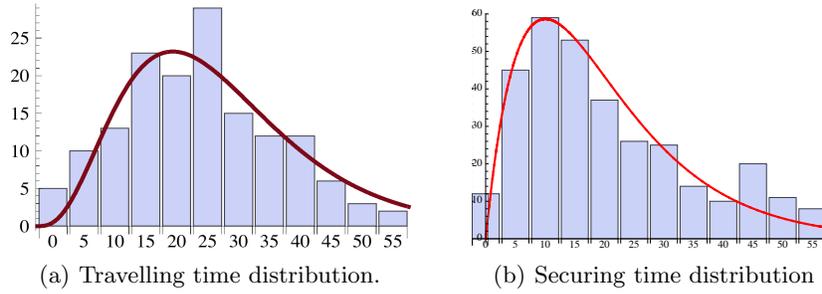


Fig. 2. Data samples from which the general distributions were derived.

Figure 2 shows the travelling time distribution and securing time distribution, extracted from the company log (about 600 samples). We used the Erlang distribution for data fitting, deriving using the company logs an `Erlang(1.15, 3)` for the travelling time, and `Erlang(1.6, 4)` for the securing time. We do not have precise timing for the repairing phase, but the company told us that the time is usually in the order of 10-30 minutes. Hence, we modeled the repairing time with a `Uniform(10, 30)`.

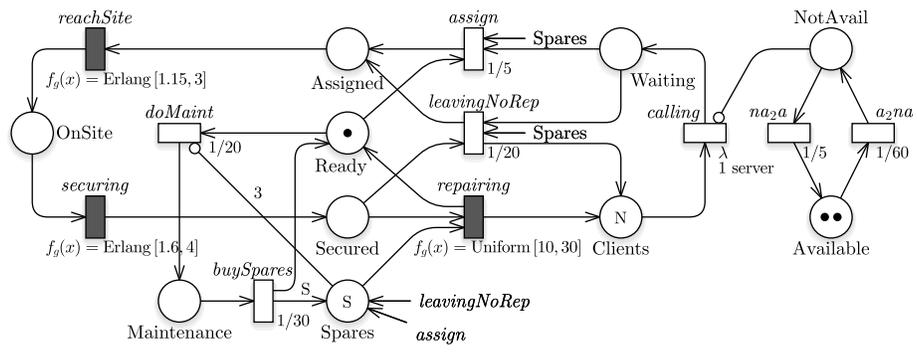


Fig. 3. Simplified MRSPN of the multi-utility company repairmen problem used in [5].

The MRSPN model of the simplified multi-utility company is depicted in Figure 3. Distribution functions are written as annotations of the general transition (black rectangles) in the graphical representation of the MRSPN. The functions are passed verbatim to `alphaFactory` during the solution process. We run the steady-state MRSPN solver on the model, for various client *inter-arrival*

times (IAT), to study the behaviour of the system and the load balance. The goal of the analysis is to find the client IAT value where the probability of having another client being served is below 10%.

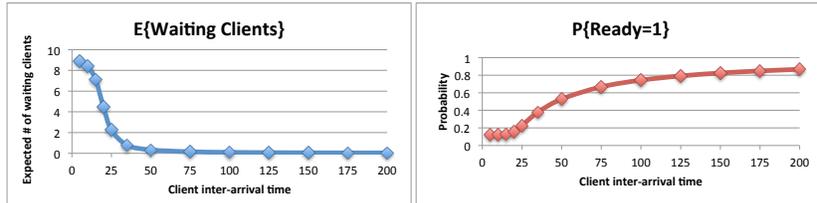


Fig. 4. MRSNP performance indexes computed in steady-state.

Figure 4 shows the expected number of clients in the system (left) and the probability distribution of finding the technician idle (right), for increasing values of the client IAT. We observe that with a client IAT of ~ 28 minutes the probability of finding another client in queue is about 1. The probability drops rapidly, and is below 0.1 with an inter-arrival time of 100 minutes. The probability of finding the technician idle is about $\sim 74\%$ when the client IAT is 100 minutes. We therefore conclude that the target system load is at a client IAT value of about 100 minutes, considering the provided travelling time distributions and securing time distributions, when a single technician is available.

Overall, the MRSNP has 1116 markings. The steady state solution time for a single run takes ~ 0.2 seconds. In general, we may say that in most cases the cost of running `alphaFactory` is negligible, since it needs to be run just once for each $f(x)$ and μ pair.

8 Conclusions and Future works

This paper describes the tool `alphaFactory`, whose purpose is the generation of the alpha-factors of a general distribution $f(x)$. Alpha-factors are used by the Extended Uniformization method to compute instantaneous/accumulated transient probabilities at time t , with t distributed as $f(x)$. The main purpose of `alphaFactory` is to make a standalone, re-usable component to ease the implementation of Extended Uniformization. This is mostly of interest for MRgP solvers, MRSNP and DSPN tools, and tools that compute transient CTMC probabilities.

We plan to extend also our Phase-Mission systems [20, 4] tools with `alphaFactory`, to allow the definition of phases of general duration. This extension is useful for many processes (like workflow processes) where phases are more naturally described with uniform distributions or distributions fitted from data. We also plan to investigate the use of `alphaFactory` inside other tools that support more sophisticated data structure representations, like state-spaces in Kronecker form [9].

Availability. The source code of `alphaFactory` is distributed under a modified BSD license, and can be found at: <https://github.com/amparore/alphaFactory>.

References

1. Ajmone Marsan, M., Chiola, G.: On Petri nets with deterministic and exponentially distributed firing times. In: *Advances in Petri Nets. Lecture Notes in Computer Science*, vol. 266/1987, pp. 132–145. Springer Berlin / Heidelberg (1987)
2. Amparore, E.G., Balbo, G., Beccuti, M., Donatelli, S., Franceschinis, G.: 30 Years of GreatSPN, chap. In: *Principles of Performance and Reliability Modeling and Evaluation: Essays in Honor of Kishor Trivedi*, pp. 227–254. Springer, Cham (2016)
3. Amparore, E.G., Donatelli, S.: DSPN-Tool: a new DSPN and GSPN solver for GreatSPN. In: *International Conference on Quantitative Evaluation of Systems*. pp. 79–80. IEEE Computer Society, Los Alamitos, CA, USA (2010)
4. Amparore, E.G., Donatelli, S.: Efficient solution of extended Multiple-Phased Systems. In: *10th Valuetools Conference*. pp. 125–132. EAI (2016)
5. Amparore, E.G., Donatelli, S., Landini, E.: Modelling and Evaluation of a Control Room application. In: *Int. Conf. on Application and Theory of Petri Nets (ATPN) 2017 (Accepted)*. Springer, Zaragoza, Spain (Jun 2017)
6. Bobbio, A., Puliafito, A., Scarpa, M., Telek, M.: WebSPN: Non-Markovian Stochastic Petri Net Tool. In: *18th Conf. on Application and Theory of Petri Nets (1997)*
7. Bodenstern, C., Zimmermann, A.: TimeNET optimization environment: batch simulation and heuristic optimization of SCPNs with TimeNET 4.2. In: *8th Int. Conf. on Performance Evaluation Methodologies and Tools*. pp. 129–133. ICST (2014)
8. Bucci, G., Carnevali, L., Ridi, L., Vicario, E.: Oris: a tool for modeling, verification and evaluation of real-time systems. *International Journal on Software Tools for Technology Transfer* 12(5), 391–403 (2010)
9. Buchholz, P.: Markov matrix market.
`ls4-www.cs.tu-dortmund.de/download/buchholz/struct-matrix-market.html`
10. Carnevali, L., Ridi, L., Vicario, E.: A Framework for Simulation and Symbolic State Space Analysis of Non-Markovian Models, pp. 409–422. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
11. Choi, H., Kulkarni, V.G., Trivedi, K.S.: Markov regenerative stochastic Petri nets. *Performance Evaluation* 20(1-3), 337–357 (1994)
12. Dingle, N.J., Harrison, P.G., Knottenbelt, W.J.: Response time densities in generalised stochastic Petri nets. In: *Workshop on Software and Perf.* pp. 46–54 (2002)
13. Fox, B.L., Glynn, P.W.: Computing Poisson probabilities. *Communications of the ACM* 31(4), 440–445 (1988)
14. German, R.: Markov Regenerative Stochastic Petri Nets with general execution policies: Supplementary variable analysis and a prototype tool. *Performance Evaluation* 39(1-4), 165–188 (Feb 2000)
15. German, R.: *Performance Analysis of Communication Systems with Non-Markovian Stochastic Petri Nets*. John Wiley & Sons, Inc., New York, USA (2000)
16. Grassmann, W.: Transient solutions in markovian queueing systems. *Computers and Operations Research* 4(1), 47–53 (1977)
17. Jansen, D.N.: Understanding Fox and Glynn’s “Computing Poisson probabilities”. Tech. rep., Nijmegen : ICIS R11001 (2011)
18. Kulkarni, V.G.: *Modeling and analysis of stochastic systems*. Chapman & Hall Ltd., London, UK (1995)
19. Longo, F., Scarpa, M.: Two-layer symbolic representation for stochastic models with Phase-type distributed events. *International Journal of Systems Science* 46(9), 1540–1571 (Jul 2015)

20. Mura, I., Bondavalli, A., Zang, X., Trivedi, K.S.: Dependability modeling and evaluation of phased mission systems: a DSPN approach. In: Int. Conf. on Dependable Computing for Critical Applications (DCCA). pp. 299–318. IEEE (1999)
21. Stewart, W.J.: Introduction to the numerical solution of Markov chains. Princeton University Press (1994)
22. Trivedi, K.S., Reibman, A.L., Smith, R.: Transient analysis of Markov and Markov reward models. In: Computer Performance and Reliability '87. pp. 535–545 (1987)
23. Zimmermann, A., Freiheit, J., German, R., Hommel, G.: Petri Net Modelling and Performability Evaluation with TimeNET 3.0. In: Proc. of the 11th Int. Conf. on Modelling Techniques and Tools (TOOLS). pp. 188–202. Springer, London (2000)