Automated Rare Event Simulation for Stochastic Petri Nets

Daniël Reijsbergen, Pieter-Tjerk de Boer,
Werner Scheinhardt, and Boudewijn Haverkort

Center for Telematics & Information Technology,
University of Twente, Enschede, The Netherlands

Abstract. We introduce an automated approach for applying rare event simulation to stochastic Petri net (SPN) models of highly reliable systems. Rare event simulation can be much faster than standard simulation because it is able to exploit information about the typical behaviour of the system. Previously, such information came from heuristics, human insight, or analysis on the full state space. We present a formal algorithm that obtains the required information from the high-level SPN-description, without generating the full state space. Essentially, our algorithm reduces the state space of the model into a (much smaller) graph in which each node represents a set of states for which the most likely path to failure has the same form. We empirically demonstrate the efficiency of the method with two case studies.

1 Introduction

The first step towards the analysis of a highly dependable system is its specification as a state transition system. When the behaviour of the system is stochastic, a common model is the (discrete- or continuous-time) Markov chain. The state space of the Markov chain can be very large (even infinite), but the chain often has enough structure to allow for implicit specification using a high-level description language. Classical examples of such languages are stochastic Petri nets (SPNs) [1], and stochastic activity networks [24].

Given an SPN, one specifies a measure for the performance of the highly dependable system in terms of its stochastic properties. The measure that we focus on in this paper is the probability that one reaches a certain uncommon set of states (the *goal* set) before reaching a more typical set (the *taboo* set). This probability can be interesting by itself, but is particularly interesting as it appears in expressions for, e.g., the Mean Time To Failure, the time-bounded unreliability and the steady-state unavailability. Numerical methods for computing this probability are well-established, but since they operate mostly on the complete state space, which is often very large, they can be computationally infeasible (an issue commonly referred to as the *state space explosion problem*).

A remedy is then to use stochastic (discrete-event) simulation [16], i.e., repeatedly generating random executions of the system model and using the average behaviour observed in the executions to obtain an estimate of the probability of

K. Joshi et al. (Eds.): QEST 2013, LNCS 8054, pp. 372–388, 2013.

interest. Discrete-event simulation can be carried out on the level of the SPN and only requires that, overall, the current state in the system is stored instead of the entire state space. A common problem is that when the goal set is rare (like failure states in a highly reliable system) one needs an infeasibly large number of executions to obtain an accurate estimate.

In order to reduce the number of executions needed, several *efficient simulation methods* have been proposed in the past few decades. They can be largely divided into two main categories: *importance sampling* methods [10], and *RESTART* and *multilevel splitting* [8,27] methods. Both can use knowledge of the typical *paths toward* or the *distance to* the goal set to their advantage. Several techniques have been implemented in the past two decades [4,12,21,26,28], but all of these rely on user input or the adequacy of heuristics in order to perform well.

In this paper we show that the required information can be obtained in an automated way from the SPN and the description of the goal and taboo sets. As such, we present a formal algorithm that achieves this. It uses the structure of the SPN to divide the implied state space into zones, in each of which the distance to the goal set can be expressed using the same distance function. In this way we can find the overall distance function, which can then be used in an efficient simulation procedure. We demonstrate the potential gain of the method, both for a simple example (which is also used as a running example throughout the paper), and a more demanding model of a multicomponent system with interdependent component types.

The structure of the rest of this paper is as follows: in Section 2, we explain the position of this paper in the context of the earlier scientific literature. In Section 3 we discuss the exact definition of an SPN that we will use throughout this paper, and explain the foundations of (rare event) simulation. The core algorithm that determines the distance function in an automated way is the topic of Section 4. Section 5 contains a simulation study involving the simple model and a more realistic model. In Section 6, we discuss a few challenges associated with the new method and ways to overcome them, before we conclude the paper.

2 Context within the Literature

One way to obtain knowledge about the way the system progresses toward the goal set is to divide the transitions in the SPN into failure and repair transitions that respectively take the system towards or away from the goal set. One can then apply *failure biasing* [25]. This has been implemented in, among others, SAVE (see [4]) and in UltraSAN [21], the predecessor to the tool Möbius [6].

One variation of failure biasing that is especially noteworthy in the context of this paper is distance failure biasing [5]. It is based on a notion of distance similar to the one we introduce in Section 3. However, the technique presented in [5] can only be applied to a very narrow class of models (namely models with independent component types) and the gains compared to failure biasing may not justify the numerical effort of the minimal cut algorithm that is used (see also the discussion in [20]).

Another technique is to split the simulation effort into two different stages: one to obtain information about the typical behaviour to the rare set and one to use this knowledge in an importance sampling scheme. This idea forms the basis of the *cross-entropy method* for importance sampling [23] [11] and Kelling's framework for RESTART in SPN [14]. The cross entropy method has recently been implemented in the PLASMA-platform [12].

For RESTART and splitting, one implicitly divides the state space of the model in several *level sets.* Some examples of how to determine these level sets are to let the user specify them by hand [18,26], or to use a two-step approach similar to the one underlying the cross-entropy method [14]. The splitting framework has been implemented in the Stochastic Petri Net Package [26] and the tool TimeNet [28]. The methods based on this principle are largely heuristic in nature.

3 Model and Preliminaries

The outline of this section is as follows. In Section 3.1, we describe the type of Petri nets we consider throughout the paper. In Section 3.2, we illustrate this with an example that we use throughout this paper. In Section 3.3, we discuss the performance property of interest, and we discuss simulation in Section 3.4.

3.1 Discrete-Time Stochastic Petri Nets

We assume that the reader is familiar with the general concept of a Petri net (if not, see e.g. [19]). We use Multi-Guarded Petri Nets as in [13], although we extend the net with marking-dependent firing rates for the transitions. We define a Petri net to be $(P, T, Pre, Post, G)$, where

- $P = \{1, 2, \ldots, |P|\}$ denotes the set of *places*,
- $T = \{t_1, \ldots, t_{|T|}\}$ denotes the set of *transitions*,
- $Pre : P \times T \to \mathbb{N}$ and $Post : P \times T \to \mathbb{N}$ are the *pre-* and *post- incidence functions.*[1]
- G denotes the set of guards (more details are given below).

We are interested in the (embedded) *discrete-time behaviour* of the Petri net; let $X_i(n)$ be the number of tokens in place i after the n-th time a transition is fired, $n \in \mathbb{N}$. Let $\boldsymbol{X}(n) = (X_1(n), \ldots, X_{|P|}(n))^{\mathrm{T}}$ be the *marking* (or *state*) of the net at time n. Let $\mathcal{X} = \mathbb{N}^{|P|}$ be the set of all possible markings; then we let transition t_i have exponential rate $\lambda_i(\boldsymbol{x})$ with $\boldsymbol{x} \in \mathcal{X}$. Importantly, although we allow the rates λ_i to depend on the marking $\boldsymbol{x}$ we assume that these rates are functions of ϵ (see below) and that numbers r_i exist such that for all $\boldsymbol{x} \in \mathcal{X}$, $\lambda_i(\boldsymbol{x}) = \Theta(\epsilon^{r_i})$, i.e.,

$$0 < \lim_{\epsilon \downarrow 0} \frac{\lambda_i(\boldsymbol{x})}{\epsilon^{r_i}} < \infty$$

---

[1] We use $\mathbb{N} = \{0, 1, 2, \ldots\}$.

The rate $\lambda_i(\boldsymbol{x}(n))$ determines the relative likelihood of the transition to *fire* at step n. The number ϵ is the so-called *rarity parameter*, which is typically a small number that signifies how rare the event of interest is.

When transition t fires, the marking changes as follows: $Pre(p,t)$ tokens are removed from place p while $Post(p',t)$ tokens are added to place p'. A transition cannot fire if this would result in a negative number of tokens in a place, nor can it fire when one of its guards is not enabled (as discussed below). The guards can be described in terms of *constraints*, a concept that we will use often in Section 4. A constraint $c = (\boldsymbol{\alpha}, \beta, \bowtie)$ is an element of $\mathbb{Z}^{|P|} \times \mathbb{Z} \times \{\leq, \geq\}$, and we say that marking $\boldsymbol{x}$ satisfies constraint c if $\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{x} \bowtie \beta$. A *guard* g is then a 4-tuple $(p, t, \beta, \bowtie)$ that imposes upon a transition t the necessary condition that it can only fire in $\boldsymbol{x}$ if the number of tokens in place p satisfies the inequality $x_p(\cdot) \bowtie \beta$. Let

$$\mathbf{1}_i(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \forall (p, t_i, \beta, \bowtie) \in G : x_p(\cdot) \bowtie \beta, \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

If $\mathbf{1}_i(\boldsymbol{x}(n)) = 1$, we say that transition t_i is enabled at time n. If there are no guards $g \in G$ such that $g = (\cdot, t, \cdot, \cdot)$ then the transition t is *always* enabled. Let the total incidence vector $\boldsymbol{u}_i = (u_{i1}, \ldots, u_{i|P|})$ of transition t_i be the vector that describes the effect of firing t_i on the marking. It is defined by $u_{ij} = Post(j,i) - Pre(j,i)$. Then the probability measure governing the marking process $\boldsymbol{X}(n)$ is uniquely characterised by

$$\begin{aligned} \mathbb{P}(\boldsymbol{x}(n) \to \boldsymbol{x}(n+1)) &= \mathbb{P}\left(\boldsymbol{X}(n+1) = \boldsymbol{x}(n+1) \,|\boldsymbol{X}(n) = \boldsymbol{x}(n)\right) \\ &= \frac{\sum_{i \in \mathcal{I}} \lambda_i(\boldsymbol{x}(n))\mathbf{1}_i(\boldsymbol{x}(n))}{\sum_{j=1}^{|T|} \lambda_j(\boldsymbol{x}(n))\mathbf{1}_j(\boldsymbol{x}(n))}, \end{aligned} \tag{2}$$

where $\mathcal{I} = \{i \in \mathbb{N} : t_i \in T,\, \boldsymbol{x}(n+1) = \boldsymbol{x}(n) + \boldsymbol{u}_i\}$.

3.2 Running Example

The running example that we use throughout this paper is a reliability model equivalent to a two-node $M/M/1$ tandem queue. It can be seen as a single component with an infinite number of hot spares; when a component or a spare breaks down, two repair phases have to be completed consecutively. Component and spares fail according to a Poisson process with rate $\lambda = \Theta(\epsilon^2)$. The times between first phase repairs are exponentially distributed with rate $\mu = \Theta(\epsilon)$. The times between second phase repairs are exponentially distributed with rate $\nu = \Theta(1)$. We assume that none of the rates depend on the marking, and that both queues will be empty most of the time. This system can be modelled using an SPN as depicted in Figure 1. The typical rare event that we are interested in is having n or more components awaiting the second phase of repair before all components have been repaired, starting from the first break-down of the main component. This rare event can be cast in the more general framework outlined in Section 3.3.

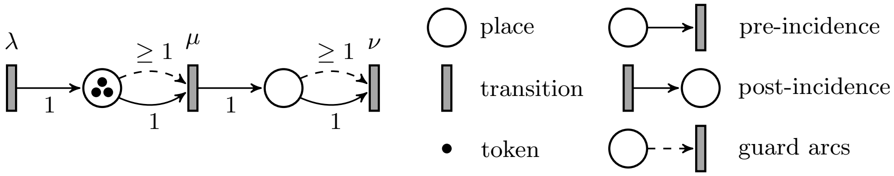

Fig. 1. Tandem queue, depicted in the form of a stochastic Petri net

3.3 Problem Setting

From now on, we will call elements of the taboo set a-markings and elements of the goal set b-markings. We then seek to estimate the probability of reaching a b-marking before reaching an a-marking starting from an initial marking $\boldsymbol{x}_0$. Let $G_a = \{g_a^1, \ldots, g_a^{|G_a|}\} \subset P \times (\mathbb{N} \cup 0) \times \{\leq, \geq\}$ be the set of a-constraints, and let

$$\mathbf{1}_g(\boldsymbol{x}) = \begin{cases} 1 & \text{if } g = (p, c, \bowtie) \text{ and } x_p \bowtie c, \\ 0 & \text{otherwise.} \end{cases}$$

for all $\boldsymbol{x} \in \mathcal{X}$. Let $X \subset \mathcal{X}$ be a a-based hyperrectangle if $\forall g \in G_a$: $\forall \boldsymbol{x} \in X$: $\mathbf{1}_g(\boldsymbol{x}) \equiv c(g, X)$, where $c(g, X) \in \{0, 1\} \forall g \in G_a$. Then let the taboo set $\mathcal{X}_a$ be any union of a-based hyperrectangles. The goal set $\mathcal{X}_b$ are defined similarly for G_b. If a marking is both an a- and b-marking, we will consider it to be a b-marking only. In LTL-notation [3], the event of interest can be written as $\neg a \mathbin{\mathsf{U}} b$; in this paper we will denote the event of interest by $\Psi_{\boldsymbol{x}_0} = \{(\boldsymbol{\omega}_0, \ldots, \boldsymbol{\omega}_m) : m \in \mathbb{N} : \boldsymbol{\omega}_0 = \boldsymbol{x}_0, \boldsymbol{\omega}_m \in \mathcal{X}_b, \boldsymbol{\omega}_k \notin \mathcal{X}_a \ \forall k = 0, \ldots, m-1\}$ in order to emphasise the dependence on the initial state, and denote its probability of interest as $\mathbb{P}(\Psi_{\boldsymbol{x}_0})$.

3.4 Efficient Simulation

We will estimate the probability $\mathbb{P}(\Psi_{\boldsymbol{x}_0})$ using a series of N simulation runs, for some constant $N \in \mathbb{N}$. In each run, we initialise the marking to be $\boldsymbol{x}_0$. We then iteratively fire transitions using the probability measure $\mathbb{P}$ as defined in (2) until we reach an a- or b-marking. When we terminate, we can set $w_i = 1$ if the event $\Psi_{\boldsymbol{x}_0}$ occurred on run i (i.e. if we ended in $\mathcal{X}_b$) and to $w_i = 0$ otherwise, and then obtain the standard Monte Carlo (MC) estimator $\widehat{p}$ for $\mathbb{P}(\Psi_{\boldsymbol{x}_0})$ as

$$\widehat{p}_{\mathbb{P}} = \frac{1}{N} \sum_{i=1}^{N} w_i.$$

A confidence interval for $\widehat{p}$ can be constructed for large N using the Central Limit Theorem [16].

Our focus will be the case where $\mathbb{P}(\Psi_{\boldsymbol{x}_0})$ is small, as this is typically the case in a highly reliable system setting. In this situation, N needs to be very large to obtain a reasonable estimate for $\widehat{p}$. To remedy this, we apply importance sampling.[2] Instead of sampling directly from $\mathbb{P}$, we use a different probability

---

[2] We note here that the distance function d could also be used to construct level sets for RESTART/splitting.

measure $\mathbb{Q}$; after sampling the runs $(\boldsymbol{x}_i(0), \boldsymbol{x}_i(1), \ldots, \boldsymbol{x}_i(n_i))$, $i = 1, \ldots, N$, we use the importance sampling (IS) estimator

$$\widehat{p}_{\mathbb{Q}} = \frac{1}{N} \sum_{i=1}^{N} w_i \prod_{j=0}^{n_i - 1} \frac{\mathbb{P}(\boldsymbol{x}_i(j) \to \boldsymbol{x}_i(j+1))}{\mathbb{Q}(\boldsymbol{x}_i(j) \to \boldsymbol{x}_i(j+1))}. \tag{3}$$

If a suitable new measure $\mathbb{Q}$ is chosen, the number of runs required to obtain a reasonable estimate can be reduced dramatically. The choice of the new measure $\mathbb{Q}$ is non-trivial, however. Typically, good simulation measures $\mathbb{Q}$ increase the likelihood of $\Psi_{\boldsymbol{x}_0}$ occurring, albeit not too strongly. In order to make $\Psi_{\boldsymbol{x}_0}$ more likely, $\mathbb{Q}$ must in some way push the marking in the direction of the goal set and away from the taboo set. The first challenge that arises is then to determine how *far* a marking is from the goal set, so that the simulation $\mathbb{Q}$ can increase the likelihood of moving to a marking with lower distance. For this paper, we define the distance function $d(\boldsymbol{x})$ as

$$d(\boldsymbol{x}) = \min\{r : \exists \omega \in \Psi_{\boldsymbol{x}} \text{ s.t. } \mathbb{P}(\omega) = \Theta(\epsilon^r)\}, \tag{4}$$

where we use the fact that, in essence, the event $\Psi_{\boldsymbol{x}}$ is simply a set of sequences of markings. In words, $d(\boldsymbol{x})$ is the *minimal distance* or *cost* in terms of the ϵ-order of the path from $\boldsymbol{x}$ to the goal set. If the set over which the minimum is taken is empty, we let $d(\boldsymbol{x}) = \infty$. Given $d(\boldsymbol{x})$, we use the following measure $\mathbb{Q}$:

$$\mathbb{Q}(\boldsymbol{x}(j) \to \boldsymbol{x}(j+1)) = \frac{\mathbb{P}(\boldsymbol{x}(j) \to \boldsymbol{x}(j+1))\epsilon^{d(\boldsymbol{x}(j+1))}}{\sum_{\boldsymbol{x}'} \mathbb{P}(\boldsymbol{x}(j) \to \boldsymbol{x}')\epsilon^{d(\boldsymbol{x}')}}. \tag{5}$$

This estimator can be proven to have so-called bounded relative error under some assumptions (more on this in Section 6.2). The remaining problem is then to find $d(\boldsymbol{x})$ for each possible marking $\boldsymbol{x}$. This will be the topic of Section 4.

4 An Algorithm for Determining the Distance Function

In this section we discuss an automated algorithm for finding the function d as defined in (4). The algorithm is executed during a pre-processing phase, before the actual simulation phase starts. Since d is the solution to a shortest path problem in a weighted graph,[3] we could apply Dijkstra's algorithm to find d explicitly for each state (i.e. marking) in the state space $\mathcal{X}$. However, since Dijkstra's algorithm uses the complete state space, it is not better than standard numerical algorithms. Hence, our aim will be to partition $\mathcal{X}$ into *zones* such that for each zone, all the states in this zone have a *similar* cost function d in a sense to be detailed below. Formally, let a zone z be a set of *constraints* $\{c_1^z, \ldots, c_{|z|}^z\}$, as defined in Section 3.1. Let the *zone set* $\mathcal{X}_z$ be the set of states that satisfy

---

[3] Namely one which corresponds to the underlying Markov chain and with the costs of the transitions in terms of ϵ-orders as weights. For another application of Dijkstra's algorithm to finding the most likely paths in a Markov chain, see [9].

all constraints in z. The idea is then to find a set of zones Z such that the sets $\mathcal{X}_z, z \in Z$, form a partition of $\mathcal{X}$ and that we can find functions $d_z(\boldsymbol{x})$ that give an easy expression for the distance to $\mathcal{X}_b$ of all states $\boldsymbol{x} \in z$.

Particularly, we aim to construct a *zone graph*; a graph where the nodes correspond to the zones of Z and in which there is an arc from zones z to z' if for each state $\boldsymbol{x} \in \mathcal{X}_z$ we can reach some state in z' through repeated firing of *a single transition.* We will call such a repeated firing a *stutter step*, as in, e.g., [2]. Furthermore, we want the shortest path from any state in z to $\mathcal{X}_b$ to correspond to the same path through the zone graph. Finally, we want the cost in terms of ϵ-orders of firing the transition of the stutter step to be the same in all states in the same zone set. If all these conditions hold, then for each zone z we can find a function d_z that is the *same* affine function for all $\boldsymbol{x} \in \mathcal{X}_z$ (a function f is affine if $f(\boldsymbol{x}) = \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{x} + \beta$ for some $\boldsymbol{\alpha} \in \mathbb{R}^{|P|}$ and $\beta \in \mathbb{R}$). In this section, we will clarify how this can be done.

To make the preceding concrete, consider the running example. The first two zone sets that we create are $\mathcal{X}_a$ *and* $\mathcal{X}_b$*; in particular,* $\mathcal{X}_b$ *consists of the states in which* $x_2 \geq n$*; we assume* $n \geq 3$*. In the state* $(1, n-1)$*,* t_2 *needs to fire once to reach* $\mathcal{X}_b$*, and the distance of this step is* 1 *(because* t_2 *needs to 'win the race' from* t_3*, which fires* ϵ^{-1} *times faster). In* $(2, n-2)$*, we need to fire* t_2 *twice, giving a total distance of* 2*. The same holds for all states* $(x, n-x)$*,* $x \geq 1$*; we fire* t_2 x *times and the total distance is* x*. It then makes sense to group all these states together in a zone set. However, for* $(n, 0)$*, we need to fire* t_2 n *times, but the total cost is* $n-1$ *as* t_2 *does not need to compete against* t_3 *in the first step. Hence,* $(n, 0)$ *and* $(n-1, 1)$ *will not be in the same zone. The complete set of zones, with their distance functions and shortest paths to the taboo set, is illustrated in Figure 2.*

Algorithm 1. Main loop.

```
initZoneGraph()
while $S \neq \emptyset$ do
    $s = (z^o, t_i, z^d)$ := some element from $S$
    possibilitySplit($s$)
    if $d_{z^o} \neq$ unassigned then costSplit($s$)
    update($s$); $S := S \backslash s$
end while
```

In Section 4.1, we will outline the main algorithm. As in the previous example, an initial partitioning is always necessary, as we will discuss in Section 4.2. However, this initialisation alone is not sufficient. It may be that it is not possible for all markings in an initial zone to reach another zone by the same stutter step; this is the topic of Section 4.3. Also, there may exist markings within a single zone for which the shortest path follows a different sequence of stutter steps; more on that in Section 4.4.

4.1 Main Loop

Let a *stutter step* s be a triple (z^o, t_i, z^d), where z^o is the source/origin zone, z^d is the destination zone and t_i is the transition that is repeatedly fired. The algorithm works as follows: we keep a list S of stutter steps that could be part of shortest paths. After initialising the list, we repeatedly take stutter steps s out of S and check whether for all markings in the origin zone of s it holds that

1) we can indeed reach the destination zone of s using only the given stutter step, and
2) the new distance function indeed gives shorter distance than what was known before.

If not, we split up the source zone and (potentially) add new stutter steps to S. Finally, we discard s, pick a new stutter step, and repeat until S is empty. The precise way in which this is done is given by Algorithm 1.

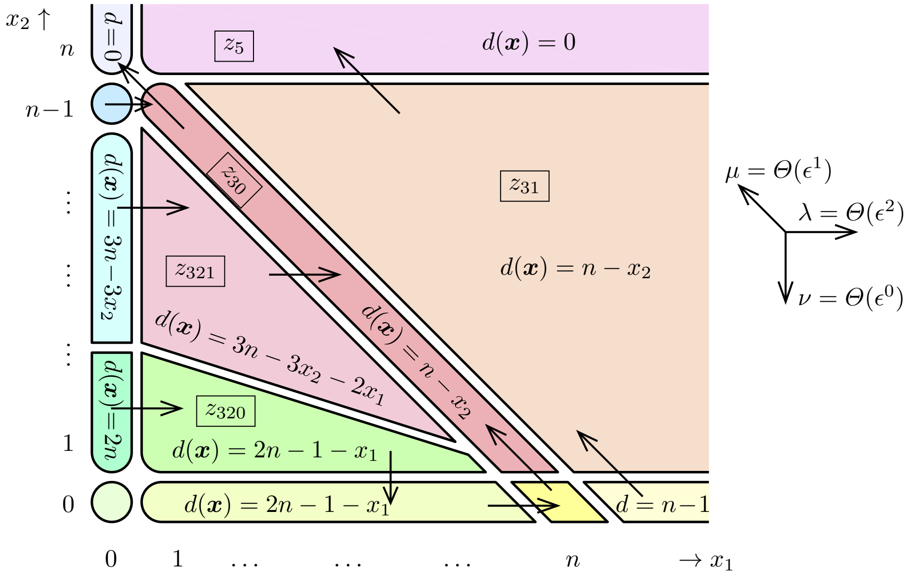

Fig. 2. The final result of a call to the algorithm, excluding lines 2 and 3 of `initZoneGraph()`

4.2 Initialisation Phase (`initZoneGraph()`)

During the initialisation phase, the state space is divided into zones such that

a) from all states in the same zone set the same transitions are enabled, and
b) all states in a zone set are either in $\mathcal{X}_a$, all in $\mathcal{X}_b$ or all in neither.

Condition a) implies that the cost of firing a transition is always the same in a zone (because the cost depends on which other transitions can be fired). During the initialisation we can already assign distance ∞ to the states in $\mathcal{X}_a$ and 0 to the states in $\mathcal{X}_b$. Furthermore, we initialise the stutter step list S during this phase; its initial elements will be those stutter steps that directly lead into $\mathcal{X}_b$. The precise way in which all this is done is given in Algorithm 2.

Lines 2 and 3 deal with a technical obstacle; when for a stutter step (z^o, t_i, z^d) it holds that $z^o = z^d$, line 1 of `possibilitySplit()` will fail. However, we cannot exclude these 'self-loops' in the zone graph; there exist cases in which the shortest path moves to the edge of an initial zone without crossing it. To remedy this, we also create 'edges' around the initial zones of line 1.

In line 4 of Algorithm 2, we use the negation $\neg c$ of a constraint c. If $c = (\alpha, \beta, \leq)$, then its negation is given by $\neg c = (\alpha, \beta + 1, \geq)$, and if $c = (\alpha, \beta, \geq)$ then $\neg c = (\alpha, \beta - 1, \leq)$. If all elements of α are at least 1, then the resulting zone sets $\mathcal{X}_{\{c\}}$ and $\mathcal{X}_{\{\neg c\}}$ are each other's complements with respect to $\mathcal{X}$.

Algorithm 2. initZoneGraph()

```
1: $C' := \{c = (p, \beta, \bowtie) : (p, \cdot, \beta, \bowtie) \in G \vee c \in G_a \cup G_b\}$
2: $u^{\max} := \max_{i=1,\ldots,|T|} \max_{k=1,\ldots,|P|} |u_{ik}|$
3: $C := \{c = (p, \beta, \bowtie) : (p, \beta + k, \bowtie) \in C', k \in \mathbb{Z}, |k| \leq u^{\max}\}$
4: $Z := \{z \in \mathcal{Z} : \forall c \in C : c \in z \vee \neg c \in z, \mathcal{X}_z \neq \emptyset\}$    ▷ $\mathcal{Z}$ = set of all zones
5: $V := \{(z^o, t_i, z^d) : \exists \boldsymbol{x} \in \mathcal{X} : \boldsymbol{x} \in \mathcal{X}_{z^o}, \boldsymbol{x} + \boldsymbol{u}_i \in \mathcal{X}_{z^d}\}$
6: $Z_a := \{z \in Z : \forall \boldsymbol{x} \in \mathcal{X}_z : \boldsymbol{x} \in \mathcal{X}_a\}$
7: $\forall z \in Z_a : d_z := \infty$
8: $Z_b := \{z \in Z : \forall \boldsymbol{x} \in \mathcal{X}_z : \boldsymbol{x} \in \mathcal{X}_b\}$
9: $\forall z \in Z_b : d_z = 0$
10: $S := \{v \in V : v = (z, \cdot, z'), z \notin Z_a, z' \in Z_b\}$
```

For the running example as displayed in Figure 1, the transition structure first gives us four initial zones: z_0 *where only* t_1 *can fire,* z_1 *for* t_1 *and* t_2*,* z_2 *for* t_1 *and* t_3*, and* z_3 *for all three. The zone structure resulting from a call to initZoneGraph() is displayed in Figure 3(a). In fact, for the running example the algorithm would also work well if we would not include margins, i.e. omit lines 2 and 3, resulting in Figure 3(b). For the sake of clarity, we will continue based on the latter, even though our implementation does include the margins. We get two additional zones,* z_4 *and* z_5*, to distinguish* $\mathcal{X}_b$*.* S *is initialised with all stutter steps leading into these two zones; the only stutter steps satisfying this requirement are the two* t_2*-stutter steps going from* z_3 *into* z_4 *and* z_5*.*

4.3 Divide Zones According to Possibility of Firing (`possibilitySplit()`)

To determine the cost of a stutter step $s = (z^o, t_i, z^d)$, we need to determine the number of times y that t_i must fire to take a marking in z^o to z^d. This is done by `findNumberOfTransitions()`. The main idea is to find a function $y(\boldsymbol{x})$ (written as y for brevity) such that after firing t_i $y - 1$ times, the marking is still in z^o, and after firing one more time the marking is in z^d. In order to find this number, we choose any constraint c_1 from z^o and c_2 from z^d that exclude each other, i.e., $\mathcal{X}_{z^o} \cap \mathcal{X}_{z^d} = \emptyset$, and chooses y to be the smallest number of firings to enable c_2. Since all constraints are non-strict inequalities, y is chosen such that $\boldsymbol{x} + y\boldsymbol{u}_i$ exactly satisfies the constraint. The remaining constraints in z^o and z^d then impose restrictions on $\boldsymbol{x}$ that must be satisfied in order for this stutter step to be carried out.

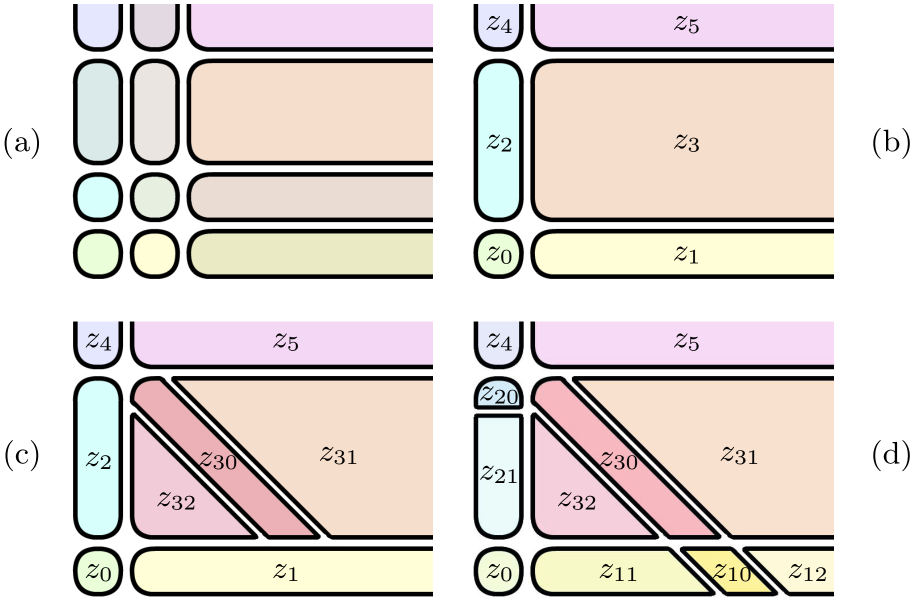

Fig. 3. Figure (a) illustrates the result of a call to `initZoneGraph()` when lines 2 and 3 are included (we only show the margins around the axes). Figures (b-d) depict the zones after several iterations of the algorithm, without lines 2 and 3 of `initZoneGraph()`.

Algorithm 3. possibilitySplit().

```
Require: stutter step $s$
$(c_1, c_2) :=$ some two constraints such that
        1) $c_1 \in z^o$,  2) $c_2 \in z^d$ and   3) $\mathcal{X}_{\{c_1\}} \cap \mathcal{X}_{\{c_2\}} = \emptyset$
$y :=$ findNumberOfTransitions($c_2$,$\boldsymbol{u}_i$)
$C_1 := \{c : c = \boldsymbol{a}(\boldsymbol{x} + (y-1)\boldsymbol{u}_i) \bowtie b \ \wedge \ \boldsymbol{ax} \bowtie b \in z^o \backslash c_1\}$
$C_2 := \{c : c = \boldsymbol{a}(\boldsymbol{x} + y\boldsymbol{u}_i) \bowtie b \ \wedge \ \boldsymbol{ax} \bowtie b \in z^d \backslash c_2\}$
$C := C_1 \cup C_2$
$Z_{\text{new}} := \{z : \forall c \in C : c \in z \vee \neg c \in z \ \wedge \ \forall c \in z^o : c \in z \ \wedge \ \exists \boldsymbol{x} \in \mathcal{X} : \boldsymbol{x} \in \mathcal{X}_z\}$
$z^n := z \in Z_{\text{new}} : \forall c \in C : c \in z$
$d_{z^n}(\boldsymbol{x}) := d_{z^d}(\boldsymbol{x} + y\boldsymbol{u}_i) + y\kappa_i(\boldsymbol{x})$        ▷ where $\kappa_i(\boldsymbol{x}) = \frac{\mathbf{1}_i(\boldsymbol{x})r_i}{\sum_{j=1}^{|T|} \mathbf{1}_j(\boldsymbol{x})r_j}$
```

Assume that we happen to first consider the μ-stutter step from z_3 to z_4. After the initialisation phase, there are two pairs of constraints from z_3 and z_4 that exclude each other; the pair $x_1 \geq 1$ and $x_1 \leq 0$, and the pair $x_2 \leq n-1$ and $x_2 \geq n$. If we consider the first pair, we end up with $y = x_1$. The two constraints that we end up through lines 4 and 5 of Algorithm 3 are $x_1 + x_2 - 1 \leq n - 1$ and $x_1 + x_2 \geq n$. If we would consider the second pair, we would have found $y = n - x_2$, leading to the same restrictions on $x_1 + x_2$.

Given the set C of constraints that must be satisfied for the stutter step s to be taken, the zone z^o may need to be subdivided such that one zone remains in which the stutter step s is always possible. This is done in line 6 of Algorithm 3; all zones that consist of combinations of constraints in C or their negations are considered. If such a zone is non-empty (which is checked using an Integer Linear Programming-solver, although this can be computationally expensive), it is added to Z_{new}, the set of new zones. The zone z^n is the subzone (i.e. a subset in terms of constraints) of z^o for which s was possible.

Since we obtained the additional constraints $x_1 + x_2 \leq n$ and $x_1 + x_2 \geq n$ for the running example, we obtain three new non-empty zones; z_{30}, z_{31} and z_{32}, all depicted in Figure 3(c). Of those, z_{30} has cost $d_{z_{30}}(\boldsymbol{x}) = x_1$ or, equivalently, $d_{z_{30}}(\boldsymbol{x}) = n - x_2$, depending on which of the two constraint pairs was considered. The other two zones do not have any cost assigned yet. When the function `update()` *in Algorithm 1 is called, the stutter steps from z_1, z_2, z_{31} and z_{32} to z_{30} are added to S. Furthermore, the stutter step from z_3 to z_5 is removed, as z_3 no longer exists. It is replaced by the μ-stutter step from z_{31} to z_5.*

Upon further calls to `possibilitySplit()`*, the zone z_1 is subdivided into three new zones and z_2 into two new zones, and distance functions are assigned to all. This is displayed in Figure 3(d). Furthermore, zones z_{31} and z_{32} have distance assigned to them. In particular, we mention the distance function of z_{32}: $d_{z_{32}}(\boldsymbol{x}) = 3n - 2x_1 - 3x_2$. In the next section, $z_{32}$is split into two zones, only one of which retains this distance function.*

Algorithm 4. costSplit()

```
Require: step $s$
$c_n := d_{z^n}(\boldsymbol{x}) - d_{z^d}(\boldsymbol{x}) < 0$
$z' := z^n \cup \{c_n\}$
$z'' := z^n \cup \{\neg c_n\}$
$d_{z'}(\boldsymbol{x}) := d_{z^n}(\boldsymbol{x})$
$d_{z''}(\boldsymbol{x}) := d_{z^d}(\boldsymbol{x})$
if $\exists \boldsymbol{x} \in \mathcal{X} : \boldsymbol{x} \in \mathcal{X}_{z'}$ then
    if $\nexists \boldsymbol{x} \in \mathcal{X} : \boldsymbol{x} \in \mathcal{X}_{z''}$ then
        $Z_{\text{new}} := Z_{\text{new}} \backslash z^n \cup z'$
    else
        $Z_{\text{new}} := Z_{\text{new}} \backslash z^n \cup z' \cup z''$
    end if
end if
```

4.4 Divide Zones According to Costs (`costSplit()`)

When the algorithm as described so far is executed, it will consecutively consider zones to which no distance function has yet been assigned yet, split them and assign costs to them. However, when a zone is considered that *already has* a distance function assigned to it, the new path may be the shortest only for a subset of the zone. We need `costSplit()` for these situations.

Say that, after running Algorithm 3, one has found a subzone z^n of z^o for which the stutter step under consideration can be applied, and for which

d_{z^n} is the distance function. If d_{z^o} has already been assigned, then the stutter step under consideration is only interesting for those markings $\boldsymbol{x}$ for which $d_{z^n}(\boldsymbol{x}) < d_{z^o}(\boldsymbol{x})$. This constraint is exactly the one constructed in line 1. The zone z^n is then divided into two new zones: z', for which this constraint holds, and z'', for which it does not. If z' is empty, the stutter step under consideration has been irrelevant, and the list S should not be updated. If only z'' is empty, then z' fully replaces z^n. However, if both z' and z'' are non-empty, the two of them are added to Z_{new} instead of z^n.

For the running example, the distance function $d_{z_{32}}(\boldsymbol{x}) = 3n - 2x_0 - 3x_1$ *had already been assigned to the zone* z_{32} *as depicted in Figure 3(d). Assume that the next stutter step to be considered is the* t_3*-stutter step from* z_{32} *to* z_{11}*. Since the distance function in* z_{11} *is* $2n - 1 - x_1$*, and the cost of firing* t_3 *in* z_{32} *is zero, the new zones* z_{320} *and* z_{321} *are separated by the line* $3x_2 \leq n - x_1$*. In the next and final iteration* z_{21} *is further divided into the zones* z_{210} *and* z_{211} *by* `possibilitySplit()`*.*

5 Empirical Results

We present numerical results obtained using the algorithm to find d in Section 5.1, while in Section 5.2 we use d to apply simulation.

Case Description. We use two case studies. The first is the running example from Section 3.2, where the system is failed if $x_2 > n$, $n \in \mathbb{N}$. The second is a more realistic multicomponent system with interdependent component types, taken from [22]. For the latter we have six component types, with n_i components of type i and $(n_1, \ldots, n_6) = (n+2, n+1, n+3, n, n+4, n+2)$. In the benchmark setting, $n = 3$. If k components of type i have failed, the rate at which the next component of type i fails is $(n_i - k)\lambda_i\epsilon$, where $(\lambda_1, \ldots, \lambda_6) = (2.5, 1, 5, 3, 1, 5)$. There is a single repairman who repairs components following a preemptive priority repair strategy, where components of type i have priority over components of type j if $i < j$. The repair rate for type i is always μ_i, $(\mu_1, \ldots, \mu_6) = (1, 1.5, 1, 2, 1, 1.5)$. The system is said to have failed when all components of any type are down. We estimate the probability that, after the first component failure (drawn randomly), the system fails before all components are repaired.

5.1 Results of the Distance Finding Algorithm

A summary of the results of our algorithm is displayed in Table 1. The number of initial constraints is the main factor that determines the runtime of the algorithm. For the initial zones, we distinguish between the $(a \cup b)$- and $\neg(a \cup b)$-zones because only the latter have an impact on the runtime of the rest of the algorithm. A few things to mention: the number of zones may depend on n because for small n some zones will be empty, which are discarded. Also, the final number of zones may depend on the way stutter steps are chosen from S in the main loop, because if a zone is split by a stutter step that later turns out to be insignificant,

Table 1. Results of the numerical analysis for the running example

	Running Example		Multicomponent System	
n	3	10	3	5
# initial constraints	5	5	18	18
# initial zones	15	18	38880	46656
# initial $\neg(a \cup b)$-zones	8	11	3071	4095
# final $\neg(a \cup b)$-zones	14	27	3557	5477
# iterations in main loop	57	114	26421	42189
# markings in $\mathcal{X}$	∞	∞	40320	241920
time to construct (sec)	1.77	1.78	41.38	194.79

these zones are not recombined by our implementation, so both the number of zones and the number of iterations are implementation-dependent. For the multicomponent system, for small n the number of zones is almost equal to the size of the state space. This is a (for this case study unnecessary) consequence of the margins defined in lines 2 and 3 of `initZoneGraph()`.

5.2 Simulation Results

The simulation results are summarised in Tables 2 and 3. In both tables, we display the results for three simulation methods: standard Monte Carlo (MC), importance sampling (IS) using Balanced Failure Biasing (BFB) and IS based on our distance finding algorithm (Zone-IS). Under BFB, the total probability of firing a failure transition is set to $\frac{1}{2}$, uniformly distributed over the individual failure transitions (and similarly for the repairs — for more information, see [25]). In our implementation, we only consider the ν-transition t_3 to be a repair transition. Next to the simulation results, we display numerical approximations obtained using the model checking tool PRISM [15].

For the efficiency of the methods we look at the *relative error* (r. error) of the estimates, defined as the ratio of the estimator's standard deviation to the estimate. A lower value generally means a better estimate; however, if a change of measure is poorly suited for the system, IS may suffer from underestimation [7]. An example of this are the results for BFB for $n = 10$ and $\epsilon = 0.01$ in Table 2. For the sake of consistency with [22], we used 200 000 000 runs per MC-estimate and 10 000 000 runs per IS-estimate. In all cases Zone-IS outperforms BFB, except for $n = 5$ in Table 3. The reason is that BFB needs a clear distinction between failures and repairs to work well.

6 Discussion and Conclusions

6.1 Conclusions

We have presented a novel method to automatically construct a change of measure for speeding up the simulation of rare events in stochastic Petri nets. Our

Table 2. Results of the simulation analysis for the running example

		MC		BFB		Zone-IS		PRISM
n	ϵ	$\widehat{p}$	r. error	$\widehat{p}$	r. error	$\widehat{p}$	r. error	$\widehat{p}$
3	10^{-1}	$1.11\cdot10^{-4}$	0.007	$1.096\cdot10^{-4}$	0.007	$1.100\cdot10^{-4}$	$6.31\cdot10^{-4}$	$1.100\cdot10^{-4}$
	10^{-2}	$1.50\cdot10^{-8}$	0.577	$1.007\cdot10^{-8}$	0.011	$1.010\cdot10^{-8}$	$2.21\cdot10^{-4}$	$1.010\cdot10^{-8}$
	10^{-3}	—	—	$1.026\cdot10^{-12}$	0.011	$1.001\cdot10^{-12}$	$7.16\cdot10^{-5}$	$1.001\cdot10^{-12}$
	10^{-4}	—	—	$1.003\cdot10^{-16}$	0.011	$1.000\cdot10^{-16}$	$2.39\cdot10^{-5}$	$1.000\cdot10^{-16}$
5	10^{-1}	$1.00\cdot10^{-8}$	0.707	$1.140\cdot10^{-8}$	0.040	$1.098\cdot10^{-8}$	0.001	$1.100\cdot10^{-8}$
	10^{-2}	—	—	$9.843\cdot10^{-17}$	0.083	$1.010\cdot10^{-16}$	$5.09\cdot10^{-4}$	$1.010\cdot10^{-16}$
10	10^{-1}	—	—	$1.638\cdot10^{-18}$	0.970	$1.109\cdot10^{-18}$	0.006	$1.100\cdot10^{-18}$
	10^{-2}	—	—	$3.144\cdot10^{-42}$	0.865	$1.017\cdot10^{-36}$	0.003	$1.010\cdot10^{-36}$

Table 3. Results of the simulation analysis for the multicomponent system

		MC		BFB		Zone-IS	
n	ϵ	$\widehat{p}$	r. error	$\widehat{p}$	r. error	$\widehat{p}$	r. error
3	10^{-3}	$7.25\cdot10^{-7}$	0.083	$7.535\cdot10^{-7}$	0.019	$7.283\cdot10^{-7}$	0.007
	10^{-4}	$1.0\cdot10^{-8}$	0.707	$4.815\cdot10^{-9}$	0.027	$4.861\cdot10^{-8}$	0.002
5	10^{-3}	—	—	$1.155\cdot10^{-10}$	0.123	$4.368\cdot10^{-11}$	0.288
	10^{-4}	—	—	$1.901\cdot10^{-15}$	0.288	$1.381\cdot10^{-15}$	0.351

approach uniquely combines two characteristics: it uses a *high-level description of the model* with much flexibility and expressivity (a Petri net) and it works *without generating the entire state-space.*

The heart of our method is an algorithm which automatically partitions the state-space into a collection of zones. Each zone comprises states in which the same so-called change of measure is needed in the rare-event simulation scheme. The zones are demarcated by a set of affine inequalities, thus avoiding enumeration of all states. The number of zones in typical models does not need to increase as the model's size increases.

We have demonstrated that our algorithm works well in two examples. More experimentation will be needed to fully understand its possibilities and limitations and to optimise the implementation, and some extensions of the algorithm may be needed to handle certain classes of models (see below).

6.2 Discussion

In order to mathematically prove that the method always performs well, it remains to deal with three issues. The first is the correctness of the algorithm; i.e., whether the returned distance function really satisfies the definition in (4). The second is *termination* of the algorithm within finite time. The third is the *efficiency* of the resulting importance sampling estimator. The first issue can be dealt with using a suitable invariant statement. For the latter two, we give a short discussion.

Termination. If the state space is infinite, it is possible that the (current) algorithm will not terminate. For example, if transition t_1 takes the system closer to the goal states and enables a transition t_2 with a very high firing rate, but firing the t_2 disables itself and does not negate the firing of t_1, then a shortest path might alternate between firing t_1 and t_2. This may result in the algorithm constructing an infinite number of zones. A possible solution is to broaden the concept of a stutter step. If a shortest path alternates between a tuple of transitions, the repeated firing of this tuple could be seen as a stutter step in itself, and the sum of the incidence vectors of the individual transitions as the net effect on the marking. Under such a restriction, the space of zones could well be bounded; this is part of ongoing research.

Importance Sampling Efficiency. The importance sampling measure as defined in (5) is inspired by the change of measure proposed in [17], where also the notion of *bounded relative error* comes up. This notion says that as ϵ approaches 0, the ratio of the standard deviation of the estimator to the standard mean remains bounded. This is desirable: since the accuracy of a simulation result is directly linked to this relative error, this means that the time to reach some level of accuracy never crosses a certain threshold value as ϵ becomes smaller. This behaviour is observed in Table 2 of Section 5, so we believe that our method will have bounded relative error, after a slight refinement.

The authors of [17] show that bounded relative error is guaranteed in their setting under the assumption that the state space is finite and that no *high-probability cycles* exist. Essentially, these assumptions imply that the number of paths ω with $\mathbb{P}(\omega) = \Theta(\epsilon^{d(\boldsymbol{x})})$ is finite. If this does not hold, it may be that $\mathbb{P}(\Psi_{\boldsymbol{x}}) \neq \Theta(\epsilon^{d(\boldsymbol{x})})$. A possible remedy would then be to perform a loop-detection algorithm on the initial graph returned by Algorithm 2 in order to detect the high-probability cycles, and remove them. This is also part of ongoing research.

References

1. Ajmone Marsan, M., Balbo, G., Donatelli, S., Franceschinis, G., Conte, G.: Modelling with generalized stochastic Petri nets. John Wiley & Sons, Inc. (1994)
2. Baier, C., D'Argenio, P., Groesser, M.: Partial order reduction for probabilistic branching time. Electronic Notes in Theoretical Computer Science (2006)
3. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
4. Blum, A.M., Goyal, A., Heidelberger, P., Lavenberg, S.S., Nakayama, M.K., Shahabuddin, P.: Modeling and analysis of system dependability using the system availability estimator. In: Twenty-Fourth International Symposium on Fault-Tolerant Computing, pp. 137–141. IEEE (1994)
5. Carrasco, J.A.: Failure distance based simulation of repairable fault-tolerant systems. In: Proceedings of the 5th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, pp. 351–365 (1992)
6. Clark, G., Courtney, T., Daly, D., Deavours, D., Derisavi, S., Doyle, J.M., Sanders, W.H., Webster, P.: The Möbius modeling tool. In: Proceedings of the 9th International Workshop on Petri Nets and Performance Models. IEEE (2001)

7. Devetsikiotis, M., Townsend, J.K.: An algorithmic approach to the optimization of importance sampling parameters in digital communication system simulation. IEEE Transactions on Communications 41(10), 1464–1473 (1993)
8. Glasserman, P., Heidelberger, P., Shahabuddin, P., Zajic, T.: Multilevel splitting for estimating rare event probabilities. Operations Research 47(4), 585–600 (1999)
9. Han, T., Katoen, J.-P.: Counterexamples in probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 72–86. Springer, Heidelberg (2007)
10. Heidelberger, P.: Fast simulation of rare events in queueing and reliability models. In: Donatiello, L., Nelson, R. (eds.) SIGMETRICS 1993 and Performance 1993. LNCS, vol. 729, pp. 165–202. Springer, Heidelberg (1993)
11. Jegourel, C., Legay, A., Sedwards, S.: Cross-entropy optimisation of importance sampling parameters for statistical model checking. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 327–342. Springer, Heidelberg (2012)
12. Jegourel, C., Legay, A., Sedwards, S.: A platform for high performance statistical model checking – PLASMA. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 498–503. Springer, Heidelberg (2012)
13. Júlvez, J.: Basic qualitative properties of Petri nets with multi-guarded transitions. In: American Control Conference, ACC 2009. IEEE (2009)
14. Kelling, C.: A framework for rare event simulation of stochastic Petri nets using "RESTART". In: Proceedings of the 28th Winter Simulation Conference, pp. 317–324. IEEE Computer Society (1996)
15. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002)
16. Law, A., Kelton, W.: Simulation modeling and analysis. McGraw-Hill, New York (1991)
17. L'Ecuyer, P., Tuffin, B.: Approximating zero-variance importance sampling in a reliability setting. Annals of Operations Research 189(1), 277–297 (2011)
18. Miretskiy, D., Scheinhardt, W., Mandjes, M.: On efficiency of multilevel splitting. Communications in Statistics – Simulation and Computation 41(6), 890–904 (2012)
19. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
20. Nicola, V., Shahabuddin, P., Nakayama, M.: Techniques for fast simulation of models of highly dependable systems. IEEE Transactions on Reliability 50(3), 246–264 (2001)
21. Obal, W., Sanders, W.: An environment for importance sampling based on stochastic activity networks. In: Proceedings of the 13th Symposium on Reliable Distributed Systems, pp. 64–73. IEEE (1994)
22. Ridder, A.: Importance sampling simulations of Markovian reliability systems using cross-entropy. Annals of Operations Research 134(1), 119–136 (2005)
23. Rubinstein, R., Kroese, D.: The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning. Springer (2004)
24. Sanders, W.H., Meyer, J.F.: Stochastic activity networks: Formal definitions and concepts. In: Brinksma, E., Hermanns, H., Katoen, J.-P. (eds.) FMPA 2000. LNCS, vol. 2090, pp. 315–343. Springer, Heidelberg (2001)
25. Shahabuddin, P.: Importance sampling for the simulation of highly reliable Markovian systems. Management Science 40(3), 333–352 (1994)

26. Tuffin, B., Trivedi, K.S.: Implementation of importance splitting techniques in stochastic Petri net package. In: Haverkort, B.R., Bohnenkamp, H.C., Smith, C.U. (eds.) TOOLS 2000. LNCS, vol. 1786, pp. 216–229. Springer, Heidelberg (2000)
27. Villén-Altamirano, M., Villén-Altamirano, J.: RESTART: A method for accelerating rare event simulations. In: Queueing, Performance and Control in ATM, pp. 71–76. Elsevier Science Publishers (1991)
28. Zimmermann, A., Freiheit, J., German, R., Hommel, G.: Petri net modelling and performability evaluation with TimeNET 3.0. In: Haverkort, B.R., Bohnenkamp, H.C., Smith, C.U. (eds.) TOOLS 2000. LNCS, vol. 1786, pp. 188–202. Springer, Heidelberg (2000)